Getting started with CYPEX 2.1

Build applications faster

Created by the
CYPEX development team

2022-10-12

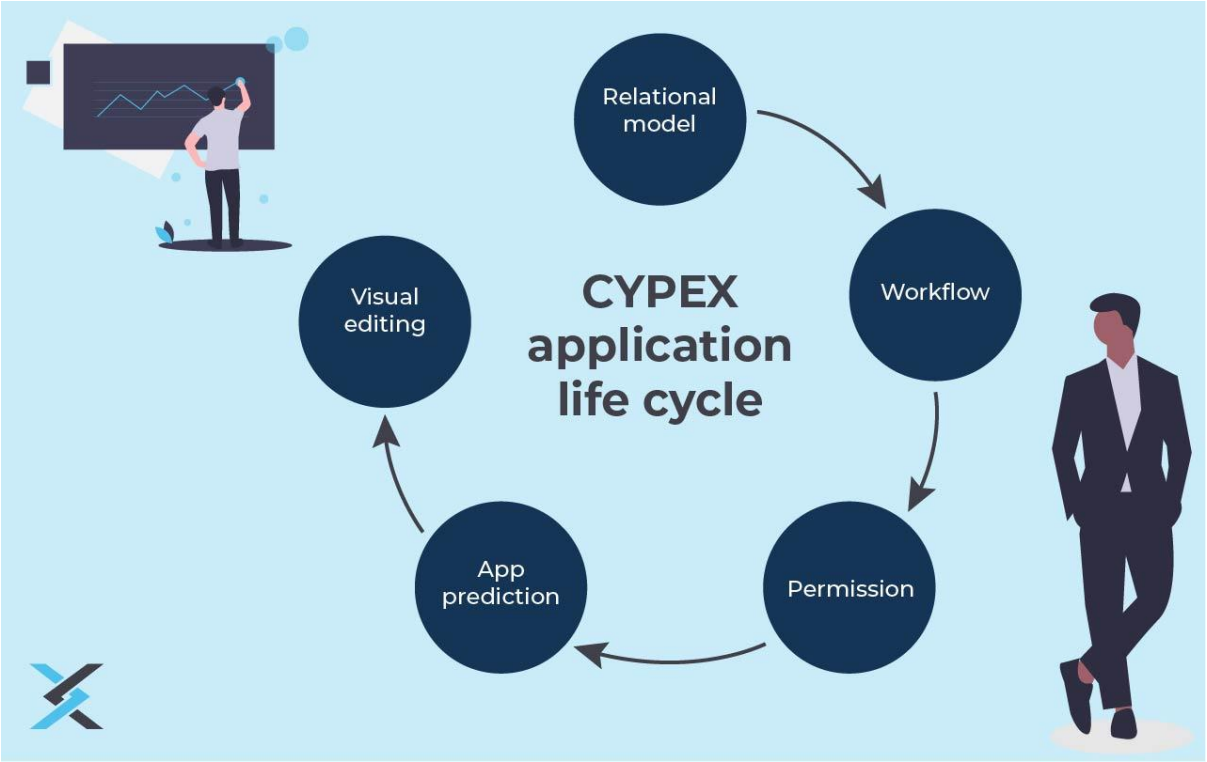
Table of contents

CYPEX: Basic concepts

In this chapter, you'll learn about important basic concepts needed to understand the use of CYPEX. You'll briefly be guided through these essential concepts. Once you're equipped with some basic understanding, we'll build a few applications, guiding you to success step-by-step.

The life-cycle of a CYPEX application

The first thing to understand is the life-cycle of a CYPEX application. Here is a primary overview of how things work:



The life of a CYPEX application begins by creating a **relational model**. This model will form the basis of every application. You can start by deploying your tables as usual.

After that, it's time to enter the world of CYPEX: Go to the **model builder** and organize your relational model (more on how to do that in "**CYPEX terminology and samples**"). Once the workflow and permissions have been added, you can predict a default application with CYPEX.

This basic predicted application is fully functional and can be shown to the end user. You can gather feedback to catch some potential misunderstandings early on in the customer communications process. Your default application can now be adjusted to the client's needs by adding charts, changing the layout, or making other minor changes.

Keep in mind that your entire CYPEX application basically consists of a set of configuration tables which is used to compile a **JSON** document. Your browser will get this JSON document and render it. The advantage of this approach is that your entire application can be transferred from one PostgreSQL database to another, using a simple dump / restore. There are no external dependencies.

If you're using standard PostgreSQL replication, your CYPEX application will be replicated just like any other data. No additional backup processes are needed.

CYPEX terminology and samples

This section will describe the most important aspects of CYPEX terminology.

CYPEX “entities”

In CYPEX you use existing SQL models to build new and powerful applications. The first term we need to discuss is the idea of an “entity”. We use the same semantics as in a standard relational model: *an entity is a table which is tracked by CYPEX*. All tracked entities will be part of the **GUI** prediction which is made based on the data structure. If an entity isn't tracked by CYPEX, it won’t be included in that prediction.

It’s important to understand that you do not work directly with entities. You use an **abstraction layer** between entities and what CYPEX sees.

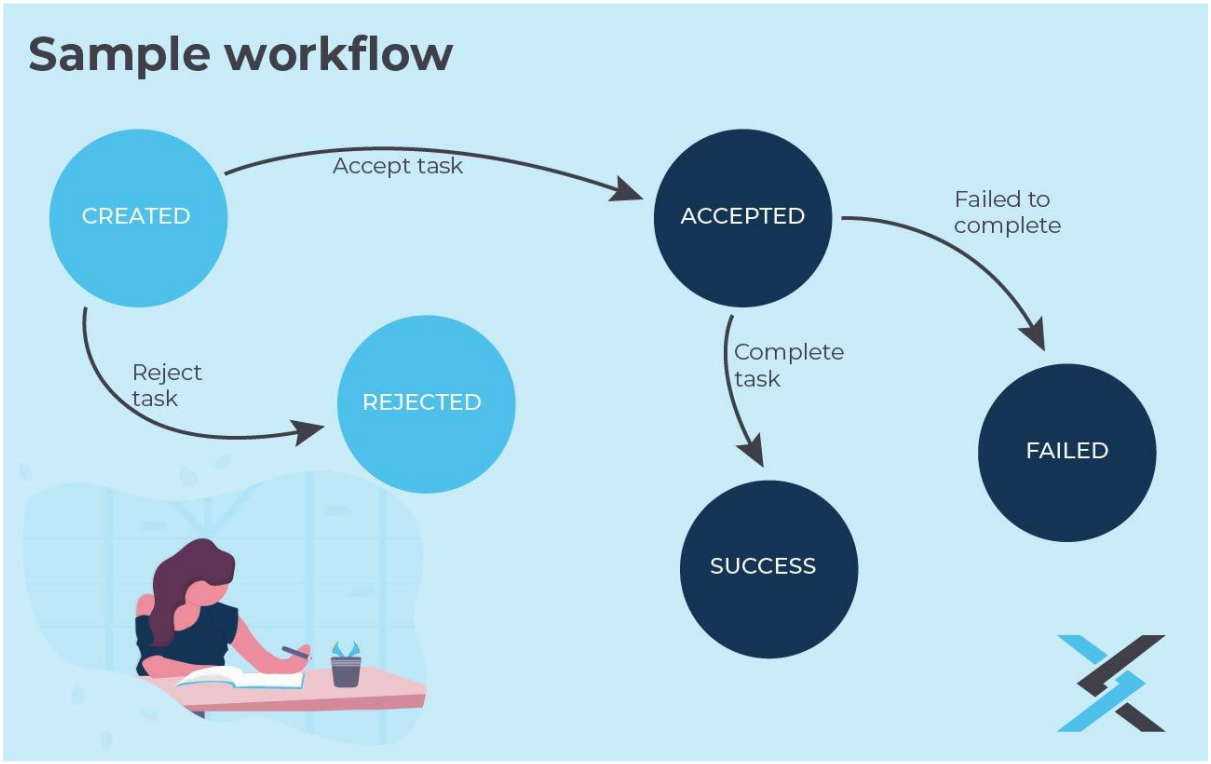
CYPEX “queries”

As already stated, entities are essentially tables in a relational model. However, this isn't what you work with in the CYPEX GUI. Usually, a table doesn’t contain data the way you need it in the GUI. You need to define a query which will be in charge of fetching data from the table and then sending data to the end user. A query can be a subset of columns, a join or any other complex SQL statement needed to pre-preprocess data. In short: A query processes the data so that it can be shown in the GUI.

States and state changes

Workflows are the next step once the initial relational model has been built. The following terms are relevant:

- State columns
- States
- State changes



Let’s take a look at an example: An offer might follow a typical workflow. It's created, edited, sent to the client and, hopefully, signed. An object has an optional “state column” which is only allowed to contain valid state entries (in our case “accepted”, “rejected”, “sent” and “created”). Changing between 2 states is what is called a “state change” - it's any kind of action associated with an object.

Please note that states and state changes occur on the entity - not on the query - level. States are deeply associated with the underlying database model.

States and state changes can be either enforced or non-enforced. In case of enforced state changes, CYPEX will create triggers on the underlying tables to make sure that only valid changes can be made. Usually, enforced state changes should be chosen because they make sure that a data model cannot contain faulty data. However, in some cases it might also make sense to work with states that aren't enforced by CYPEX. This is especially true if the underlying data model must not be modified for some reason.

An entity may have either no state column, or one state column. Combined or multiple state columns aren't supported.

Database permissions

Database permissions are of great importance and are usually assigned on the query level to ensure that tables remain mostly unchanged.

CYPEX offers visual tooling to define these query permissions. However, setting permissions might not be enough. PostgreSQL supports “Row Level Security” which is an easy way to filter rows: A table might contain 1 million people (500.000 women and 500.000 men). User A might only be allowed to see women while user B is only allowed to see men. Depending on who you are, PostgreSQL will only return the subset of data you are permitted to see. Row-Level-Security can therefore be seen as a kind of mandatory filter added for a user. **In case you are using RLS (= Row Level Security), make sure that your policy is assigned to PUBLIC rather than to a normal user.** The reason for this is that a view will only honour an RLS policy if it's assigned to “public”. This is because views in PostgreSQL are basically a separate security context. Assigning policies to the wrong entity on PostgreSQL is a fairly common mistake.

Permissions heavily impact default rendering. In CYPEX a GUI is created for a user or a group of users. In case a group of users does not have access to a query, the GUI won’t contain those elements at all. In other words: Permissions drive the way default rendering is done at the most basic level. This also implies that **2 people accessing the same database might see totally different applications, depending on their permissions and security settings**.

Writing clever relational models

As previously stated, a relational model is the foundation of every CYPEX application. However, not all relational models are created equal. Some are more suitable for application prediction than others. In this section, you’ll learn more about how to write suitable relational models and what to avoid.

Using single column primary keys

CYPEX contains the concept of “identity columns”. If you want to build an online form, CYPEX has to uniquely identify a row to ensure that the right things are updated. Identity columns usually represent some kind of ID. It's important to understand that these keys must be single-column keys. In order to manage complexity and maintain good performance, CYPEX doesn’t handle composite keys. It therefore makes sense to ensure that every table (entity) has a synthetic key. Single-column keys aren't only important if you want to create forms. They

can also be important if you're creating a dashboard: You can't click into a chart if you can't easily identify what it was that you clicked on.

We therefore suggest that you ALWAYS add an ID column (even if it's not strictly necessary), for easier handling.

Handling NULLs wisely

NULLs are a bit tricky. Often, a web GUI can't distinguish between NULL and empty strings. This is especially difficult in the case of checkboxes.

The following types of modelling should therefore be avoided:

```
...
column     boolean          NOT NULL,
...
```

In these cases, if CYPEX does not see any input, it sends NULL to the backend. This is also true for text fields. In order to be consistent across the platform, the same behavior is used for boolean fields.

Circular dependencies

Circular dependencies aren't perfectly suited for web applications. The first question is: What is a circular dependency? Here's an example:

```
test=# CREATE TABLE a (id int UNIQUE);
CREATE TABLE
test=# CREATE TABLE b (id int UNIQUE);
CREATE TABLE
```

In this case we have two tables. If these two tables reference each other, we'll end up with a problem:

```
test=# ALTER TABLE a ADD FOREIGN KEY (id)
             REFERENCES b (id);
ALTER TABLE
test=# ALTER TABLE b ADD FOREIGN KEY (id)
             REFERENCES a (id);
ALTER TABLE
```

The problem here is that you can't insert data into any table without violating the other table's constraint:

```
test=# INSERT INTO a VALUES (1);
ERROR:  insert or update on table "a" violates foreign key constraint
"a_id_fkey"
DETAIL:  Key (id)=(1) isn't present in table "b".
test=# INSERT INTO b VALUES (1);
ERROR:  insert or update on table "b" violates foreign key constraint
"b_id_fkey"
DETAIL:  Key (id)=(1) isn't present in table "a".
```

This problem can be solved by marking a constraint as INITIALLY DEFERRED or marking the entire transaction as such. However, CYPEX changes the content of a query. That means that if you want to use circular dependencies, you'll need to adjust the code behind the scenes on your own. However, it's usually better to avoid circular dependencies entirely, if possible.

Using data types cleverly

Data types are the backbone of every relational model. CYPEX maps data types used by the relational model to GUI elements. Be aware that data types used by extensions are generally mapped to "text" because CYPEX does not support obscure types.

Handling "interval" fields

Intervals are basically treated as text fields by CYPEX. This is important, as CYPEX does not contain full support for this data type yet.

Performance and efficiency

Mind that CYPEX enables you to create applications quickly and efficiently - that does not mean that applications will perform "quickly" by default. If you want to achieve good database performance, work through the following CYPEX performance recommendations:

- Index columns you are searching on
- Index both sides of a join
- Enable pg_stat_statements
- Deploy proper pgwatch2 monitoring
- Materialize large aggregations
- Avoid expensive live queries

CYPEX tries to avoid expensive queries whenever possible. Tables will only fetch a handful of rows - which greatly improves performance. However, if a table is fed by a fairly complicated query, performance might still suffer, particularly if

indexing isn't done properly. For that reason, we highly recommend that you frequently check up on pg_stat_statements.

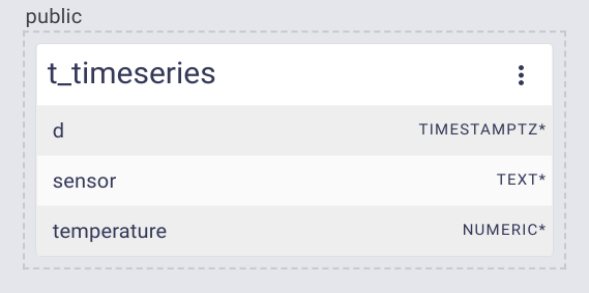
Furthermore, we recommend testing CYPEX applications using representative amounts of data to ensure that performance is close to what you can expect in production.

Using partitioning

Many people use PostgreSQL partitioning. It's important to understand how this feature can be used in CYPEX. The following listing shows how partitions can be created:

```
BEGIN;

CREATE TABLE t_timeseries (
       d             timestamptz   NOT NULL DEFAULT now(),
       sensor        text          NOT NULL,
       temperature   numeric       NOT NULL
) PARTITION BY RANGE (d);

CREATE TABLE t_timeseries_2020
       PARTITION OF t_timeseries
       FOR VALUES FROM ('2020-01-01') TO ('2021-01-01');


CREATE TABLE t_timeseries_2021
       PARTITION OF t_timeseries
       FOR VALUES FROM ('2021-01-01') TO ('2022-01-01');

CREATE TABLE t_timeseries_2022
       PARTITION OF t_timeseries
       FOR VALUES FROM ('2022-01-01') TO ('2023-01-01');

COMMIT;
```

Please note that the model builder is only going to show the parent table in this structure:



Keep in mind that not all versions of PostgreSQL behave the same way when dealing with partitions. This is also true for index creation. You need to understand how your version of PostgreSQL behaves when it comes to partitioning and index creation. Thus we recommend carefully testing the application, and limiting the use of workflows in combination with partitions in general. Future versions of PostgreSQL may exhibit further differences, so it's important to stay up-to-date on the changes in partitioning behavior in the latest version.

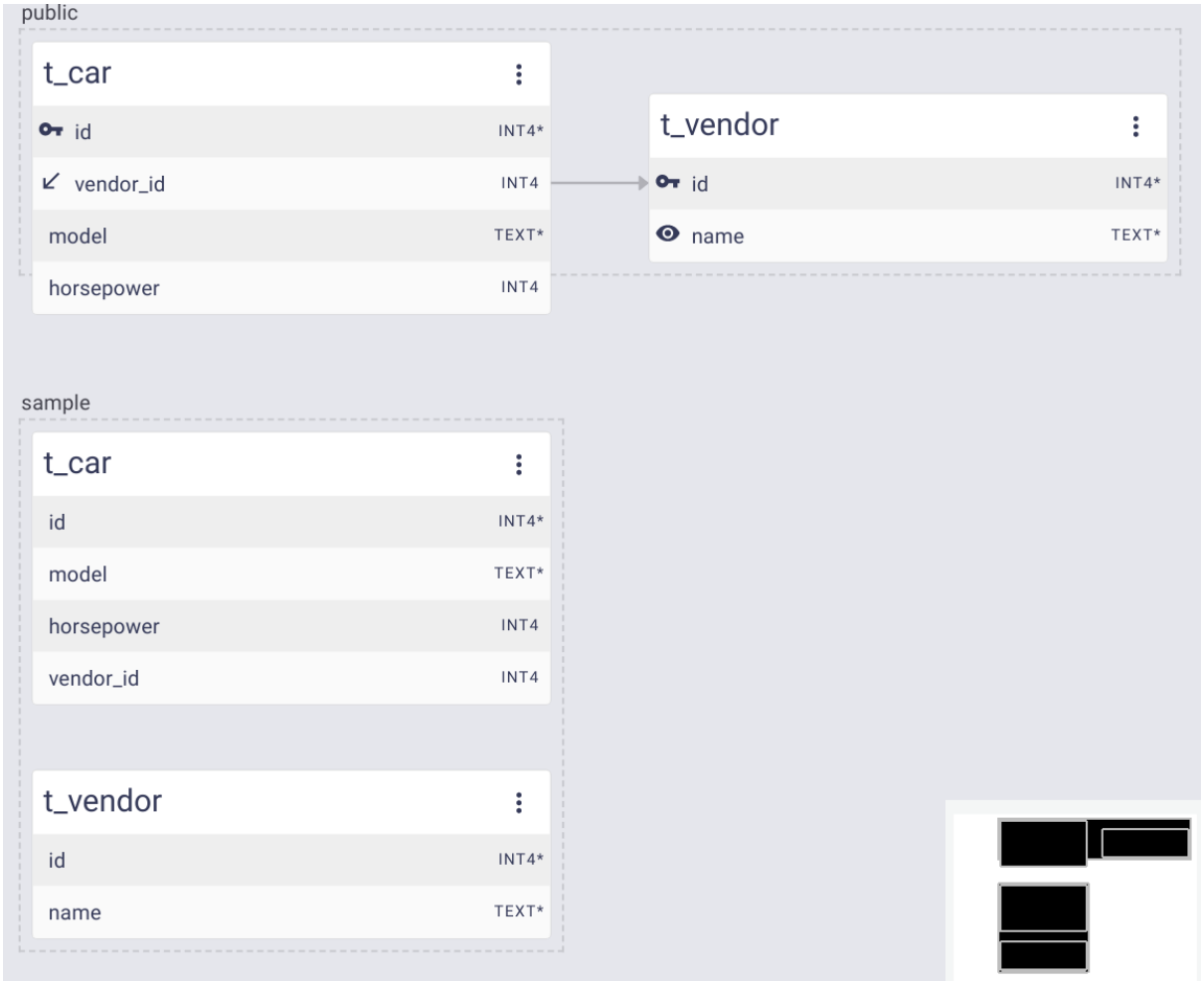
Processing FDWs (Foreign Data Wrappers)

CYPEX supports PostgreSQL-style **FDWs**. However, there are some special cases which have to be taken into account when using FDWs. You should keep several limitations in mind:

- No support for workflows
  - Unable to enforce constraints on the remote side
  - Unable to deploy reliable triggers
  - Can't rely on constant data structures on the remote side
- No support for advanced auditing
  - Unable to reliably track changes on the remote side
- No support for reliable foreign keys

Therefore the only useful situation is to use FDWs as data sources, or as target tables (if this is supported by the FDW in general).

In the model builder, FDWs are shown as normal tables:



In this case, a FDW was created as follows:

```
CREATE EXTENSION postgres_fdw;

CREATE SERVER pgserver
      FOREIGN DATA WRAPPER postgres_fdw
      OPTIONS (host 'localhost', dbname 'cypex');

CREATE USER MAPPING FOR public
      SERVER pgserver
      OPTIONS (user 'postgres');

CREATE SCHEMA sample;

IMPORT FOREIGN SCHEMA public
      FROM SERVER pgserver
      INTO sample;
```

```
SELECT * FROM sample.t_vendor;
```

More features will be added in this area in future versions of CYPEX.

Sample applications

After this brief introduction, it's time to create your first sample applications. We'll show you a set of basic apps which will guide you through the process. Every application will allow you to dive deeper into CYPEX and to learn more about its features.

Your first application

Let's dive headlong into the first CYPEX application. The goal is to create a form, as well as a dashboard showing a report.

Step 1: Creating an SQL model

The first step is to create an SQL model. In our example, we'll create a "sales" schema with couple of tables, initially populated with some of sample data:

Sample data

To demonstrate how CYPEX works, we have compiled a data set. We'll use the following tables and permissions to define queries:

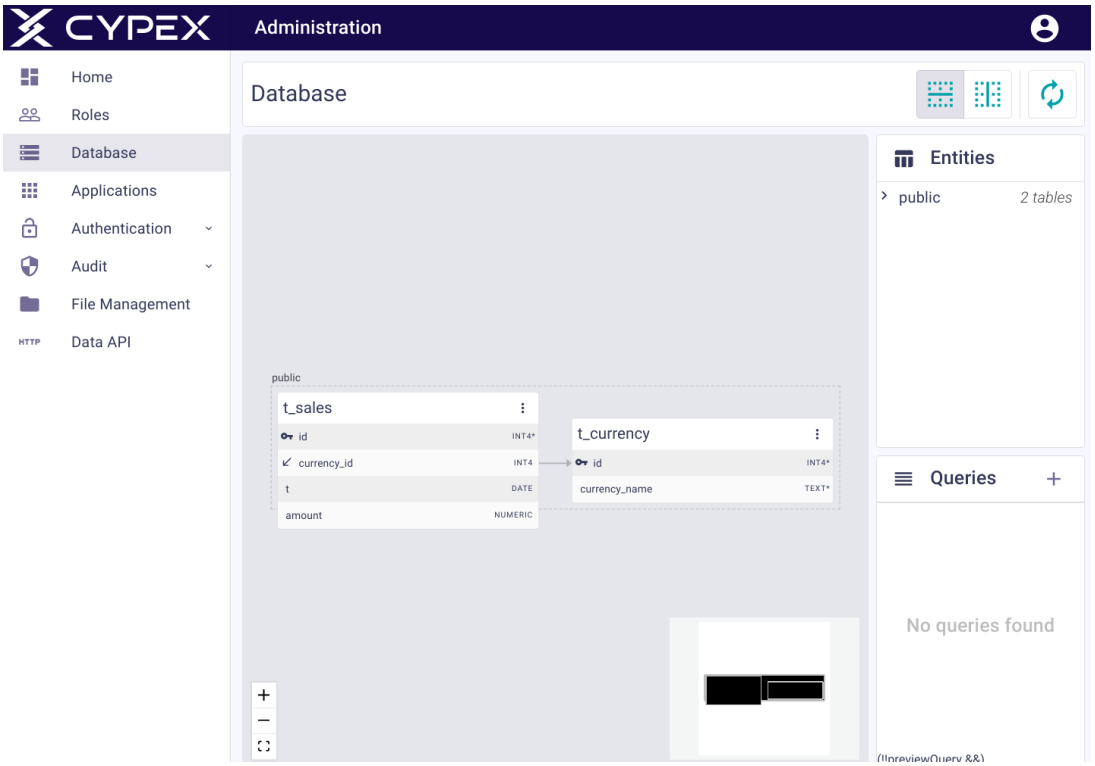
```
cypex=# GRANT USAGE ON SCHEMA public TO cypex_user;
cypex=# GRANT whoever TO authenticator;
cypex=# SET SESSION AUTHORIZATION cypex_admin;

cypex=# CREATE TABLE t_currency (
                id              serial     PRIMARY KEY,
                currency_name   text       NOT NULL
);
CREATE TABLE
cypex=# INSERT INTO t_currency (currency_name)
          VALUES ('USD'), ('EUR'), ('CHF'), ('GBP');
INSERT 0 4
cypex=# SELECT * FROM t_currency;
 id | currency_name
----+---------------
  1 | USD
  2 | EUR
  3 | CHF
```

```
  4 | GBP
(4 rows)
```

The second table shows sales amounts::

```
cypex=# CREATE TABLE t_sales (
          id              serial     PRIMARY KEY,
          currency_id     int        REFERENCES t_currency (id),
          t               date,
          amount          numeric(10, 2)
);
CREATE TABLE

cypex=# INSERT INTO t_sales (currency_id, t, amount)
          VALUES     (1, '2022-01-04', 3243.45),
                     (1, '2022-01-05', 4324.43),
                     (2, '2022-01-09', 1242.98),
                     (2, '2022-01-10', 985.34),
                     (2, '2022-01-11', 684.32);
INSERT 0 5
cypex=# SELECT * FROM t_sales;
 id | currency_id |     t      | amount
----+-------------+------------+---------
  1 |           1 | 2022-01-04 | 3243.45
  2 |           1 | 2022-01-05 | 4324.43
  3 |           2 | 2022-01-09 | 1242.98
  4 |           2 | 2022-01-10 |  985.34
  5 |           2 | 2022-01-11 |  684.32
(5 rows)
```

This is a 1:n relationship. The model builder will display the new model:

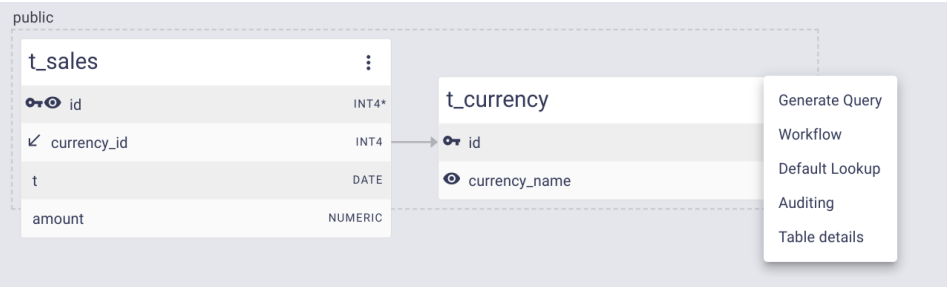


You see above that currency_id references t_currency.id.

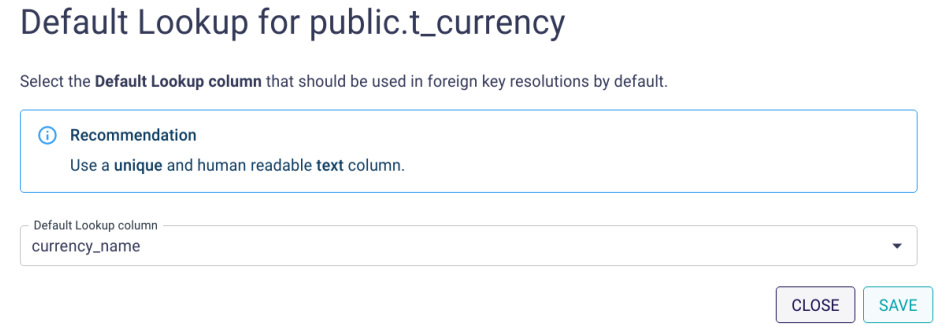
NOTE: It's important to grant permissions to "authenticator", otherwise the login process won't work as desired.

Step 2: Defining default lookups

CYPEX supports "default lookups". What does that mean? In a relational model, foreign keys are quite common. The problem is: If you want to display a table containing a foreign key, you might not be interested in seeing every single type of ID displayed. Let's take a look at our "t_sales" table as shown in the model builder:

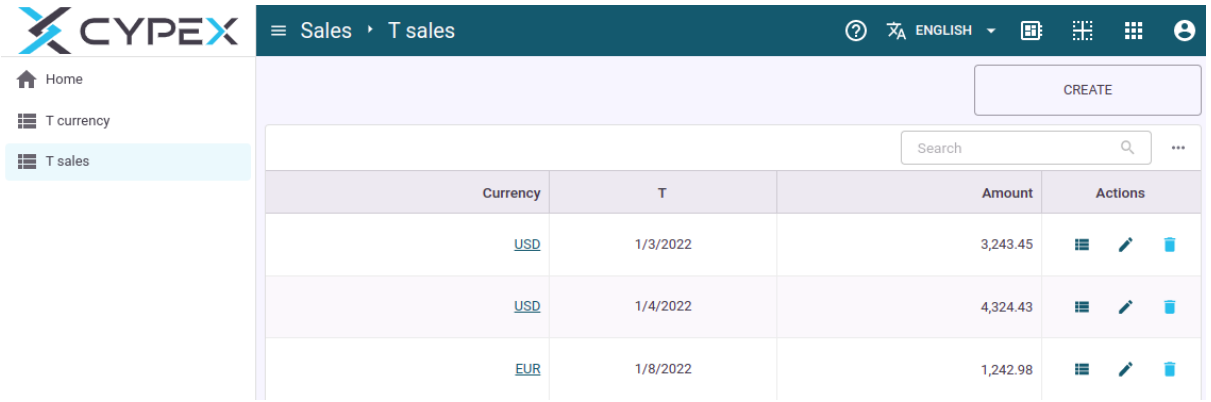


What we really want to display in our form is the ID, the name of the currency, a timestamp and the amount. To make sure that the GUI easily resolves the key in the manner desired, click on the currency table and select “default lookup”. There you can define how to resolve an ID pointing to this table. CYPEX will reverse-engineer the model, and will always display the name instead of the plain ID.
Here’s what the lookup form looks like:



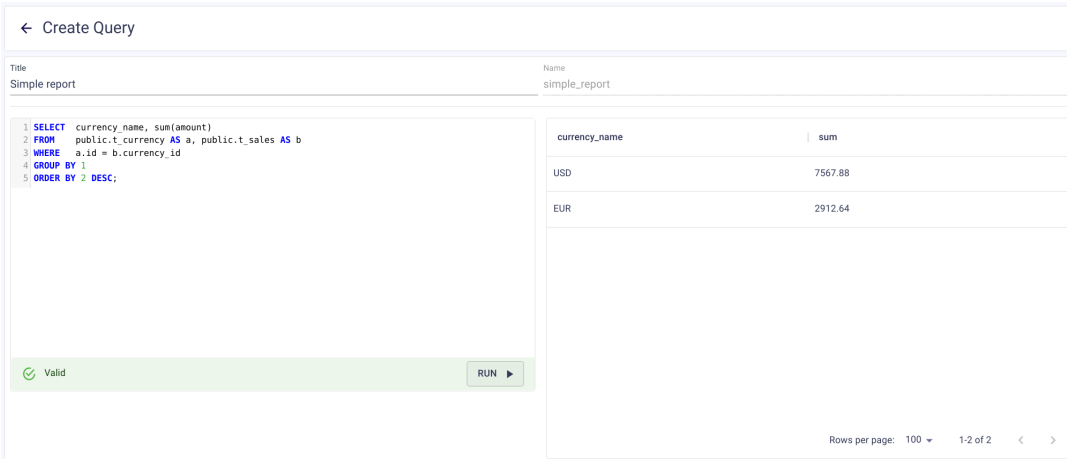
Once you’ve saved this info, the model knows how to resolve ID’s in an elegant way. Note that this default resolution happens at the model level.

In the end, the application generated will display the content of the default lookup table:

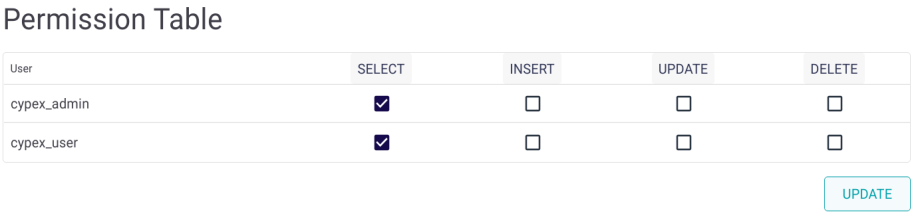


Step 3: Defining a query

After defining default lookups to make life easier, let's move forward and define a query. To create a query click on the "+" icon in the queries menu on the right side of the page:



What you'll find is an SQL editor as well as options to test your queries. The second half of this form is all about permissions:

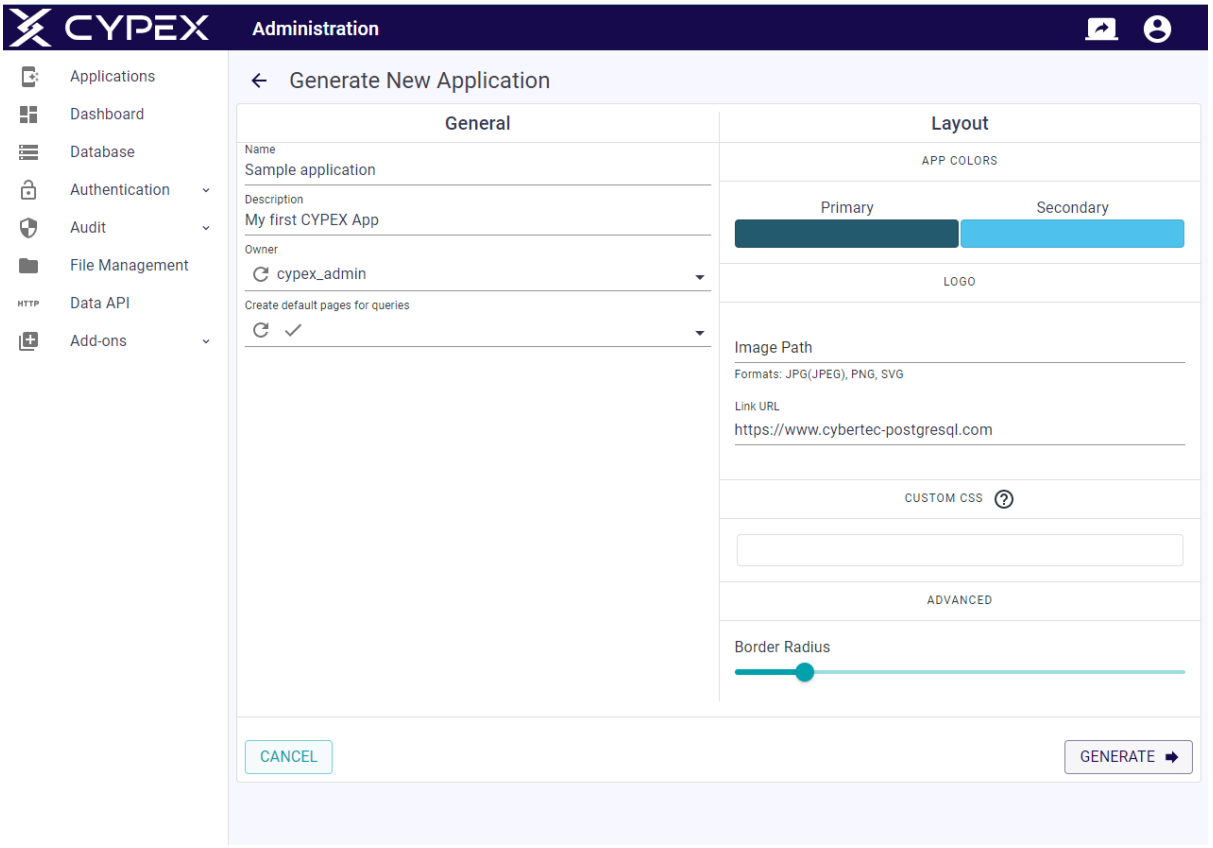


You can visually define who is allowed to perform which operation(s) on this query. Note that in some cases, a trigger will be needed to handle insertions. Therefore, read-only queries such as reports should only have SELECT permissions.

Step 4: Predicting an application

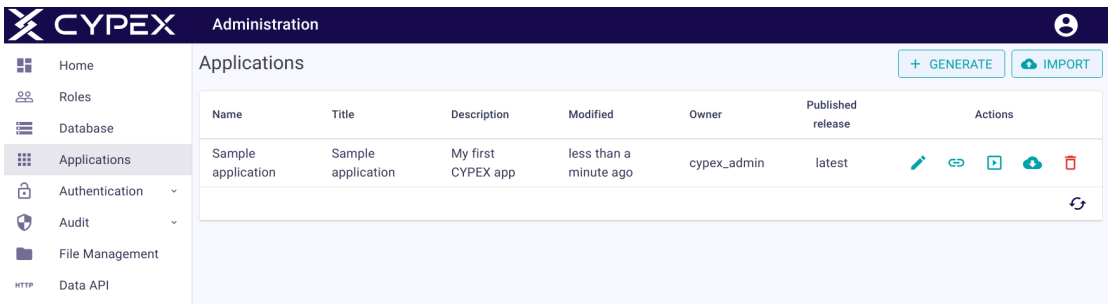
Now that you've defined your first data model, you can create your first application. Go to the model builder and choose "Applications" in the menu on the left. Then press the "Generate" button.

CYPEX will open the app generation form:



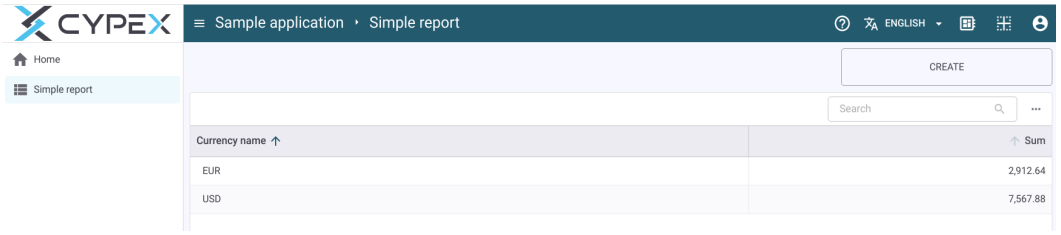
Your application needs a name - that name is going to be the title of the entire application. What's also important is the owner: you'll render the app for this GUI, so **make sure the owner of the GUI has all the permissions needed to handle the underlying data**.
Finally, decide which queries default pages will be generated for, when rendering the application. What is the logic here? Suppose you have 10 queries. You might produce 3 applications (each of them using 5 queries). Keep in mind: you can build as many CYPEX apps as you want on top of those queries.

After hitting the "generate" button, you have your first CYPEX app:



Trying it all out

Click on the button in the middle and execute the app. You'll see one menu entry, and one table which has been generated for us by CYPEX:

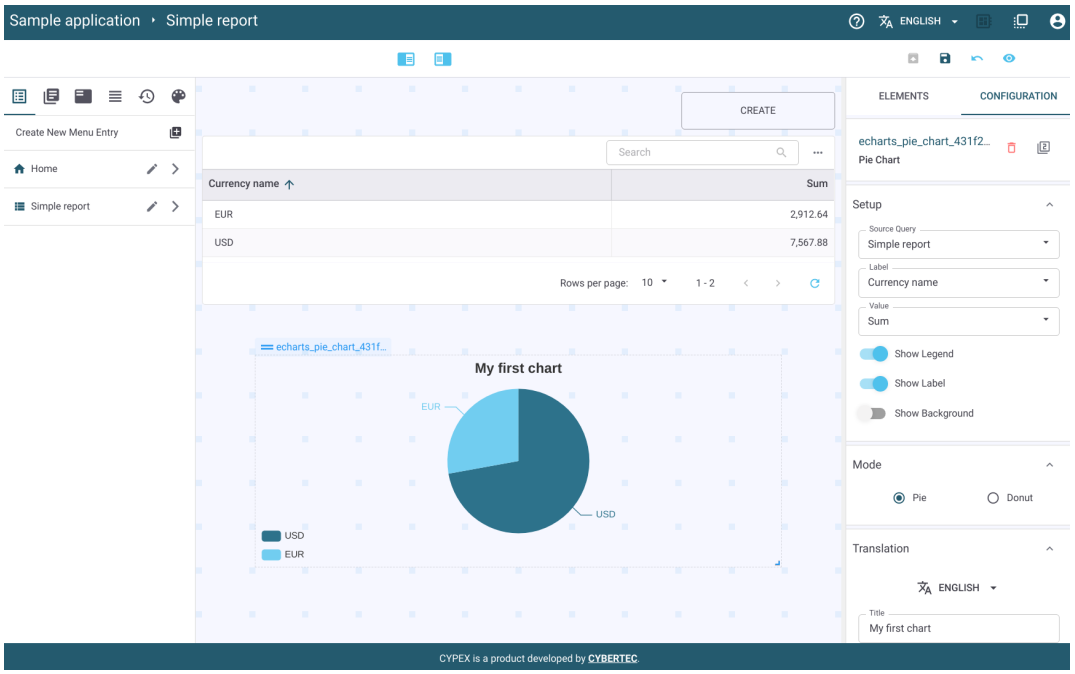


Welcome to CYPEX.
You have just built your first application.
Congratulations!

Building a dashboard

So far, the default rendering process has created a menu entry and a table. However, what we really want is a dashboard. The goal is to modify the application and add some charts.

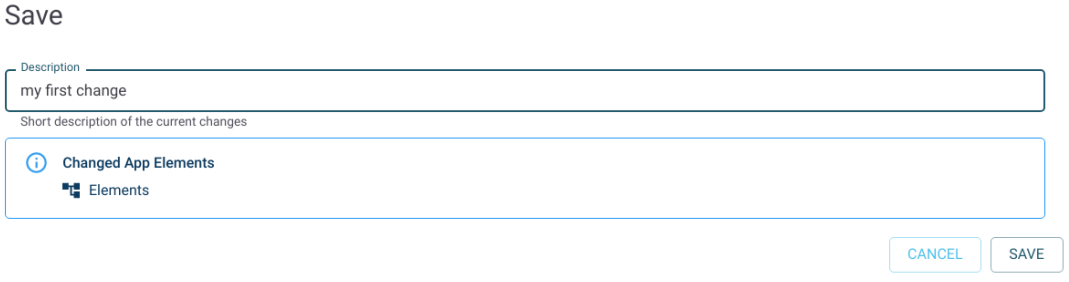
Click the “edit application” button in the application. The application will then be in edit-mode which allows you to change all graphical elements:



In this case, we’ve used drag & drop to add a pie chart to the app. The important part is the configuration of the data sources: **Select the “query” in “source query” to tell CYPEX which data source to use.**

Then select the axis needed by the pie chart. You can decide which titles to use, what type of chart you want, and a whole lot more. The basic idea is the same for all the types of GUI elements which can be added.

Once the changes are done, save them:



The “save” function will tell you what has been changed and release a new version of the GUI.

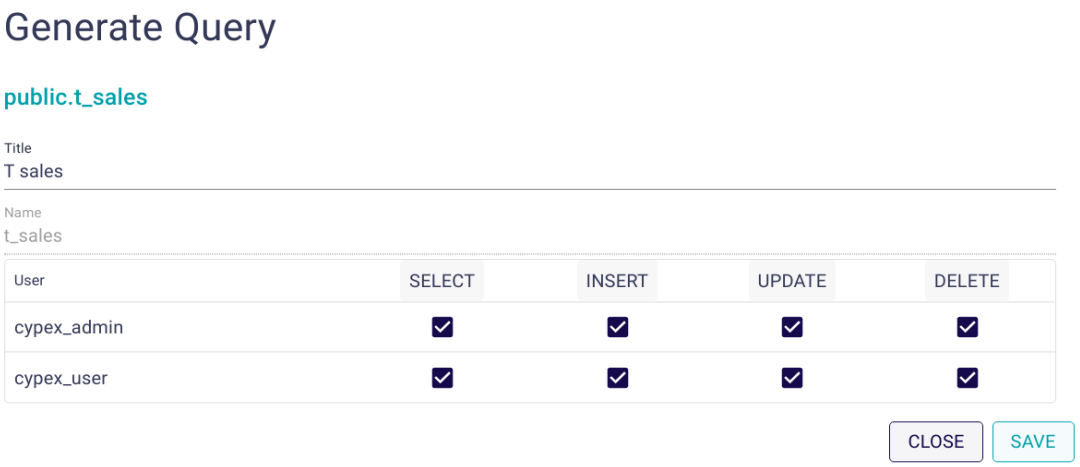
You can add as many widgets as you want. Pie charts, line charts, bar charts - CYPEX has them all. The underlying concept explaining how to configure things is the same for almost all charts. Only maps require a different infrastructure - (geo JSONs) but more on that later.

Creating forms

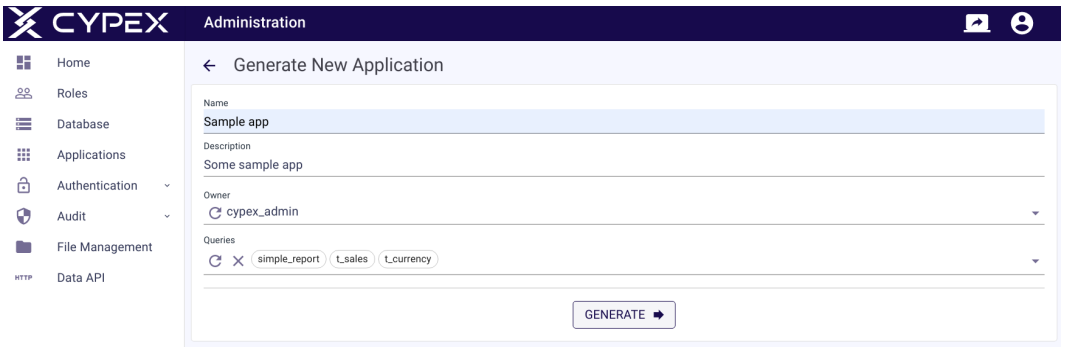
It's easy to create a form for an entity. CYPEX allows you to create a "default query" for this purpose.

The main question is: What is a default query? In CYPEX you don't usually change tables directly, rather you create a view behind the scenes. What happens here is that the default query will be a "SELECT * FROM tab". The advantage is that you can nicely abstract user permissions that way and separate the underlying data from the access layer.

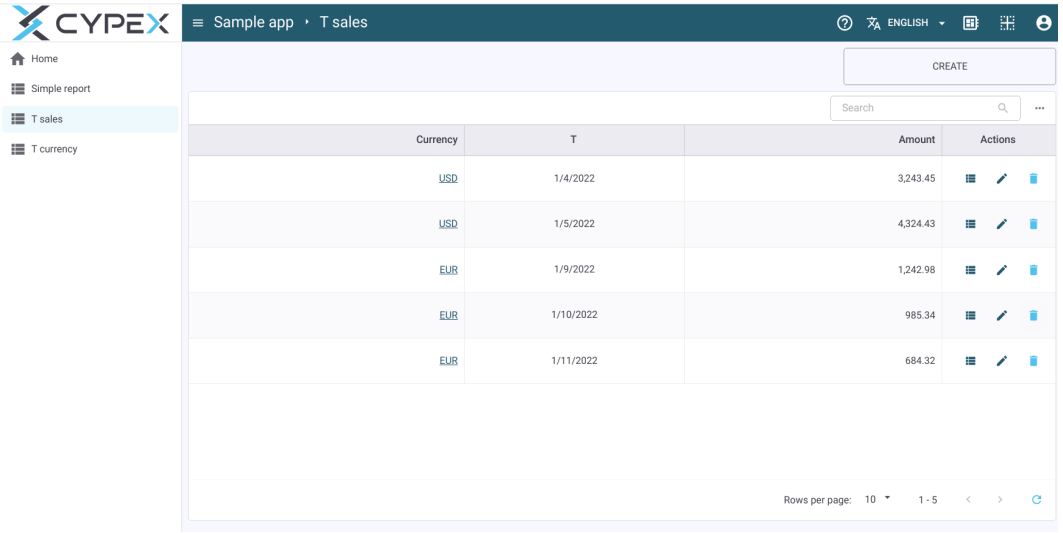
In order to create a default query, click on a relation and select the first entry ("Generate Query").



All you have to configure in this case are the title and the permissions. **Note that in order to create a new form, the user needs INSERT or UPDATE permissions.** Otherwise, those forms will not be generated by default.
In our example, we've created default queries for both tables:



As you can see, the new queries aren't accessible when rendering the new application. Incremental rendering is also possible. Alternatively, you can add new queries and build their forms manually. However, it's generally easier to use the rendering infrastructure which automatically creates all necessary forms.
After generating the application, you see additional menu entries:



It's important to notice that the permissions set before ensured that the table including the edit buttons was generated. Modify a row:

What's important to note here is that the default currency is displayed in the drop-down menu. The reason for that is that we've defined a default resolution for this column. Therefore CYPEX already knows how to handle this field.

Making forms more sophisticated

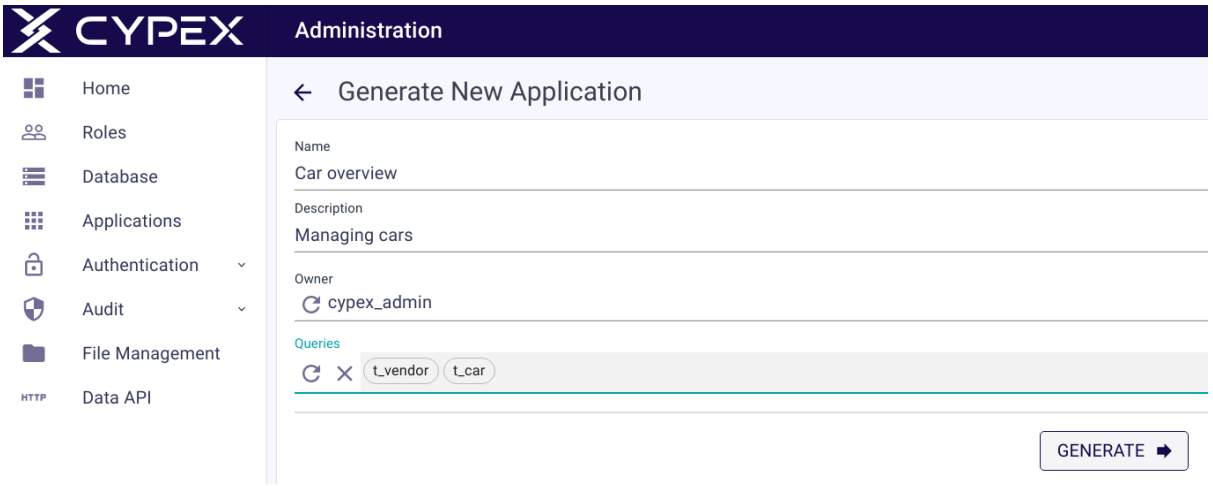
So far, you've seen how to generate simple forms. Each input field is represented as a text field. However, this might not be desirable at all. Let's take a look at the following sample data structure:

```
BEGIN;
```

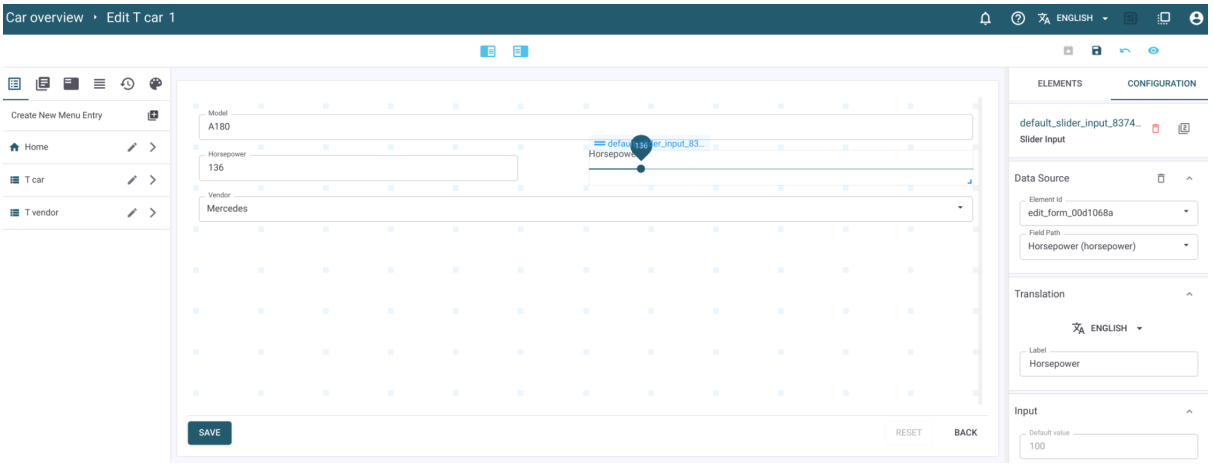
```
CREATE TABLE t_vendor (
       id     serial        PRIMARY KEY,
       name   text          NOT NULL
);

INSERT INTO t_vendor (name)
       VALUES ('Mercedes'), ('Opel'), ('Tesla');


CREATE TABLE t_car (
       id            serial        PRIMARY KEY,
       model         text          NOT NULL,
       horsepower    int           CHECK (horsepower > 0),
       vendor_id     int           REFERENCES t_vendor (id)
                                          ON UPDATE CASCADE
                                          ON DELETE CASCADE
);

INSERT INTO t_car (model, horsepower, vendor_id)
       VALUES ('A180', 136, 1), ('A200', 163, 1),
              ('Mokka', 96, 2), ('Insignia', 174, 2);

GRANT ALL ON t_vendor, t_car TO authenticator;

COMMIT;
```

What we have here are vendors and cars. The parts to focus on are: a.) the "horsepower" field as well as b.) the foreign key. You've already learned that in order to build smarter forms, you can use default resolutions. In order to achieve that, go to the model builder, then click in "Default lookup" on the vendor table. Select the name column. Then, generate the default queries for both entities. Now CYPEX knows that it has to ask for text input rather than ID's.

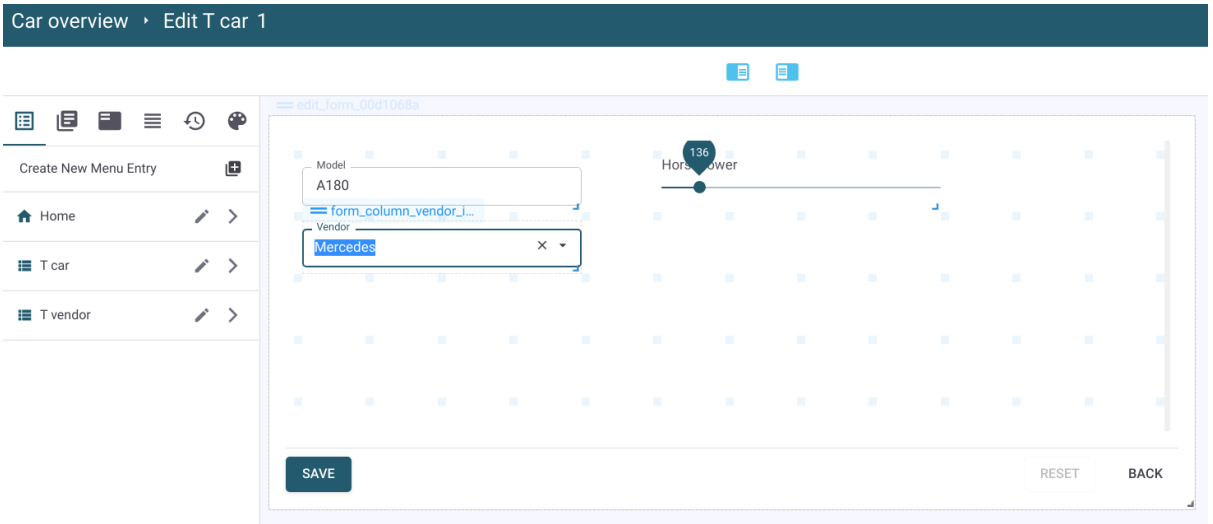
Now let's render the application:



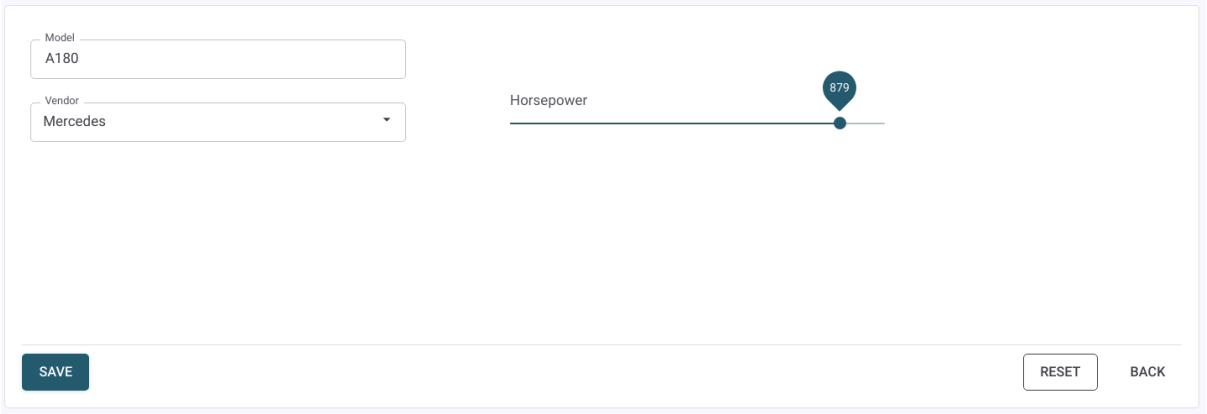
Select both queries you want to render and click the “Generate” button. Start the application, select any car you want to modify, and start the edit mode. What you'll see is a text field for the model (which is fine), a text field for the horsepower value and a drop-down created by the default resolution. Now, replace the text field for the horsepower entry with a slider. The way to do this is to select a slider from those elements, drag them in and voilà, you’re ready to configure the element. You need to assign the same data sources to the slider. Select the same field as in the old horsepower field and configure the remaining variables you want to see:



In my example, the slider will range from 0 to 1000. Then you can delete the old element and arrange these elements exactly the way you want them to be arranged:



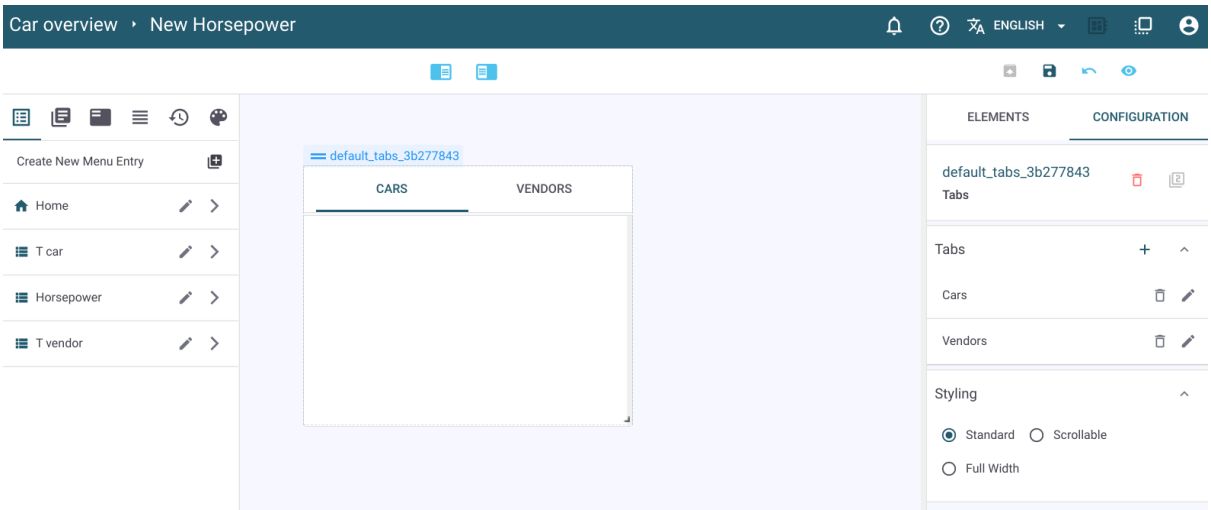
Save the application, and the form is ready to use:



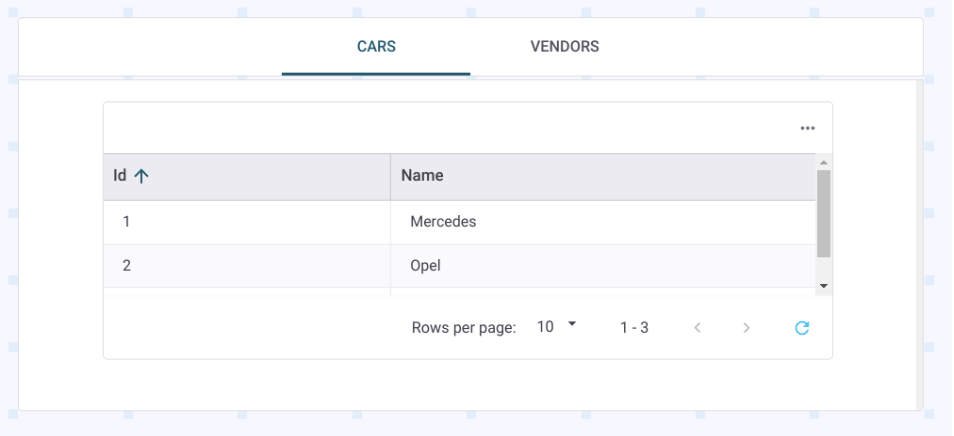
You have successfully replaced a simple text input with a more advanced element.

Working with tabs

Sometimes you want input forms or tables to be in tabs. CYPEX provides this feature and allows you to easily add tabs . Drag and drop a “Tabs” element into your WYSIWYG editor.
By default, your element will be empty so you have to add tabs to it:



To do that, check out the configuration menu and add tabs. Once the tabs are created, you can fill them with elements of your choosing:



In this case, a table has been added to the GUI.

Incremental changes

Data models can change over time, which means default rendering can become a problem. Imagine that you have an existing application and you want to extend it with additional entities, queries, and so on.

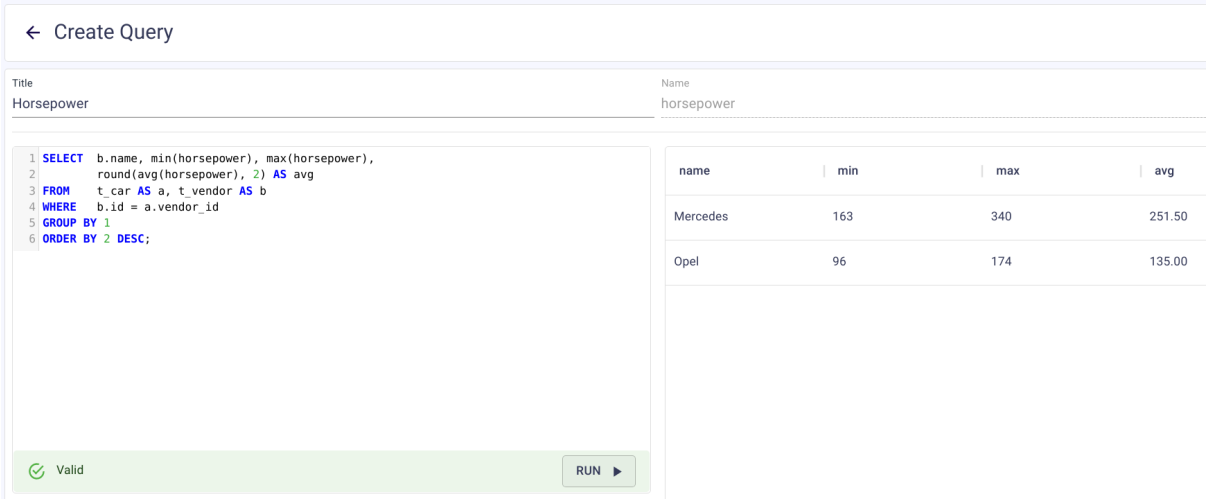
CYPEX supports changing data structures. Let's outline two relevant cases:

- Incremental rendering
- Changing query definitions

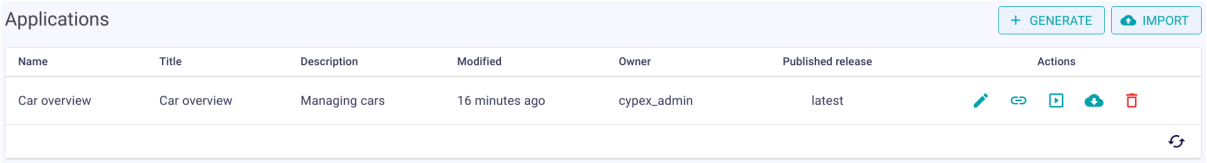
Incremental rendering

Building forms by hand after default rendering is done can be quite cumbersome and slow.

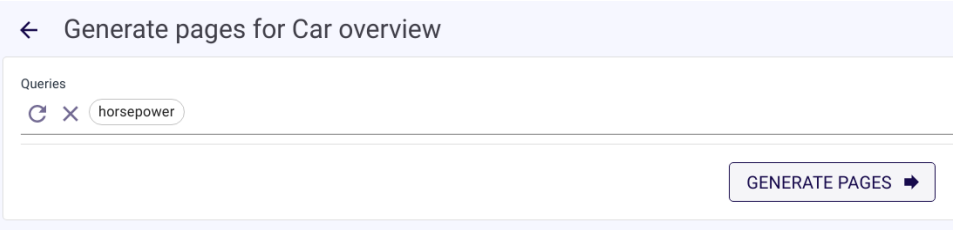
CYPEX supports incremental rendering. Once an application is done, you can easily create new queries:



Once you have created the query, you can jump to the applications overview in the model builder. Select the second icon ("Generate and add pages"):



Then tell CYPEX which queries you want to add to the application:



In this case the “horsepower” query will be added as a new menu entry and thus a new table:

Note that the data source will be available for other elements as well.

Changing query definitions

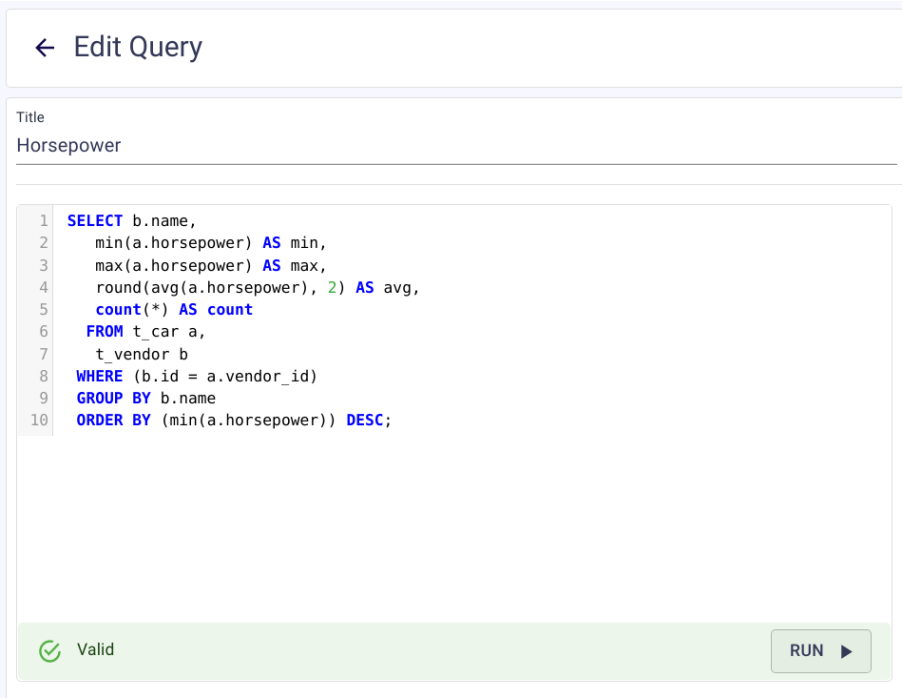
There are other changes which have to be addressed. It often happens that a query definition has to be changed. Such changes are partially supported.

Let’s take a more detailed look.

Adding columns to queries

Adding a column to a query is always possible. Go to the model builder and modify the query accordingly. Note that the query editor uses the “real” PostgreSQL parser to check if the syntax is correct. You can therefore rely on the fact that the query is OK, as long as you can actually save it.

Here’s a possible modification which works for the previous example:



Make sure that you do not change column names, because doing so may break your frontend application. Instead, add columns.

Changing columns of a query

If you want to change a query, life is a bit more complicated. Again, it's **not recommended to change column names**.
However, what is possible is changing the definition of a field without changing the data type. What does that mean? Consider:

```
(count(*) + 1) AS count
```

It's perfectly feasible to change “count(*)” to “count(*) + 1”. It does not change the data type, nor does it change the column definition. However, the following change will result in an error:

```
count(*)::numeric(100, 10) AS count
```

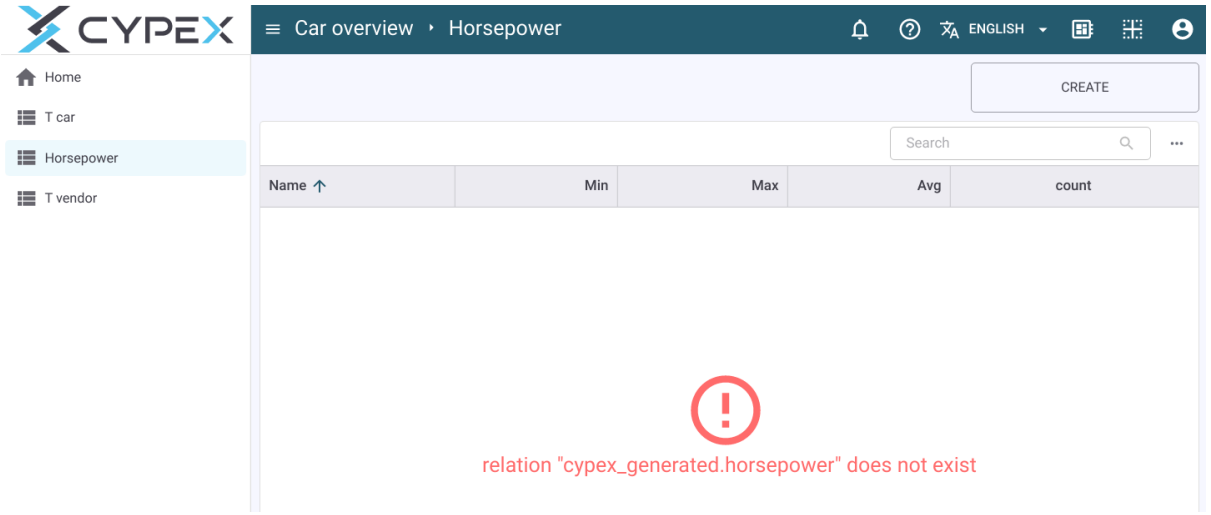
Note that the query is correct from an SQL point of view, but the data type will change, which isn't allowed:

```
cannot change data type of view column "count"
        from bigint to numeric(100,10)
```

Instead of changing view definitions, it might make sense to create a new view, providing the data you need. It has the advantage of not breaking your existing application.

Dropping queries

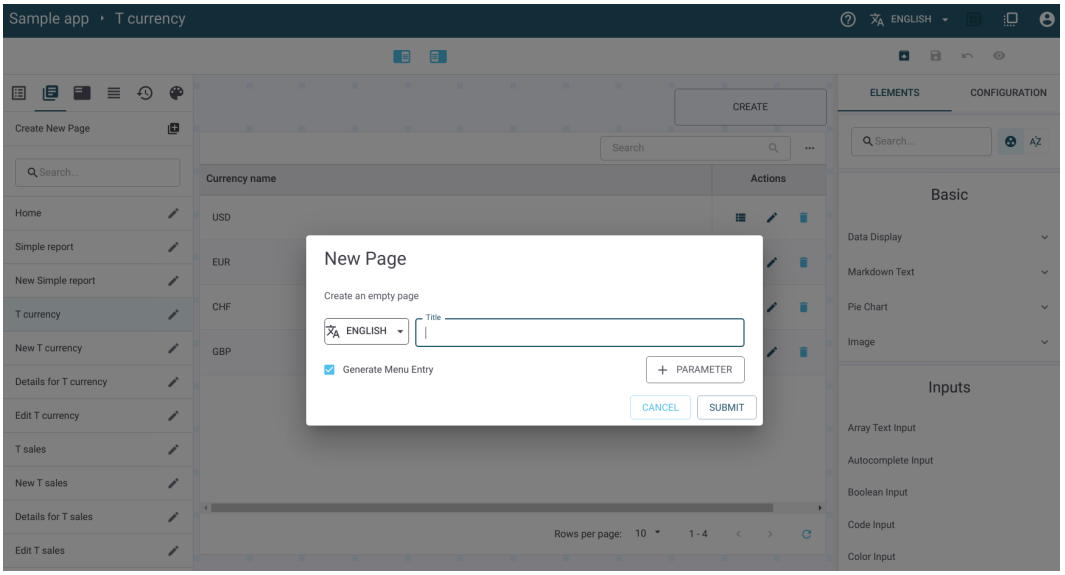
Dropping a query is easy and can be done in the model builder directly. However, it will have implications and it might indeed break your application. The following screenshot shows what happens when a query is dropped:



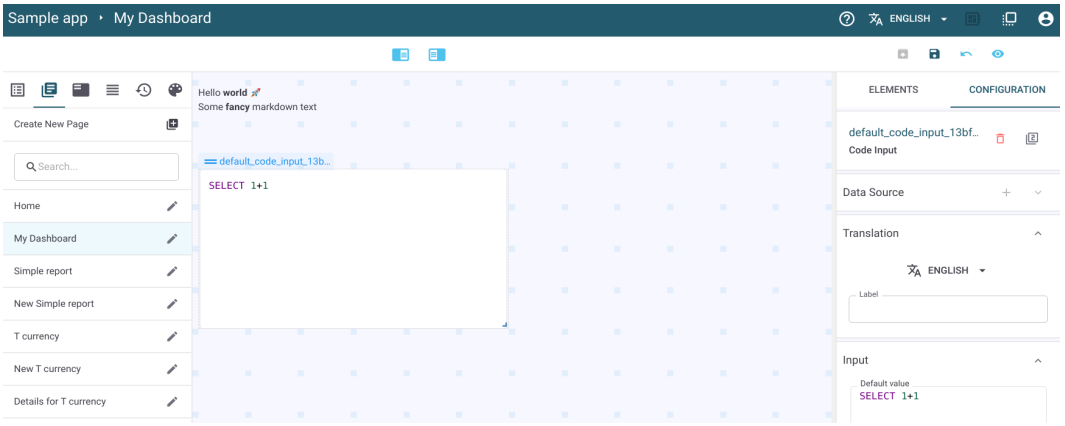
Changing the application becomes necessary in this case, as the underlying data source is lost.

Adding pages to an application

Sometimes you might want to add a new page to an application. To do that, go to edit mode and add a new page:



CYPEX will produce an empty page, which you can then use to add elements later on:



In this case, you can see two elements have been added: a markdown field, as well as a code window.

Creating workflows

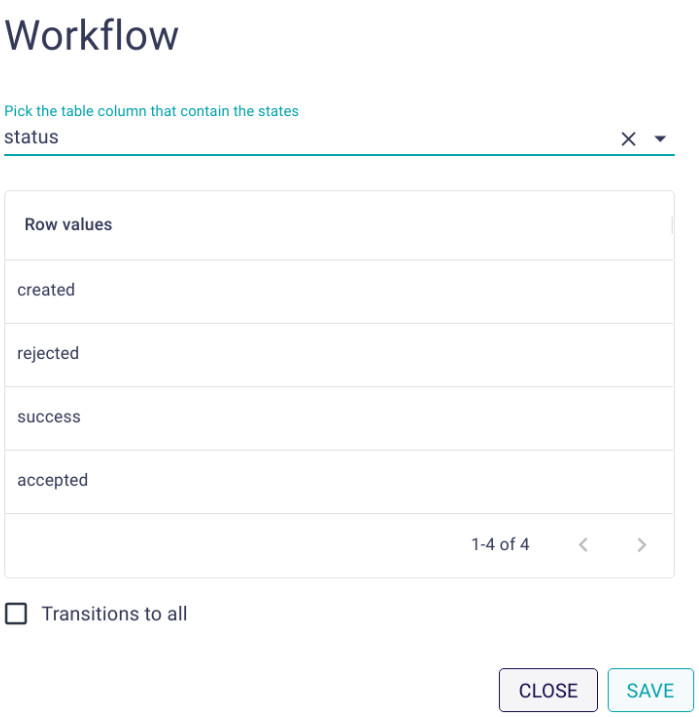
After successfully creating this first application, it's time to move forward and dive into workflows. The goal of the next application is to create a TODO list which can be modified by end users.

Here's some sample data:

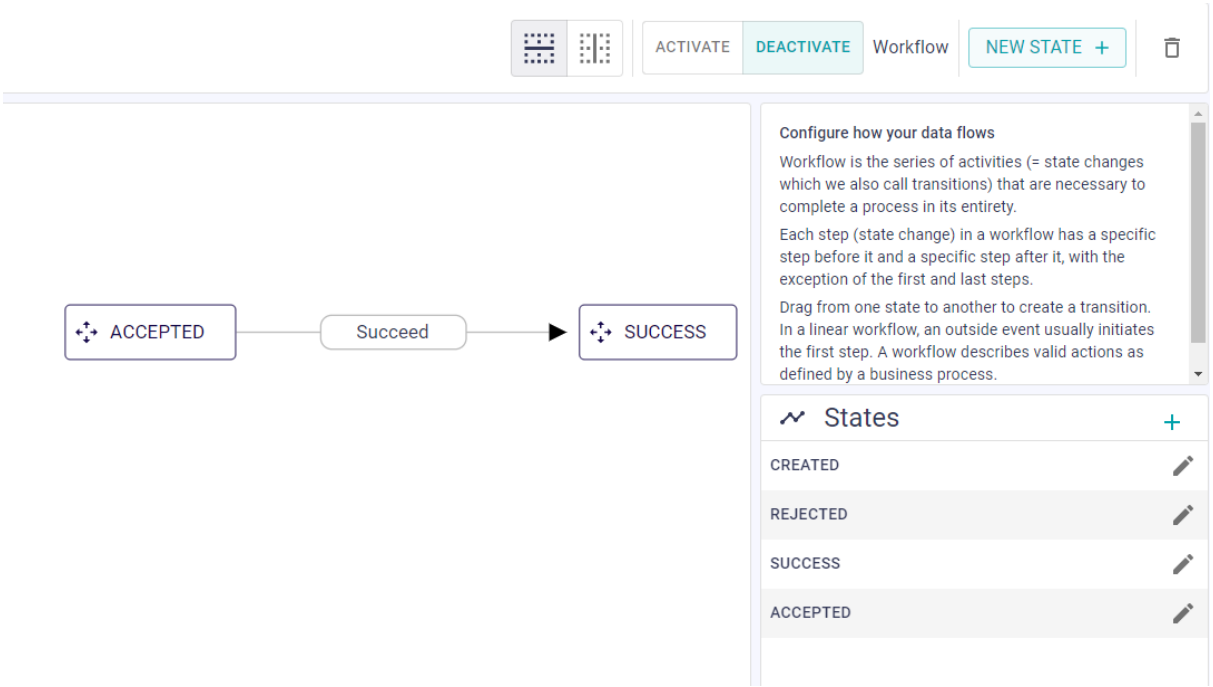
```
BEGIN;

CREATE ROLE todo_owner LOGIN;
GRANT todo_owner TO authenticator;

CREATE SCHEMA todo AUTHORIZATION todo_owner;

CREATE TABLE todo.t_todo
(
    id                serial     PRIMARY KEY,
    tstamp            date       DEFAULT now(),
    todo_item         text       NOT NULL,
    status            text
);

INSERT INTO todo.t_todo (tstamp, todo_item, status)
VALUES
 ('2021-03-04',' Do the laundry', 'created'),
 ('2021-03-06',' Cut the grass', 'accepted'),
 ('2021-03-09',' Eat a steak', 'success'),
 ('2021-03-12',' Slaughter a chicken', 'rejected');

COMMIT;
```

For the sake of simplicity, the TODO list consists of just one table. What is noteworthy here is the last column: The status informs us about the state of an object. A task might have succeeded, failed or it might have been rejected.

You can both enable workflows and configure them in the model builder:

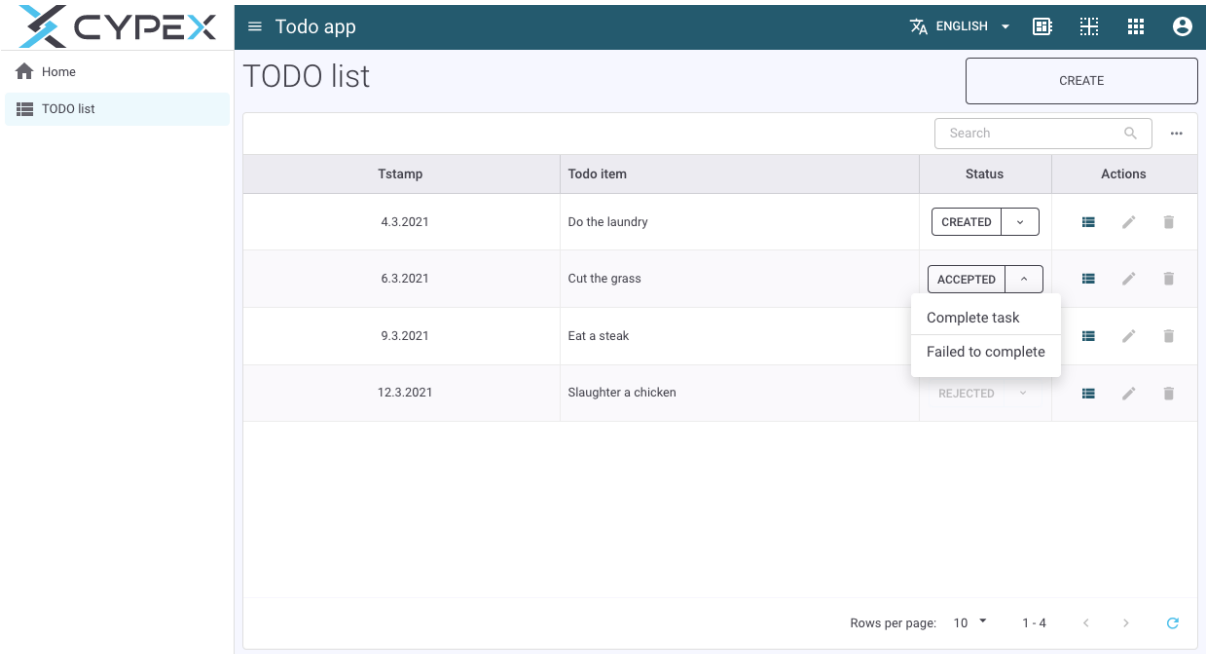


The workflow can easily be drawn using drag-and-drop functionality.



The end result will reflect the changes and allow only the changes defined in the workflow.

Finally, create the query permissions and generate the application:

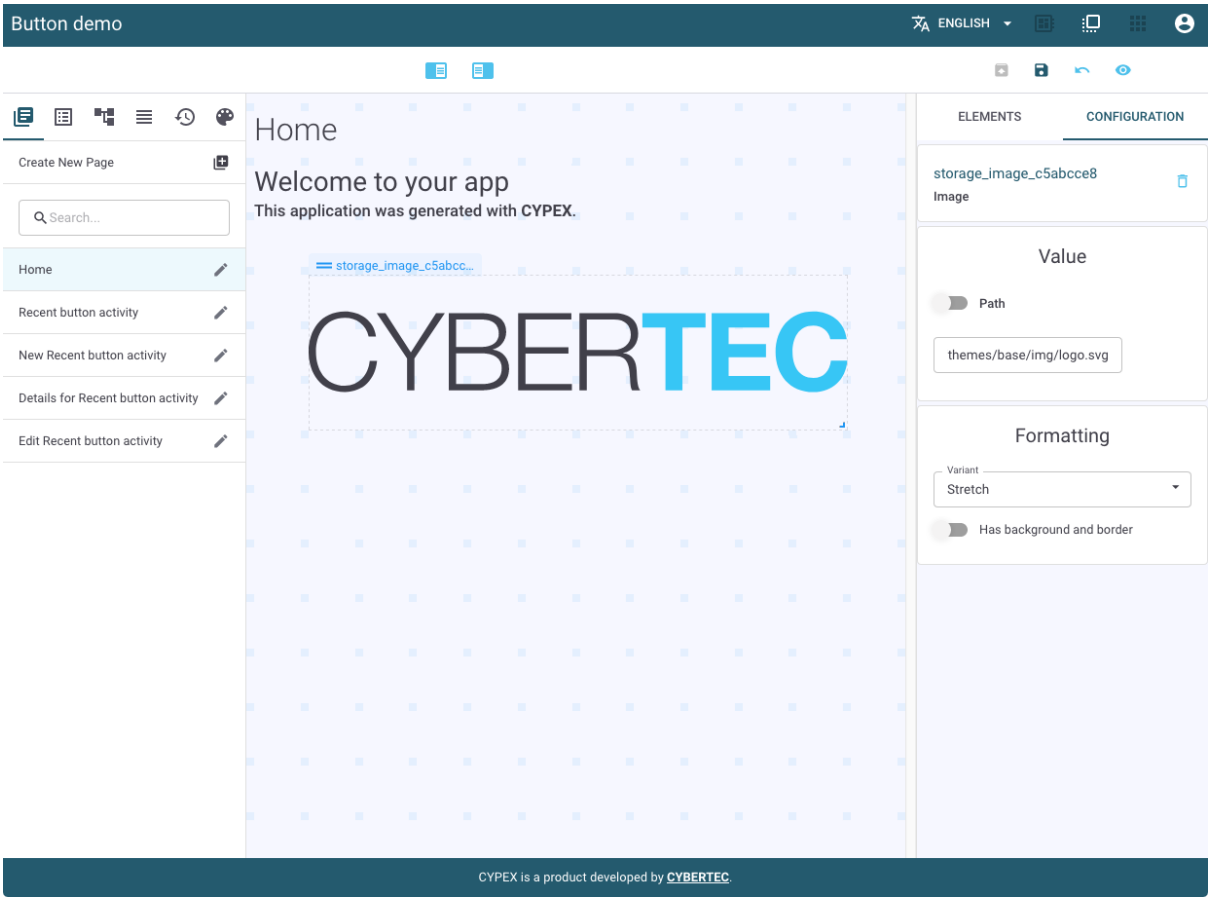


The application is rendered normally. The magic is in the state or status column: CYPEX has generated a dropdown which allows us to make changes.

Note that you can't just select any value from the drop-down. If a row is in the "accepted" state, you can only either fail, or complete the task. Once you are in a "completed" or "rejected" state, the workflow is over - you can't change the data anymore.

Image and file handling

CYPEX supports the integration of external images. Add an image element to your desired page and click on the element for configuration: You may add the desired link to the image, resize the element and define whether the picture can be resized or stretched.



Handling GIS data

CYPEX supports GIS (Geographical Information Systems) data. However, in order to use GIS data in CYPEX, there are some things which have to be taken into consideration.

Let's take a look at a sample table:

```
cypex=# CREATE EXTENSION postgis;
CREATE EXTENSION
cypex=# CREATE TABLE t_area (
            id          serial       PRIMARY KEY,
            name        text,
            g           geometry
);
CREATE TABLE
```

The keys to GIS data are the “geometry” and “geography” columns. These aren't directly visible in a web frontend. Let's take a look at how default queries are generated:

When we generate a default query, the end product will still contain a geometry column:

```
cypex=# \d+ cypex_generated.t_area
                        View "cypex_generated.t_area"
 Column |   Type   | Collation | Nullable | Default | Storage  | Description
--------+----------+-----------+----------+---------+----------+-------------
 id     | integer  |           |          |         | plain    |
 name   | text     |           |          |         | extended |
 g      | geometry |           |          |         | main     |
View definition:
 SELECT f0.id,
    f0.name,
    f0.g
   FROM t_area f0;
```

As it stands, this one isn't readable. To fix this issue, you have to take care of GeoJSON creation on your own. The reason is that the developer has to define what the GeoJSON is supposed to contain. Check out the ST_AsGeoJSON function to transform your column to the desired format.

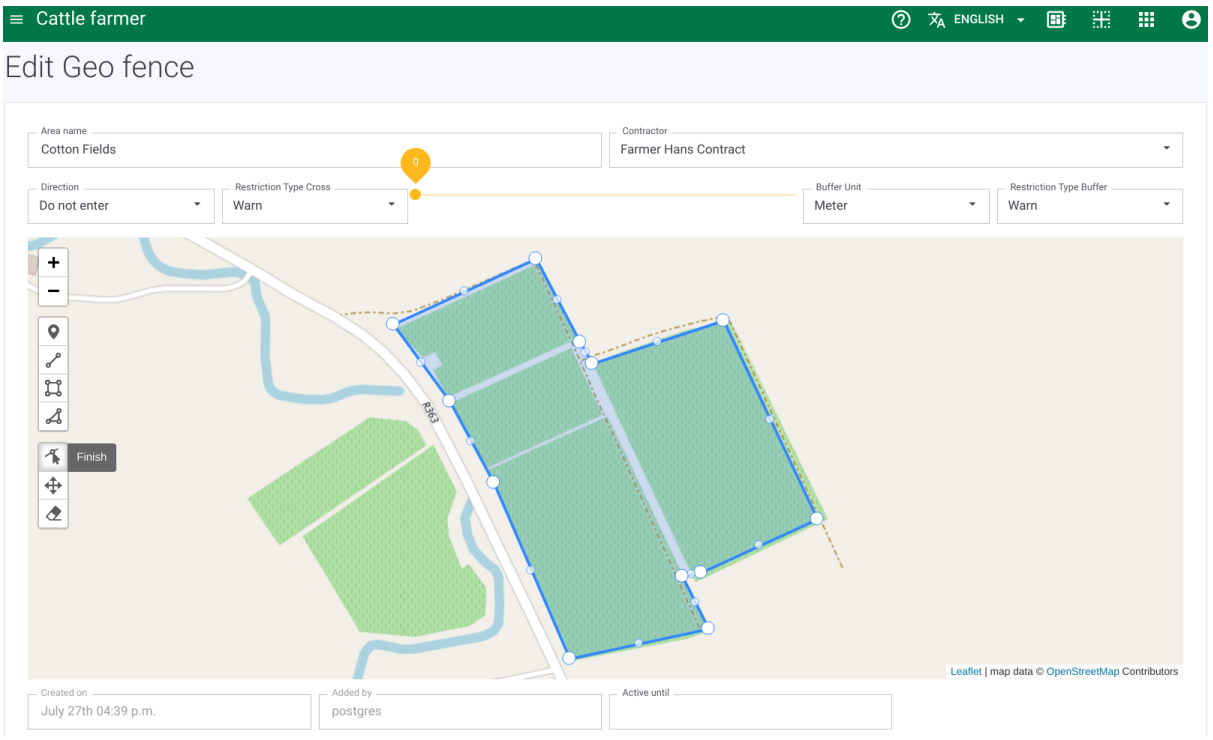
The following example shows how a GeoJSON can be created using a custom query (instead of a default one):

```
SELECT id,
       name
       (st_asgeojson(t_area.*, 'g'::text))::jsonb AS json_position,
FROM   t_area;
```

You also have to create a trigger, in case you want to modify the GeoJSON coming in. You need to define how to transform things back to “geography” or back to “geometry”.

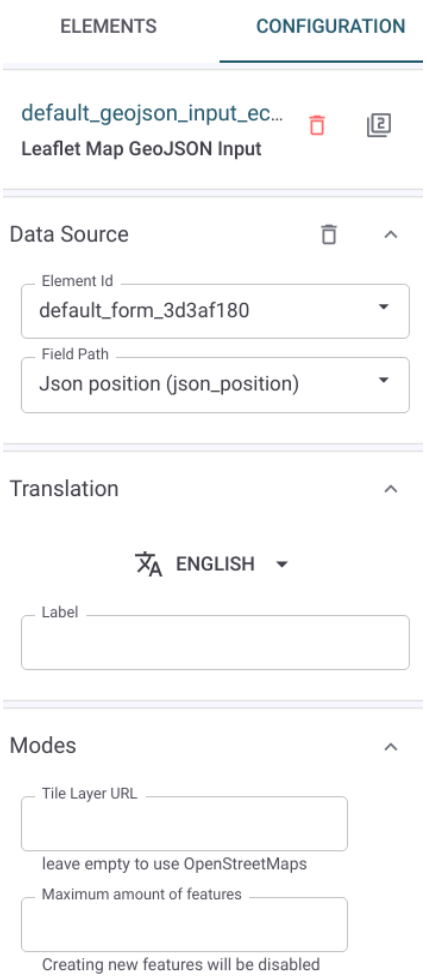
GIS apps in action

With CYPEX you can build powerful GIS apps. The following screenshot shows an example of what’s possible. What you see below is a visual editor which allows you to modify polygons.



It's important to understand how this image was created: let's take a look at what was done in the WYSIWYG editor. A Leaflet Map element was used and the JSON column was selected as the data source. If all triggers are correctly in place, you'll see a map similar to the one above.

The configuration of such a widget is similar to any other widget known to CYPEX. The important part is to use the GeoJSON column to feed the widget with GIS data. In addition to that, you can use background layers to display additional information:



In general, working with GIS data is easy. The CYPEX development team will expand this capability in the future, and add more features to the GIS backend.

Calling server side code

A workflow is a good start if you want to build an application. However, sometimes it's still necessary to add control elements. In this section, you'll learn to add buttons and to write server side code to make your application even more powerful.

Let's start with a basic data model:

```
BEGIN;

CREATE SCHEMA calculator;

CREATE TABLE calculator.t_date
(
    id         serial         PRIMARY KEY,
    t          timestamptz     DEFAULT now()
);

INSERT INTO calculator.t_date (t) VALUES (now());

CREATE OR REPLACE FUNCTION cypex_generated.add_entry()
RETURNS void AS
$$
    INSERT INTO calculator.t_date (t) VALUES (now());
$$ LANGUAGE 'sql';

COMMIT;
```

After creating the module, an entity, the query and permissions, you can generate the application. The result once this is done is a basic application showing nothing more than a table containing a timestamp.

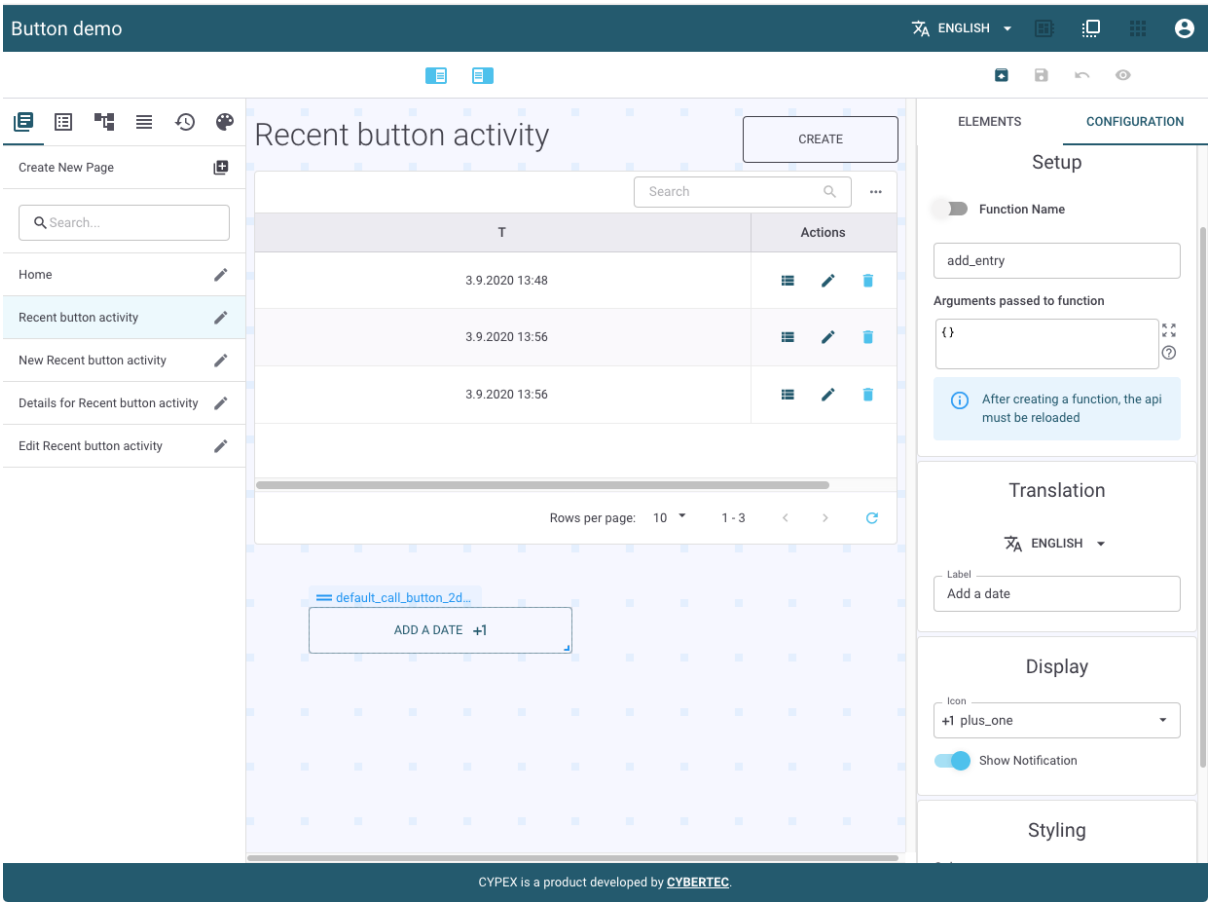
Now the goal is to add a button calling the add_entry() function on the SQL side.

Before you get started, various factors have to be taken into account: The function called by the button must exist in the cypex_generated schema - no other schema will be taken into consideration - because it's the only schema exposed via the REST API which is generally available.

Also: Make sure that permissions for those functions called are set properly. It's also necessary to create the function you want to use BEFORE you generate the

GUI. Otherwise the metadata of the function won't be visible on the API side. In future versions of CYPEX, that won't be necessary anymore.

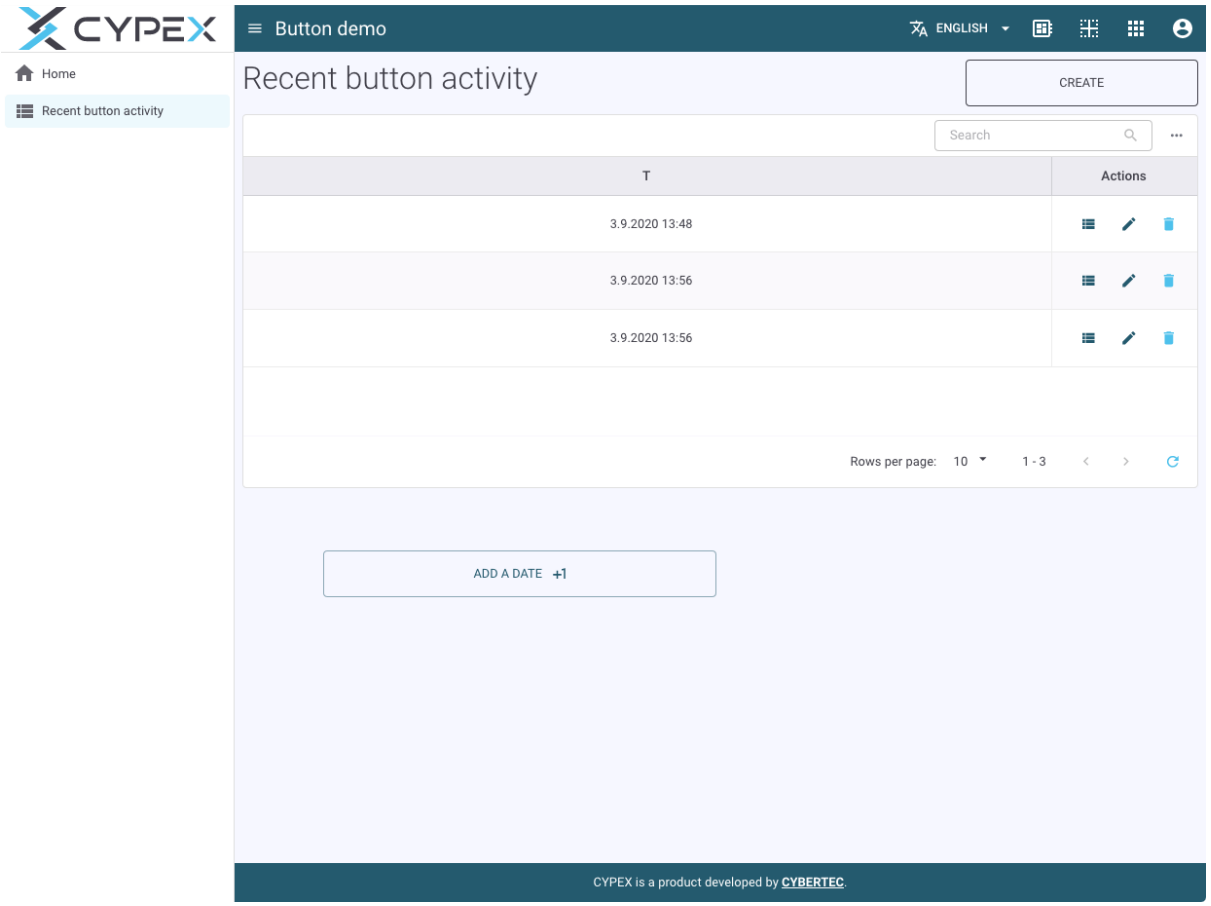
After these preparations have been completed, you can enter edit-mode, select the "call button" element, and add it to your app.



Note the name of the function. Make sure that you choose the right function. There is no need for parentheses.

Also: The name of the schema isn't relevant - CYPEX knows that the function has to be in the cypex_generated schema. In our case, no arguments are needed. (If arguments are needed, specify them in the argument list.) Finally, add a label and select a nice icon. Voilà, you have just created your first button and your first server-side business logic.

The end product looks like this:



Buttons are useful to trigger server-side business logic such as aggregations. But you can also directly impact your workflows. Sometimes, more complex operations are needed. Triggers are the best and most appropriate way to make that happen.

Scheduling jobs and notifications in CYPEX

CYPEX is in charge of handling everything from rapid prototyping to full application development.
When building a full application, it can become necessary to schedule jobs. CYPEX offers the means to make that happen using pg_timetable, a job scheduler developed by CYBERTEC. It's able to handle all kinds of job execution tasks.

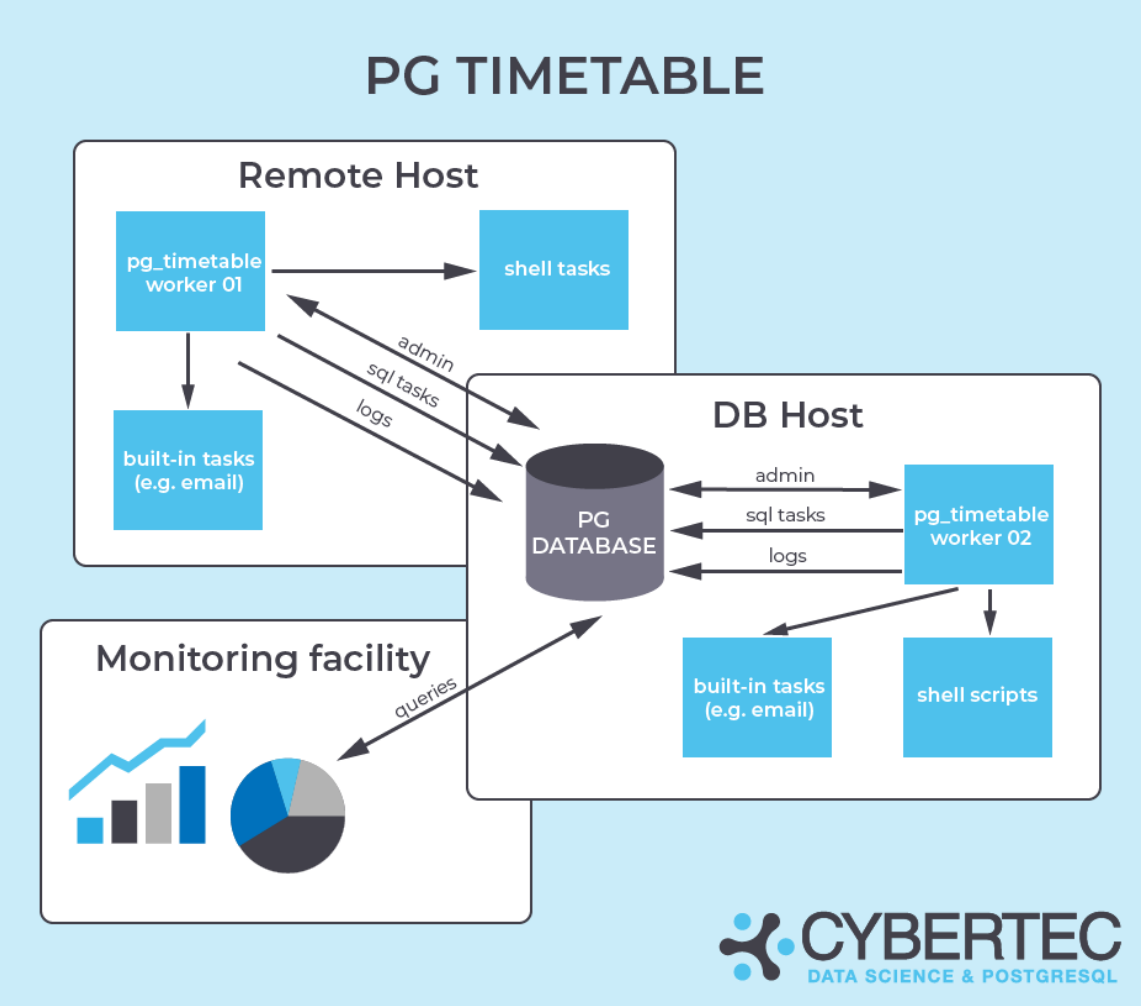
Let's take a look at a sample use case:

- When a contract is entered, somebody else should be notified
- If there is no response, try again in two weeks

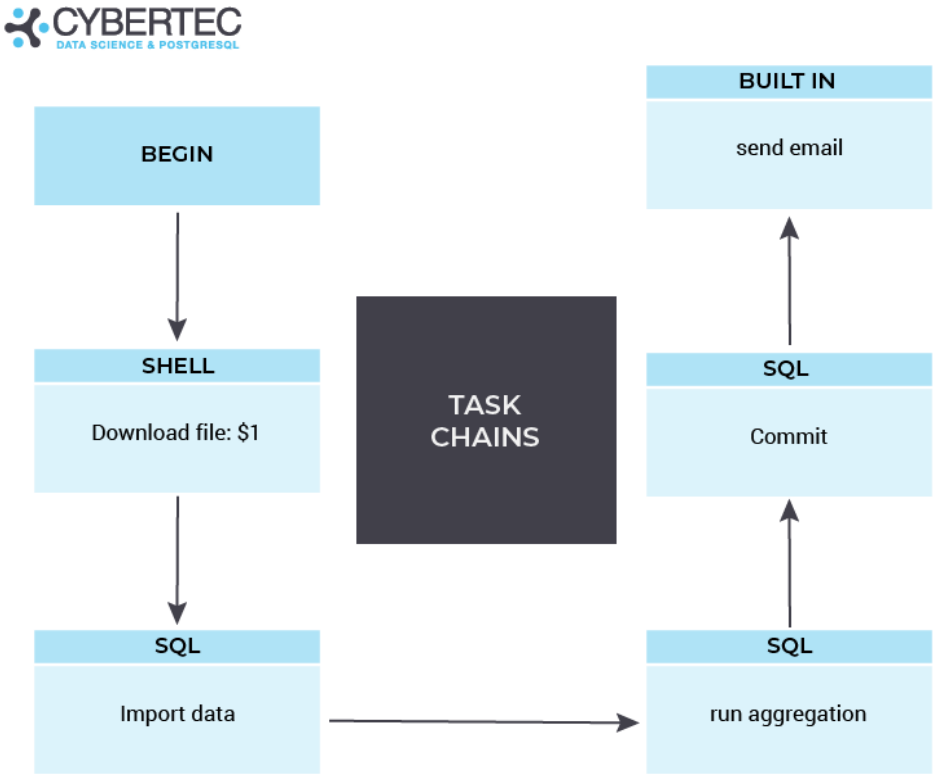
The way to integrate job scheduling with CYPEX is by using standard SQL tables. In pg_timetable, every job is stored in tables. By writing database-side code, you have a transactional way of scheduling jobs.

pg_timetable architecture

Before you explore further, you need to get familiar with the basic architecture of pg_timetable:



All configuration data is stored in tables, which allows you to model fairly complex operations:



Note that shell operations are only possible when running CYPEX outside of a cloud context. When starting pg_timetable, you can set a switch to control this behavior.

Scheduling jobs

If you want to learn more about pg_timetable, please see the official pg_timetable documentation to get more information about the basic processes.

Handling notifications

Notifications and job scheduling often go hand-in-hand. In CYPEX all notifications are stored in tables. What we want to achieve are:

- Full transactional semantics
- Being able to have everything in one backup
- Easy integration.

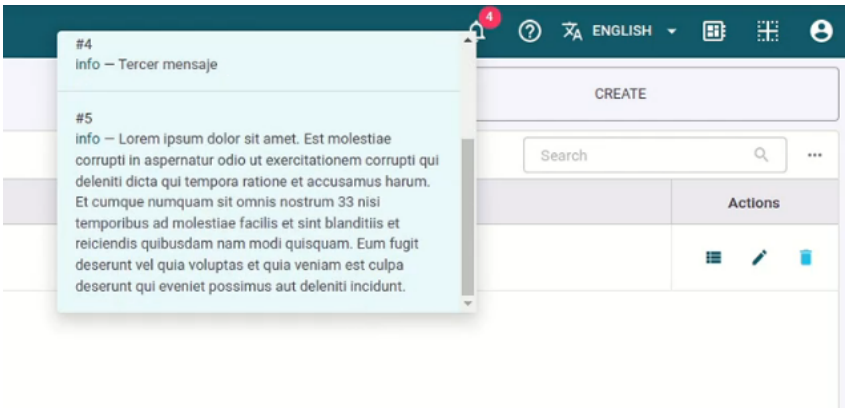
The data structure looks as follows:

t_notification		[table]
id		bigserial[19]
created_at		timestamptz[35,6]
created_by_user		int8[19]
created_by_role		text[2147483647]
recipient		text[2147483647]
level		text[2147483647]
message		text[2147483647]
target		text[2147483647]
read_at		timestamptz[35,6]
< 0	0 rows	0 >

To send a notification to an end user, all you have to do is to call a server-side function:

```
CREATE FUNCTION cypex.create_notification (
  recipient int8,
  message text,
  level text DEFAULT 'info',
  target text DEFAULT 'gui'
)
RETURNS void
AS $$
BEGIN
  INSERT INTO cypex.t_notification(recipient, message, level, target)
       VALUES (recipient, message, level, target);
END;
$$ LANGUAGE plpgsql;
```

The notification will be sent to the notification table and then displayed in the graphical user interface:



When the message is selected it will be marked as “read”. However, you can easily mark it unread using SQL queries (= UPDATE statement).

pg_timetable: Advanced job scheduling

pg_timetable is an Open Source job scheduler for PostgreSQL. It’s fully transactional, offers the ability to handle complex tasks and can be fully configured using standard database tables. pg_timetable is a core component of CYPEX - all configuration tables are automatically pre-installed and are therefore ready-to-use.

Why is pg_timetable part of CYPEX in the first place?
The reason is that CYPEX needs scheduling capabilities to handle various important things such as but not limited to:

- Asynchronous execution
- Notifications
- Sending emails
- Job scheduling

Let's discuss those tasks in more detail:

Asynchronous execution

Often users want to run long operations. Just imagine some data pre-aggregation which might take 20 minutes to complete. The problem is: If you have a button in a CYPEX UI you’ll face timeouts and many other usability-related issues which can cause inconvenience. The solution to the problem is asynchronous execution.

How can you do that? pg_timetable has a feature which allows for the execution of “self-destructing chains”. This type of chain is executed only once and is then removed by the system. In case the execution is interrupted, pg_timetable will try again. All you have to do to run a chain asynchronously (single execution) is to write a server-side function which schedules a pg_timetable job. Your GUI will then simply call this quick function and wait for pg_timetable to handle things asynchronously.

Your server side function can do whatever is needed. It can schedule the task to execute what your business logic requires, send an email or issue a notification when the task is complete.

On the GUI side all you need is a button, a state change or some other operation capable of scheduling a job.

The following example contains a simple method to create a self-destructing chain:

```
CREATE OR REPLACE FUNCTION raise_func(text)
  RETURNS void LANGUAGE plpgsql AS
$BODY$
BEGIN
   RAISE NOTICE '%', $1;
END;
$BODY$;

SELECT timetable.add_job(
    job_name          => 'notify then destruct',
    job_schedule      => '* * * * *',
    job_command       => 'SELECT raise_func($1)',
    job_parameters    => '[ "Ahoj from self destruct task" ]'::jsonb,
    job_kind          => 'SQL'::timetable.command_kind,
    job_live          => TRUE,
    job_self_destruct => TRUE
) as chain_id;
```

Notifications

This leads us directly to the next important topic: notifications. As you have already seen all your chain has to do is to send a simple INSERT:

```
INSERT INTO cypex.t_notification(recipient, message, level, target)
       VALUES (recipient, message, level, target);
```

This is enough to send a notification. Note that in PostgreSQL all notifications are fully transactional. For all practical purposes, this means that the notification is issued on COMMIT to ensure that the message isn't seen too early and to avoid race conditions.

Sending emails

Sending email is of great importance. pg_timetable and consequently CYPEX offer this vital capability.

The following example shows how such a job can be scheduled by server-side code:

```
DO $$
```

```
        -- An example for using the SendMail task.
DECLARE
        v_mail_task_id bigint;
        v_log_task_id bigint;
        v_chain_id bigint;
BEGIN
        -- Get the chain id
        INSERT INTO timetable.chain (chain_name, max_instances, live) VALUES ('Send Mail', 1, TRUE)
        RETURNING chain_id INTO v_chain_id;

        -- Add SendMail task
        INSERT INTO timetable.task (chain_id, task_order, kind, command)
        SELECT v_chain_id, 10, 'BUILTIN', 'SendMail'
        RETURNING task_id INTO v_mail_task_id;

        -- Create the parameters for the SensMail task
                -- "username":          The username used for authenticating on the mail server
                -- "password":        The password used for authenticating on the mail server
                -- "serverhost":      The IP address or hostname of the mail server
                -- "serverport":      The port of the mail server
                -- "senderaddr":      The email that will appear as the sender
                -- "ccaddr":            String array of the recipients(Cc) email addresses
                -- "bccaddr":           String array of the recipients(Bcc) email addresses
                -- "toaddr":          String array of the recipients(To) email addresses
                -- "subject":           Subject of the email
                -- "attachment":      String array of the attachments (local file)
                -- "attachmentdata":  Pairs of name and base64-encoded content
                -- "msgbody":           The body of the email

        INSERT INTO timetable.parameter (task_id, order_id, value)
                VALUES (v_mail_task_id, 1, '{
                                        "username":     "user@example.com",
                                        "password":                "password",
                                        "serverhost":     "smtp.example.com",
                                        "serverport":     587,
                                        "senderaddr":   "user@example.com",
                                        "ccaddr":                  ["recipient_cc@example.com"],
                                        "bccaddr":                 ["recipient_bcc@example.com"],
                                        "toaddr":       ["recipient@example.com"],
                                        "subject":                 "pg_timetable - No Reply",
                                        "attachment":   ["D:\\Go stuff\\Books\\Concurrency in Go.pdf","D:\\Go
stuff\\Books\\The Way To Go.pdf"],
                "attachmentdata": [{"name": "File.txt", "base64data": "RmlsZSBDb250ZW50"}],
                                        "msgbody":                 "<b>Hello User,</b> <p>I got some Go books for
you enjoy</p> <i>pg_timetable</i>!"
                                        }'::jsonb);

        -- Add Log task and make it the last task using `task_order` column (=30)
        INSERT INTO timetable.task (chain_id, task_order, kind, command)
        SELECT v_chain_id, 30, 'BUILTIN', 'Log'
        RETURNING task_id INTO v_log_task_id;

        -- Add housekeeping task, that will delete sent mail and update parameter for the
        -- previous logging task
        -- Since we're using a special add_task() function we don't need to specify the `chain_id`.
        -- Function will take the same `chain_id` from the parent task, SendMail in this particular case
        PERFORM timetable.add_task(
                kind => 'SQL',
                parent_id => v_mail_task_id,
                command => format(
$query$ WITH sent_mail(toaddr) AS (DELETE FROM timetable.parameter WHERE task_id = %s
        RETURNING value->>'username')
INSERT INTO timetable.parameter (task_id, order_id, value)
SELECT %s, 1, to_jsonb('Sent emails to: ' || string_agg(sent_mail.toaddr, ';'))
FROM sent_mail
ON CONFLICT (task_id, order_id) DO UPDATE SET value = EXCLUDED.value$query$,
                                        v_mail_task_id, v_log_task_id
                        ),
                order_delta => 10
```

```
        );

-- In the end we should have something like this. Note, that even Log task was created earlier
-- it will be executed later
-- due to the `task_order` column.

-- timetable=> SELECT task_id, chain_id, kind, left(command, 50) FROM timetable.task ORDER BY task_order;
--  task_id | chain_id | task_order |  kind   |                                left
-- ---------+----------+------------+---------+-----------------------------------------------------------------
--       45 |       24 |         10 | BUILTIN | SendMail
--       47 |       24 |         20 | SQL     | WITH sent_mail(toaddr) AS (DELETE FROM timetable.p
--       46 |       24 |         30 | BUILTIN | Log
-- (3 rows)

END;
$$
LANGUAGE PLPGSQL;
```

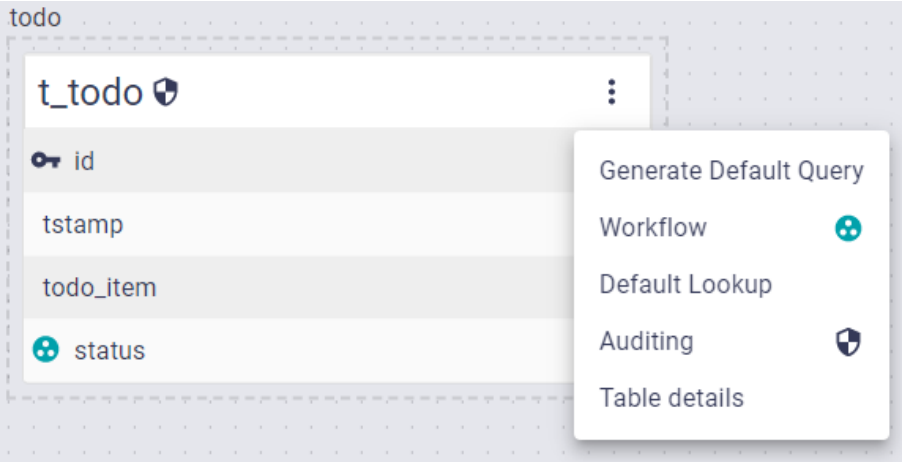
Job scheduling

You can schedule normal jobs which are to be executed repeatedly or at a given point in time. We recommend checking out the pg_timetable documentation to learn more about job scheduling.

Tracking history

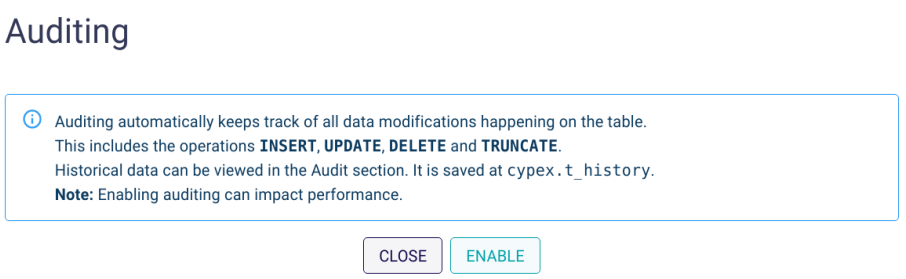
CYPEX may be used to store highly critical data. In those cases, it's necessary to track changes made to an entity. However, it's not only about critical data - sometimes you simply want to debug an application and check what's going on.

Enabling history tracking is easy: Go to the database setup page, and select the entity you want to track. Use the "Auditing" button to control this behavior.

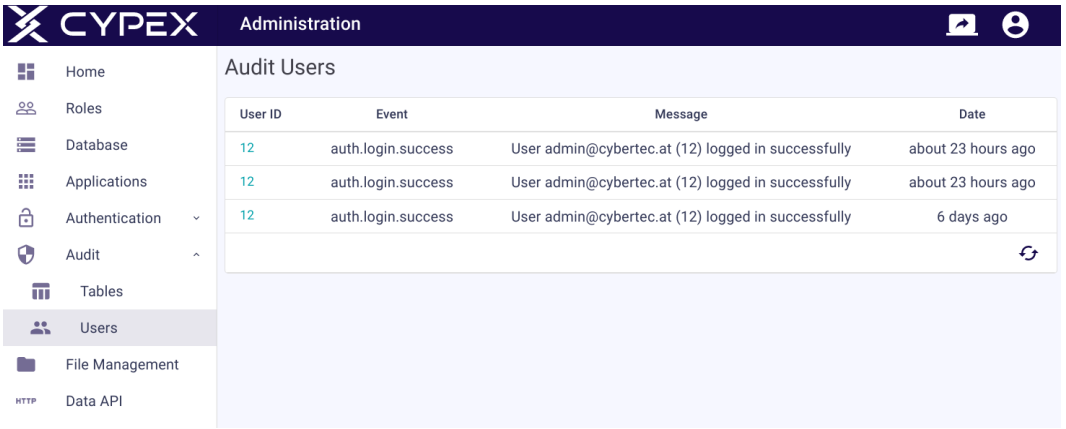


Once this is done, the table is tracked by CYPEX. In the background, a couple of changelog triggers are deployed, which store all changes made to the desired objects in JSON format.

If you want to inspect these changes - and if you are a superuser - you can go to the admin panel and take a look at the data in detail:



It's the task of the administrator or the person in charge of the application to handle the cleanup. We strongly believe that audit data should not be deleted automatically. For that reason, it's necessary for end users to explicitly control what's deleted, when it's deleted and how it's deleted.

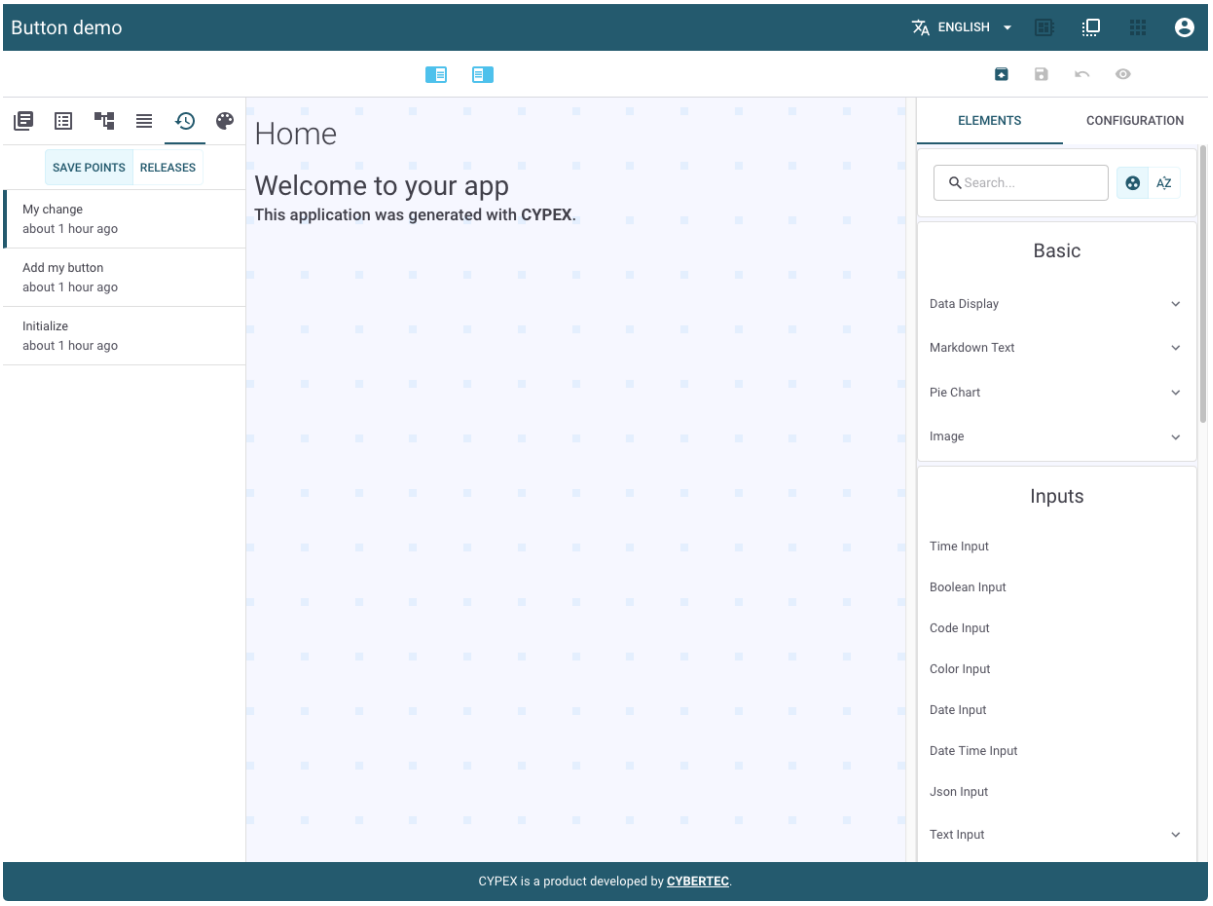


CYPEX GUI release management

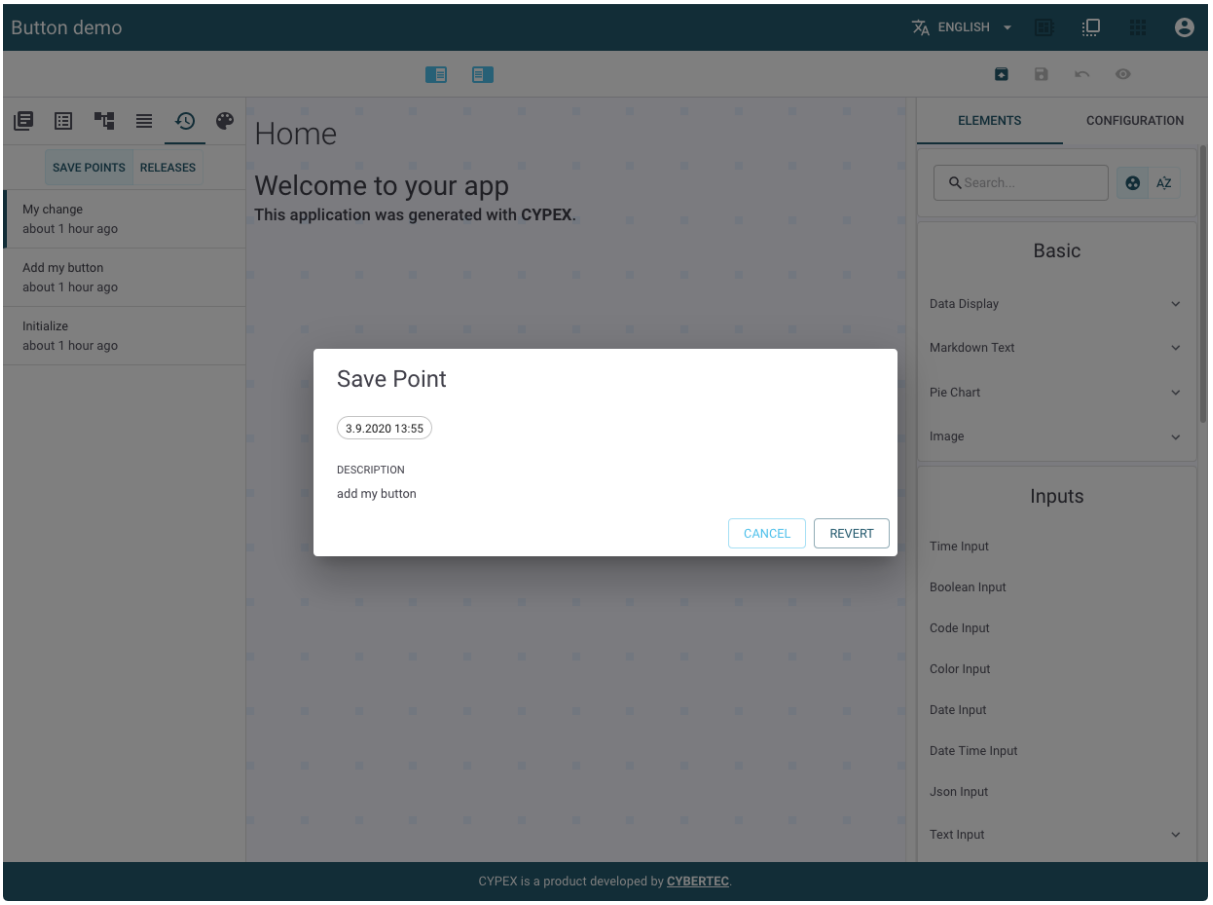
CYPEX allows superusers to make immediate changes. The edit-mode is only accessible to superusers.

However, in some cases you might want to change the application without actually using it immediately. To achieve live editing without harming productive users currently working with the application, you'll need to use release management. Before we dig into that, it's worth pointing out that CYPEX actually allows you to revert to a previous version of your application.

Here's how it works:

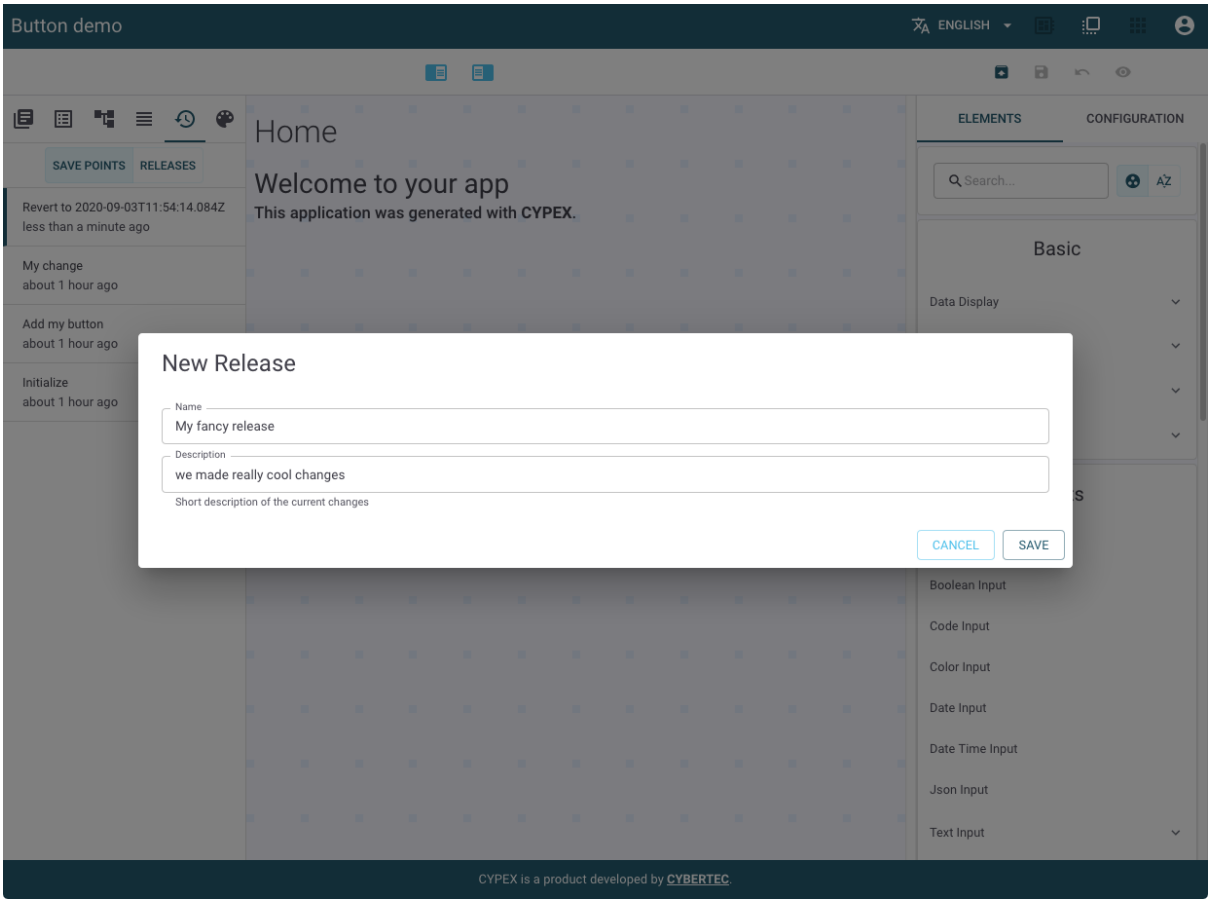


Your application's history can be seen in edit mode. By clicking on a previous version and confirming your request, you can go back to that version of your application.



This is exactly why we stressed earlier in the document that it's important to write proper comments in case you change your application. It makes it easier for you to go back and find the right release.

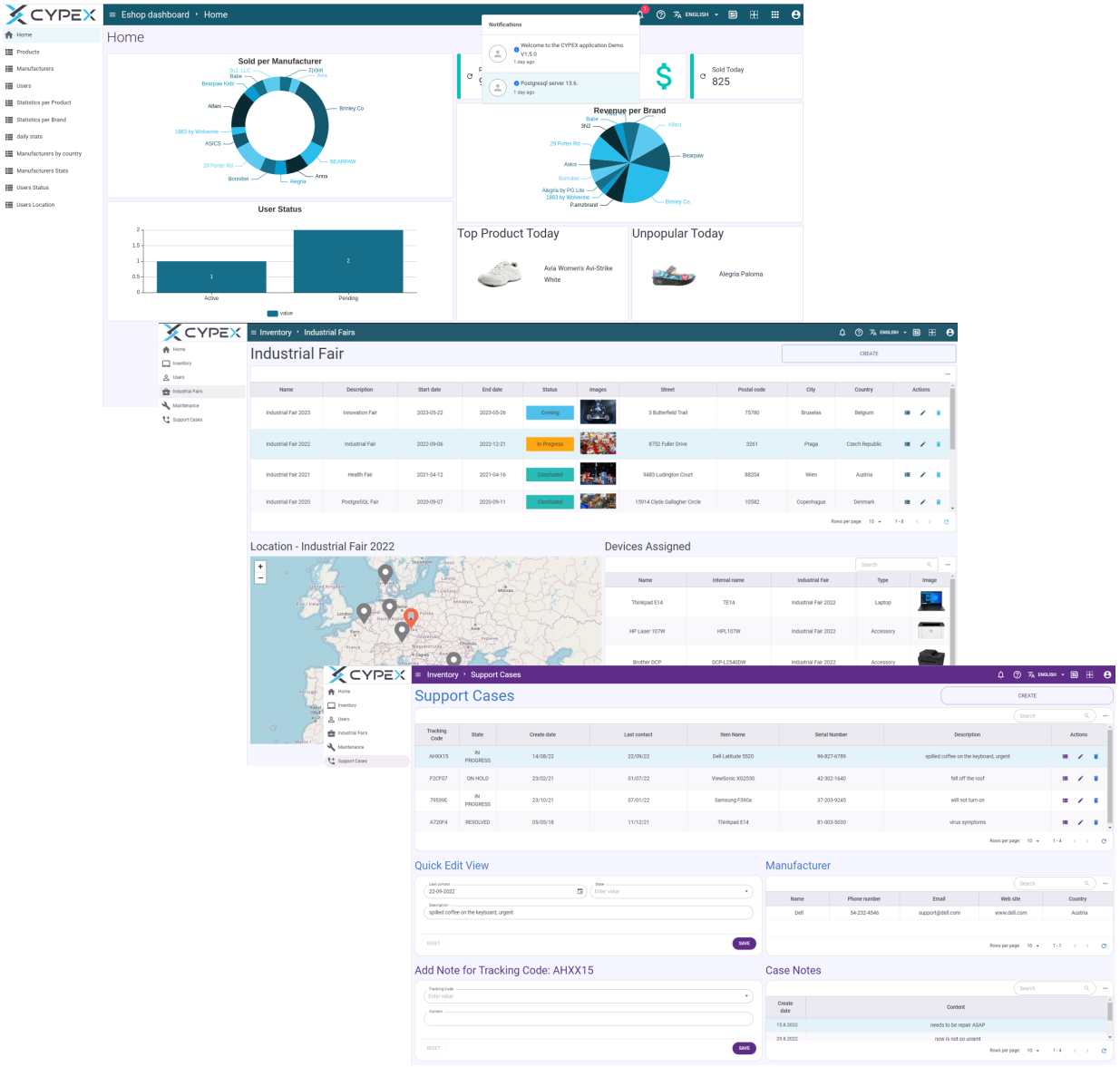
So far, you have seen how to make changes and how to revert them. To make a release, you have to click on the blue icon in the right upper corner of the app (next to the normal "save" button).



In the CYPEX GUI, versions are associated with users. That means it's possible to run various versions of the app in parallel - without any problems. Different users will see different variations of your solution - which might be exactly what you want, in case changes made to your GUI are highly critical and you don't want to risk breaking things in production.

Changing the layout of your application

Apart from the menu entry which allows you to switch back to a prior version, you can also make other adjustments to your application. You can change colors, upload logos and set the way corners will be displayed. In the future, the variety of changes that can be made will be expanded even more. Below are some example screenshots of what's currently possible:

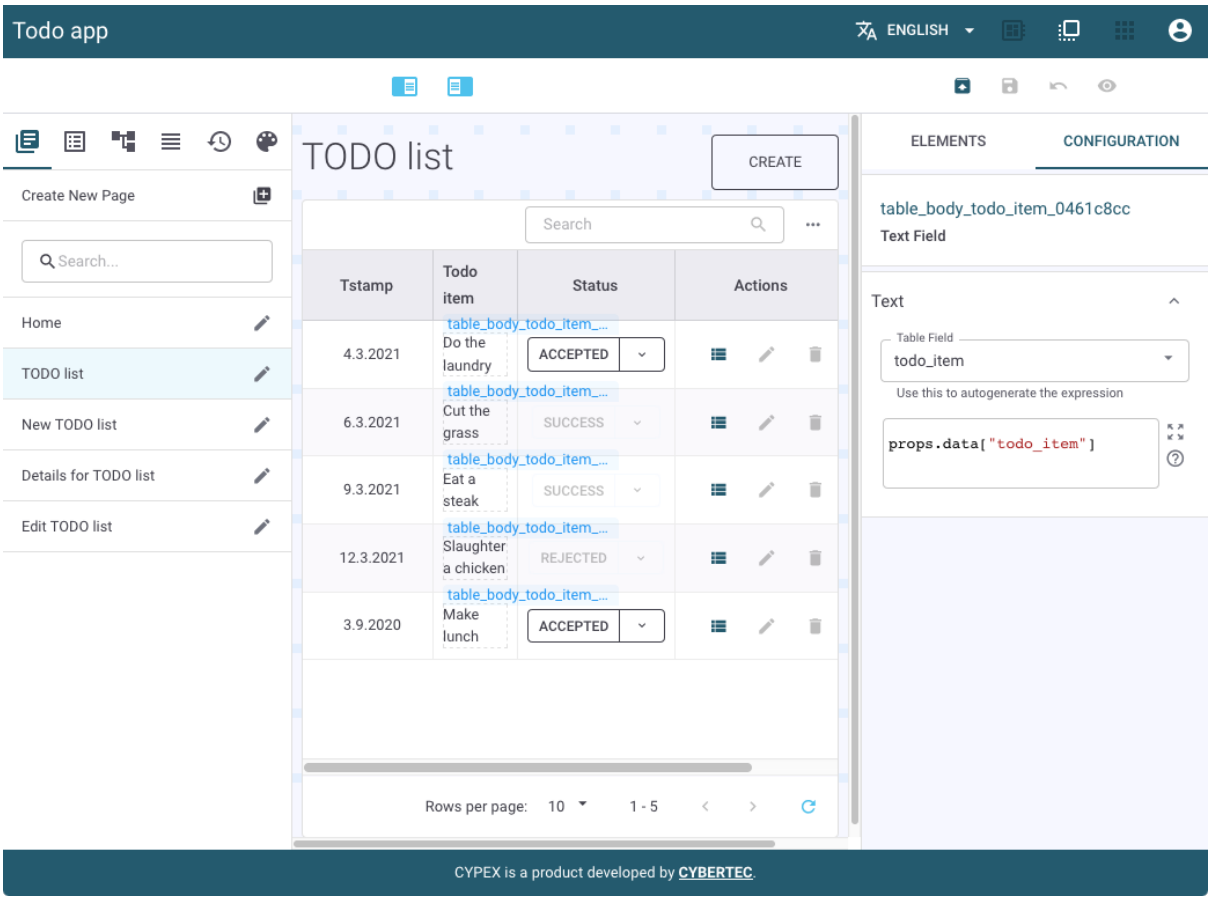


CYPEX built-in expressions

Let's come to a core concept of CYPEX: expressions. What you actually see in the GUI isn't just some static field, but in fact, a JavaScript expression which can be modified. This gives you a great deal of flexibility and allows you to tailor the GUI to your needs. Using JavaScript expressions, you have great power at your fingertips.

However, most people aren't heavy JavaScript users and therefore a lot of the more common tasks have been simplified by adding graphical shortcuts. One of these shortcuts was shown before: ID resolution. What the GUI element actually does is to modify the underlying JavaScript relation in the desired way.

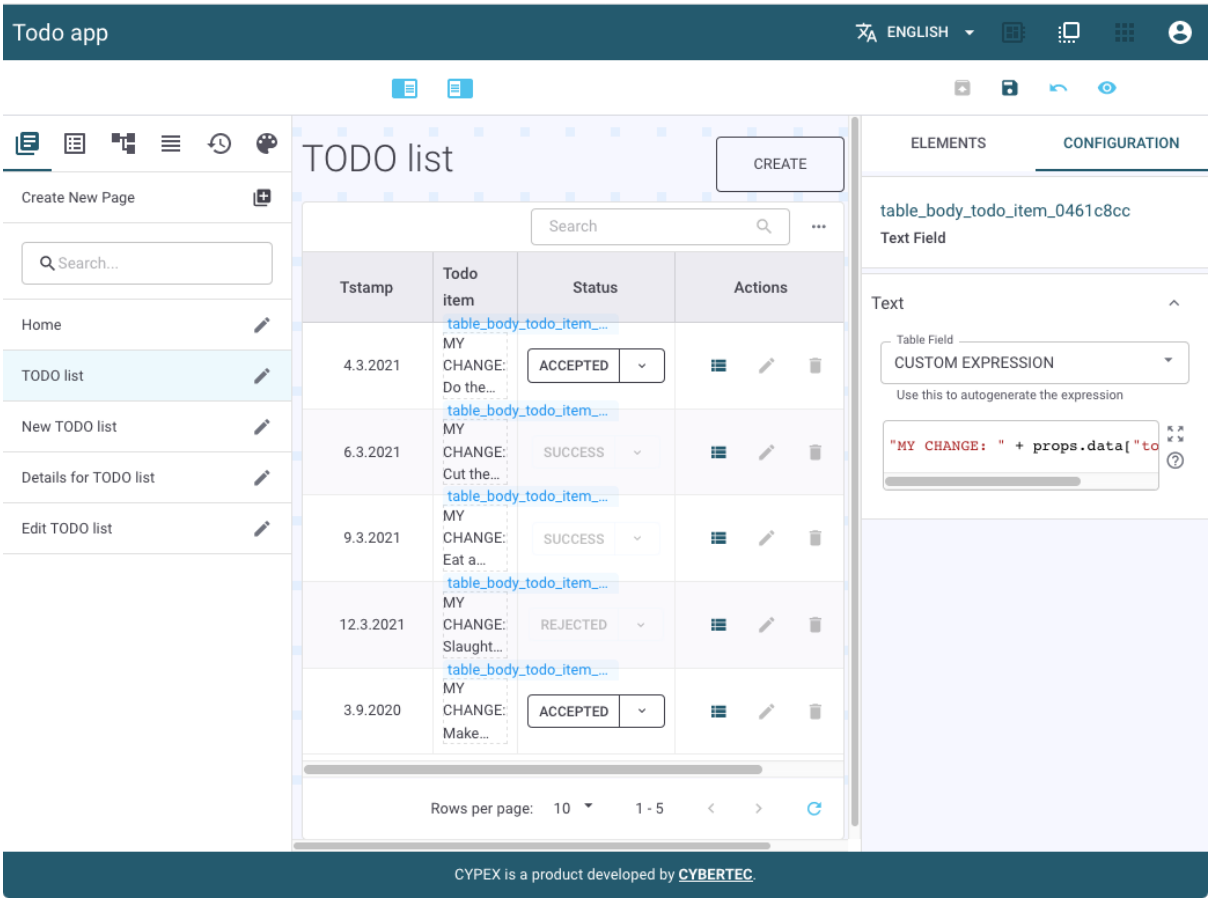
But let's not get lost in technical details: Let's move forward and see what you can do in real life to build more useful applications.



To show how things work, look at the TODO list built in one of the previous chapters. Go to edit mode, and click on the "TODO item" column in your main table. If you look closely, you'll see "`props.data`". This is the JavaScript expression mentioned a moment ago.

In this case, the column is supposed to display the "todo_item" element coming from the backend. But you can modify that - you can apply basically any expression to this data.

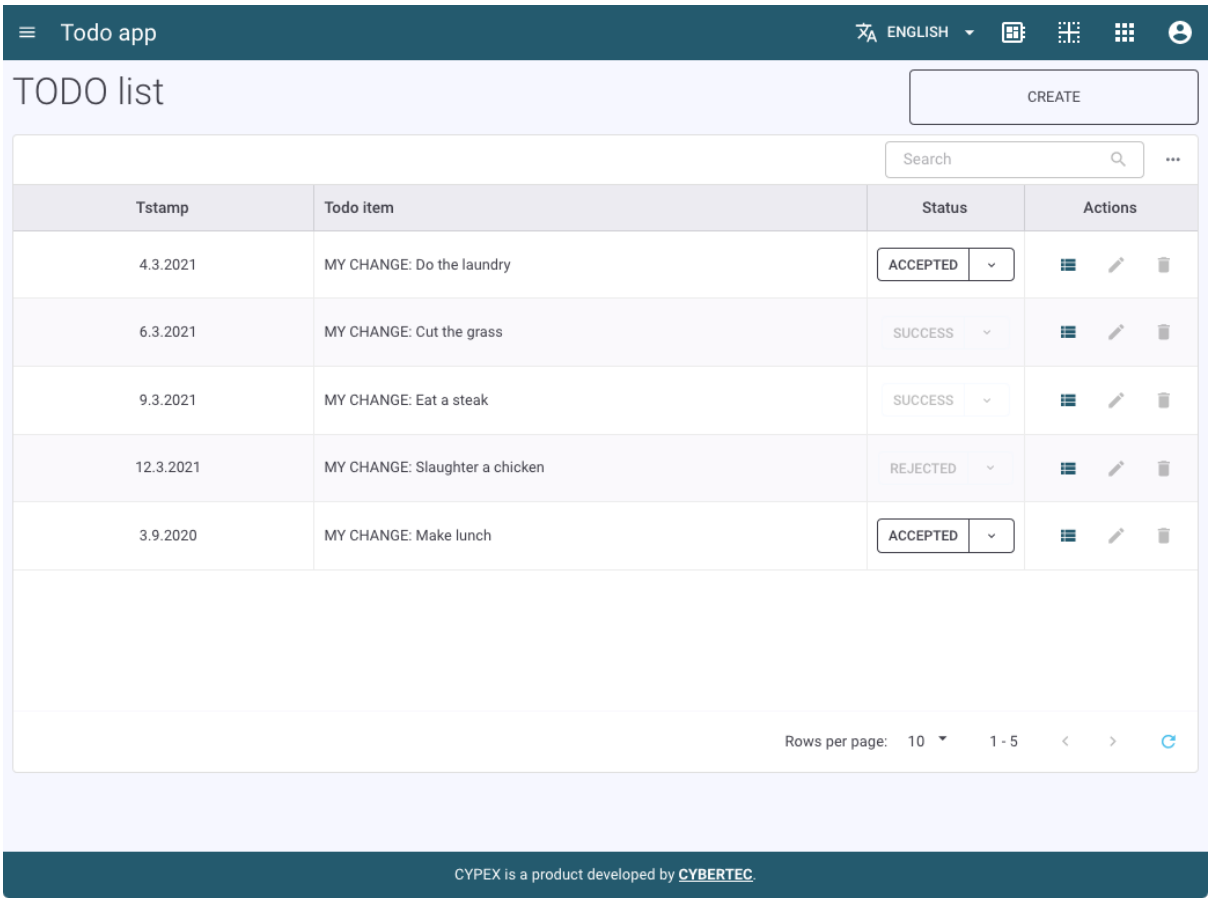
For the sake of simplicity, let's add a prefix to the content of the column:



What we're using here is pure JavaScript code.

"MY CHANGE: " will be used as a prefix. If you have a basic knowledge of JavaScript, you'll be able to do really powerful things using simple expressions.

The final product will look as follows:



The way data is displayed has been changed on the fly. The workflow stays unchanged. The data in the backend is also going to stay unchanged - we're only talking about the way CYPEX displays data.

CYPEX Custom Expressions

The CYPEX development team wants to provide our customers with the most flexible solution possible. We've visualized abstract tools to make them as easy to use as possible. However, many applications need more than just display elements which put a 1:1 copy of data on the screen. To make a truly beautiful application, it's necessary to add format options, dependencies and a lot more.

The solution to the problem of providing end users with a powerful and easy-to-use GUI is the introduction of "custom expressions". In the GUI, most elements can be fine-tuned by using custom JavaScript code. Why is that necessary? Here are some examples:

- Hiding or showing elements depending on a value in a data source
- Applying colors which depend on the content of a variable
- Calculating values on the fly

Of course there are many more examples proving why expressions make sense. In this section, we'll take a look at custom expressions and understand how they can be used.

Basic "custom expression" concepts

In CYPEX, each element on the page has access to its selectors. So what are selectors? Let's dive in and find out. Selectors are predefined JavaScript objects with properties and values. CYPEX uses selectors to make elements on the page interact with each other in a controlled way.

Many GUI elements allow for custom expressions. The configuration editors provide a way to define "Custom Expressions" as input. Depending on the element configuration, this input should return a value, e.g., string, number, array, object, or function. We have documented the required value for each element to make it easier for developers to adjust the configuration. We recommend checking out our video tutorial series.

Accessible JavaScript objects

Location

The first thing to understand is how to navigate inside the page. There are many variables which are of key importance. These can be used to figure out where we are and how to navigate through the application. Let's inspect these variables in more detail to figure out how it all works, and what is possible:

location.pathname

`pathname` is a string which contains the URL's path for the location, which will be an empty string if there is no path.

location.queries

This variable is a string containing a '?' followed by the parameters of the URL. In CYPEX this is also an object containing arguments.

page

This object represents the current page of the application.

page.id

Identifier of the current page.

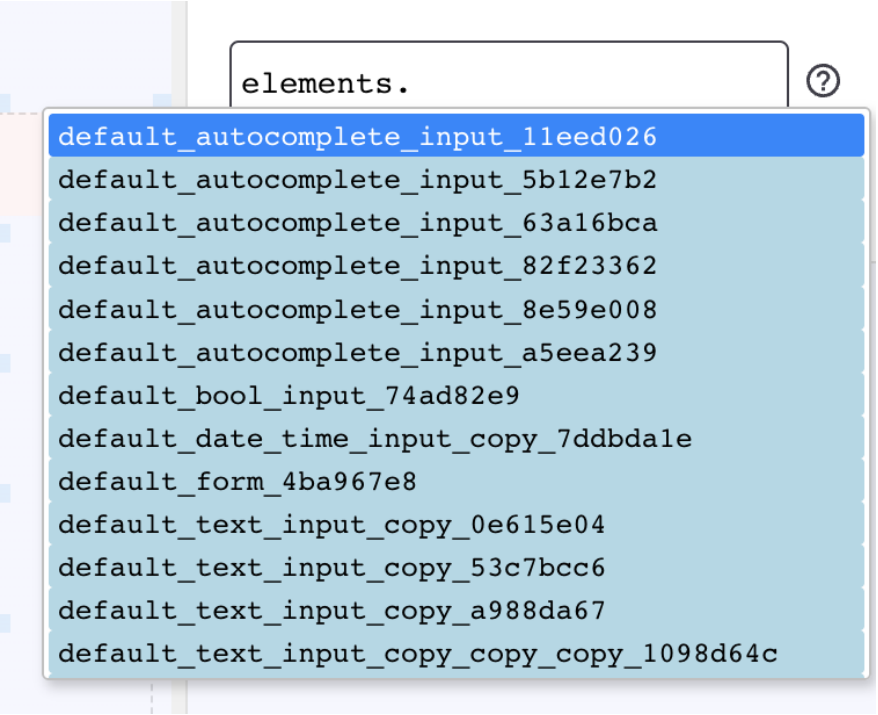
page.loadedAt

Date and time of the last page load.

elements

Elements is an object which contains all elements located on the current page. To access the element selector, the element identifier should be picked from the list, e.g., `elements.<element_id>`.

Here is an example showing all element names on the page. Mind that there is tab-completion at work. Simple type “elements” and CYPEX will immediately display all variables inside the object:



element

The interface of the element itself. It can be used in the custom expression field for the current scope of the element. For example: **element.value**

element.i18n

An object containing translated texts in the current language, e.g., title, label, etc. for this element

props

The properties passed down by the parent element, for example, table or form, so their child elements will have access to props.data (= variable containing data elements)

lodash

A modern JavaScript utility library delivering modularity and performance can be used inside the "Custom Expression" editor.

Check out Lodash for more information.

Chart Filter as “Custom Expression“

Server data can be filtered by any element selector value.
Advanced filters might look like this:

```
{
  "combinator": "AND",
  "filters": []
}
```

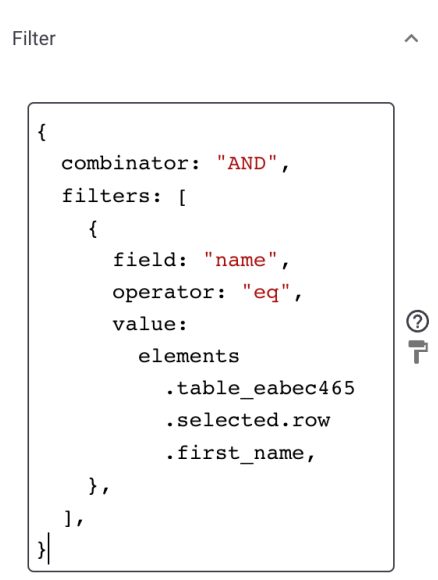
It uses the selected table row value as a column to filter.

Syntax: `elements.<table_id>.selected.row.<column_name>.`

To create advanced filters, CYPEX uses PostgREST, so the filters array must contain a collection of possible combinations like

```
{
    field: <column_name>,
    operator: "eq",
    value: elements.<table_id>.selected.row.<column_name>
 }
```

The following image contains a real-world example:



Values will be returned as strings.
The full list of PostgREST filters can be found here:
https://postgrest.org/en/stable/api.html?highlight=operators#operators

JavaScript

A scripting or programming language, running inside the web-browser that allows you to implement complex display logic and features for websites.

The web reference for JavaScript:
https://developer.mozilla.org/en-US/docs/Web/JavaScript

The following listing contains a little cheat sheet for your daily work:

Basics

Literal values

```
500                                  number
"Star Wars"                          string
true                                 boolean
[1, 2, 3]                            array
{success: "green", error: "red"}     object
```

Expressions

```
100 (+ - * / ) 2.5                   types of calculation
"Star " + "Wars"                     string concatenation
true && || false                     opperators
```

Inline conditionals

```
6 / 2 == 3
1 > 2 ? "success" : "error"
"Han Solo".endsWith("o")
```

You can use all modern JavaScript features available in your browser

```
"Luke Skywalker".split(" ")
["Luke", "Leia"][0]
{success: "green", error: "red"}["green"]
Object.keys({foo: 1, bar: 2})
```

Modern JavaScript features

```
({data:null}).data?.value

null ?? "fallback"

`${5 * 4} years old`
```

Available expressions as table child

1. Access a specific field of the current row

```
props.data["first_name"]
```

```
props.data["created_at"]
```

```
props.data["Name with non-alphanumeric characters!"]
```

2. Access all data

This can be used in a json field to visualize the whole row at once

```
props.data
```

Accessing Own Element Data

```
element.data
```

```
element.value
```

Accessing Other Elements Data

```
elements["some_markdown_field"].text
```

```
elements["some_table"].data
```

Accessing Props

```
data.props["name"]
```

Accessing Element Translations

```
i18n.text
```

Accessing The Location Object

```
location.query.identifier
```

Accessing The Page

```
page.loadedAt
```

In one of the previous sections, you learned that there are actually two ways to resolve IDs in a relational model: a.) use queries and joins or b.) fix things on the client side, using expressions. The beauty of expressions is that things are usually far easier to handle, since you don't have to touch the database at all. However, you also need to keep an eye on performance. Depending on your situation, one or the other might result in faster performance.

Displaying elements conditionally

All examples shown in this tutorial so far rely on the fact that elements have always been shown - regardless of the the situation on the page and the data displayed. In reality this isn't always the case. Sometimes it's necessary to show elements only in certain situations.

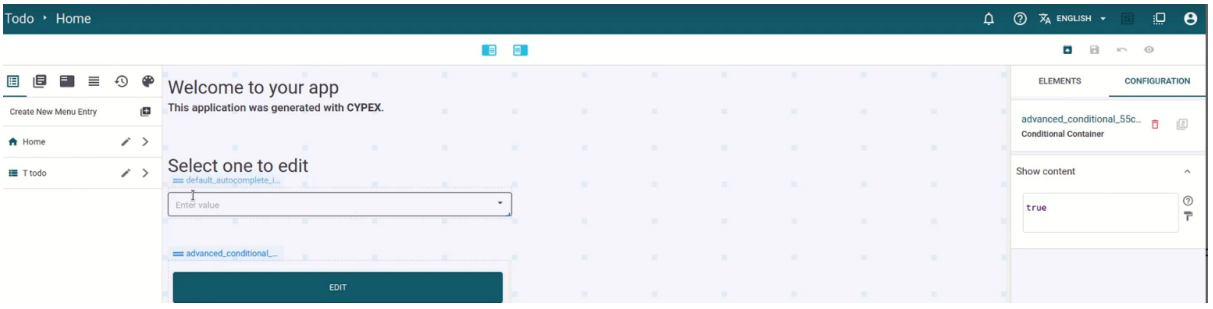
What are examples of this? Suppose you only want to display an image in case some checkbox is ticked. Or maybe you want to display a button, but only when some fields are filled out. There are countless scenarios where you need conditional elements.

CYPEX supports the notion of a conditional container. What that means is that it's possible to use a condition to display a group of elements which depend on that condition.

Hiding a button conditionally

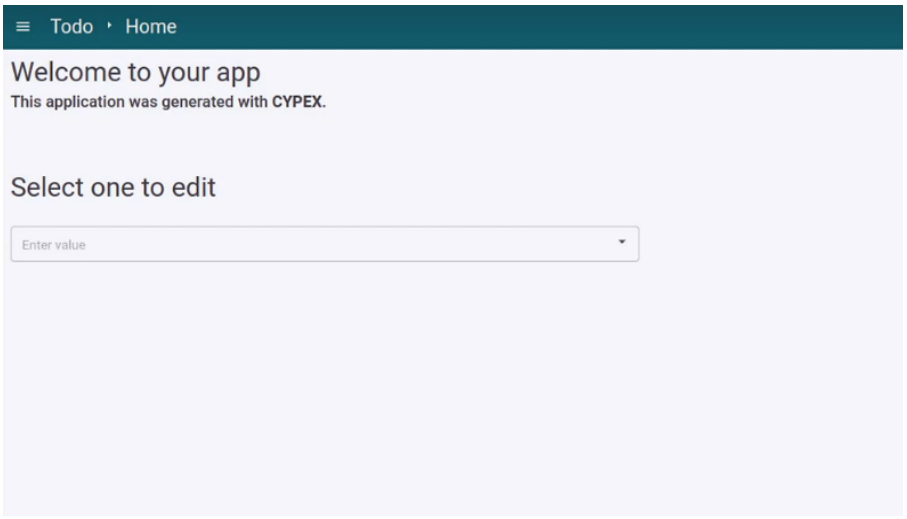
Let's see how it works and see how to hide a button. The goal is: the "Edit" button should only be visible if a value in a dropdown has been selected. If there's no value, the button should be hidden.

The way to do that is by adding a "Conditional Container" element to the GUI. The elements you want to show / hide can then be added to this element. Then you need to assign a JavaScript expression to the element. In case it returns true, everything is visible:

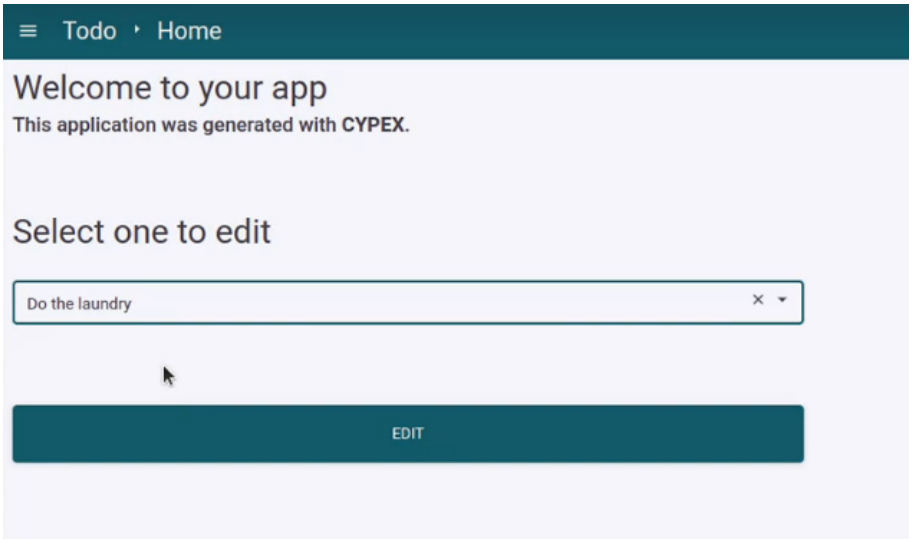


However, if this expression does not return true, but false, the elements in the container will be hidden from view. The advantage is that you can basically access all elements on the page and use those values to control this kind of behavior.

Before you take a look at the expression you need to put into the "Show content field", you can see what the desired output looks like:



No value has been selected and therefore the button is hidden. As soon as you select a value, the button will be displayed:



Let's take a look at the JavaScript expression we need to use:

```
!!elements.default_autocomplete_input_2bedb141.value
```

This expression is sure to return true or false. But what does it actually mean? You can access all elements on the screen ("elements"). Every element on the page will automatically have a name. In this case CYPEX decided to call the element "default_autocomplete_input_2bedb141" (check the name of the element in the configuration window). Then you can access the value of this element. If it's there it returns true - if it isn't there, it returns false.

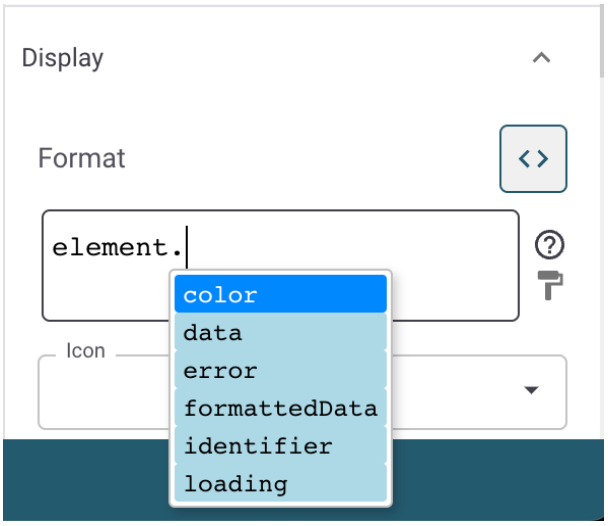
Almost any level of complexity is allowed here. All you have to do is to produce "true" or "false" to tell the container what to do.

List of element interfaces :

The following examples will show how you can make use of variables, access fields and information using the graphical editor.

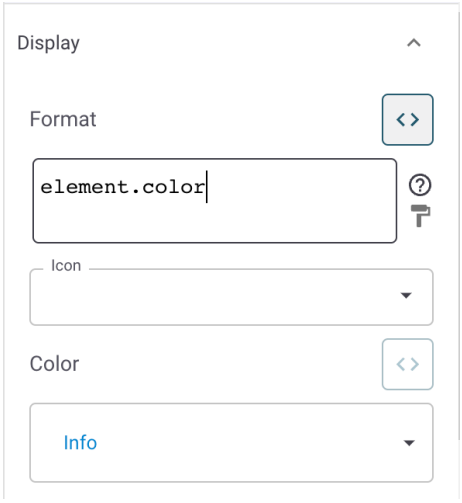
Data Display

`element.` itself or `elements.<data_dispaly_id>`.



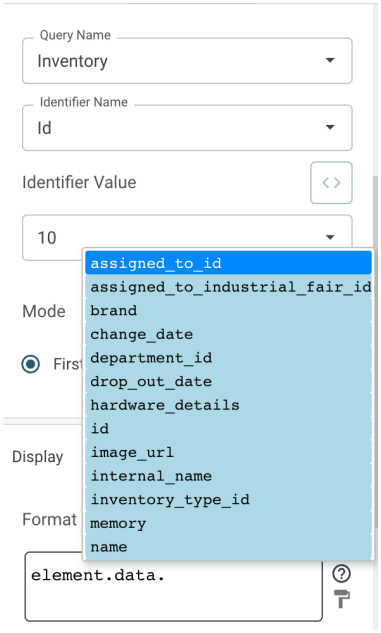
element.color

`color`: Access to element color, specified below.

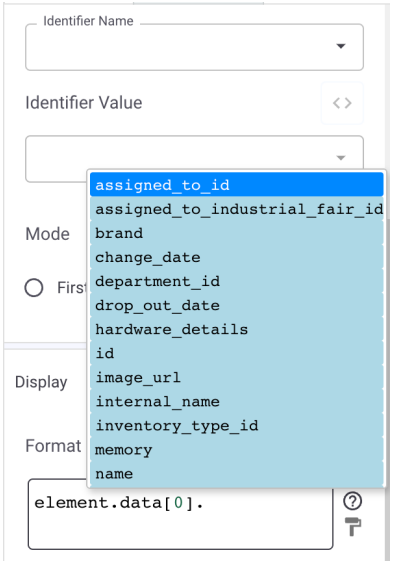


element.data

Access to the element data if “Query name“ is set as a data source. Depending on the configuration mode, the data can be an object or an array.



For the mode “First Row“ is `element.data` an object.



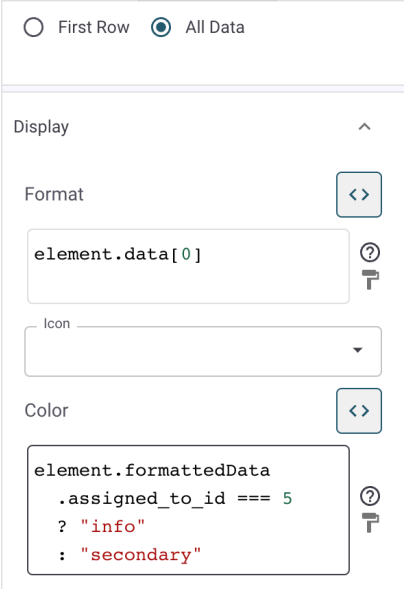
For the mode “All Data“, `element.data[ ]` is an array, use element index access to specific record e.g.

element.error

Since this element fetches data by the “Query“ name, the error message is accessible if the request fails.

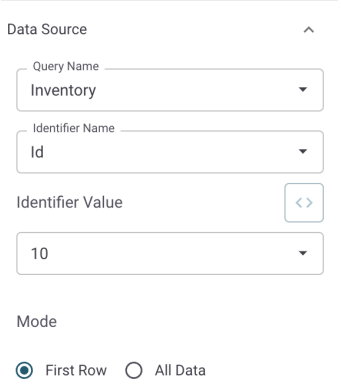
element.formattedData

Get the value of the preformatted data, e.g.,



element.identifier

Returns identifier value if it is set in the configuration



element.loading

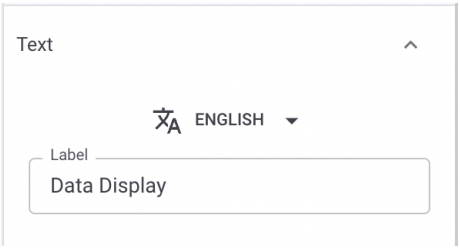
Boolean value. Indicates if the data is in the process of being loaded from the server

i18n

This section describes internationalization and multi-language support.

i18n.label

Returns a string containing the current translated label



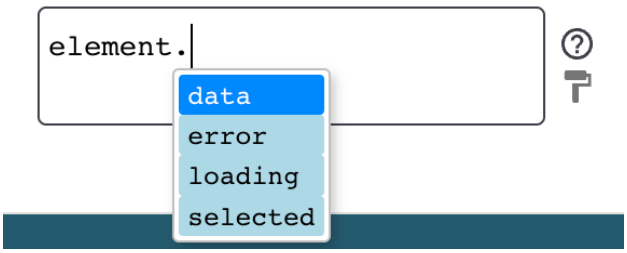
i18n.text or elements.<markdown_text_id>.i18n.text

Translated texts in the current language, returned by markdown editor

Pie / Bar / Line Chart

Let's focus on configuration parameters available to control charts.

`element` or `elements.<chart_id>`.



element.data

An array of records, server data fetched by “Query “ name.

element.error

Returns the error message if the request fails.

element.loading

The boolean value indicates if the data is in the process of being loaded from the server.

element.selected

Returns selected object(record) if selection in chart configuration is enabled.

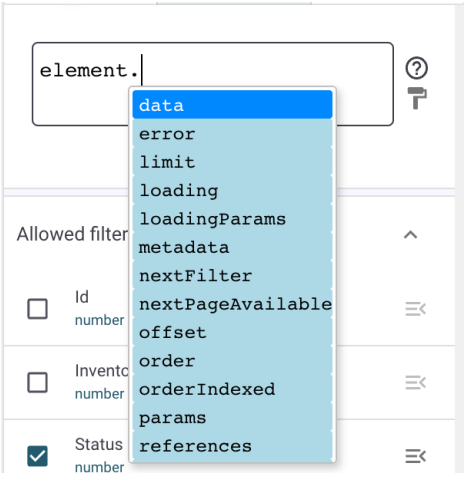
element.i18n.title

Returns a string that is translated into the current language “Title“.

Table

Tables also support custom expressions. This section describes which features are available and what can be done to make this important GUI element more powerful.

`element.` or `elements.<table_id>`.



element.data

Server data, fetch by setting the proper “Query Name“.

element.error

Returns an error message if the request fails.

element.limit

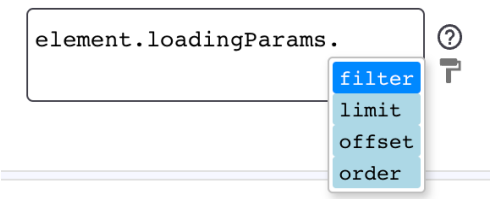
The number of rows per page.

element.loading

The boolean value indicates if the data is in the process of being loaded from the server.

element.loadingParams

A partisan object of the params set during data loading



element.loadingParams.filter

An object of advanced table filters, if it exists.

element.loading Params.limit

The number of max rows can be fetched.

element.loadingParams.offset

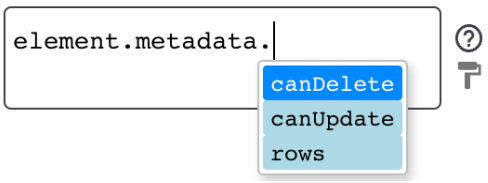
The number of rows to skip before beginning to return rows.

element.loading.order

An optional array of objects like

```
{
  fieldName: string;
  asc: boolean;
  hidden?: boolean;
}
```

element.metadata



element.metadata.canDelete

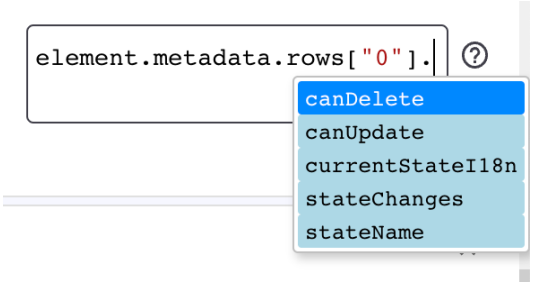
A boolean value, returns a value indicating if the user has sufficient permissions to delete records.

element.metadata.canUpdate

A boolean value, returns a value indicating if the user has sufficient permission to update records.

element.metadata.rows

Metadata related to each row / record where the property is a row key;



element.metadata.rows[0].canDelete and canUpdate

Specify permission for the current row.

element.metadata.rows[0].currentStateI18n

Translation object generated on the server during table creation

```
  .short_desc: string
  .title: string
  .long_desc: string
```

element.metadata.rows[0].stateChanges

An array of objects with possible workflows:

```
{
  to: string,          // workflow value
  i18n: object              // translation object, e.g., { title:
string }
}
```

element.metadata.rows[0].stateName

Is a column name which contains workflow values.

element.nextFilter

This is an advanced option. An object of advanced table filters is a filter object which can be fetched while the user is about to build the filter. You will need this to quickly preview output.

element.NextPageAvailable

Is a boolean value that shows if the last page has already been fetched or not.

element.offset and element.order

Same as in `loadingParams`. The only difference is that these values can be configured in the element editor.

element.orderIndexed

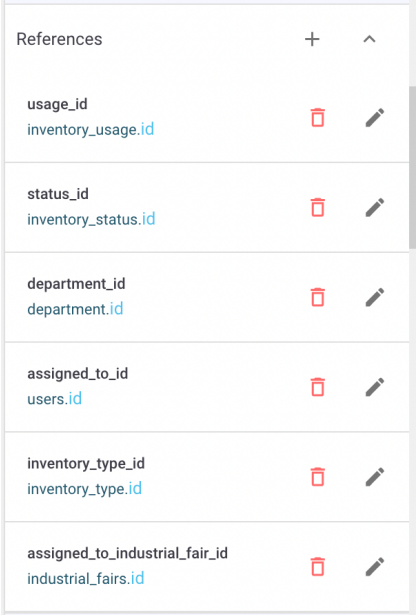
Has the same definition as `order`, but is used as a helper for column sorting.

element.params

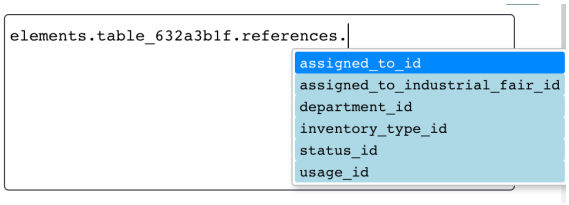
Also contains `filter`, `limit`, `offset`, and the order specified in the URL to fetch table data.

element.references

An object of joined tables (referenced table), if such tables exist (they do exist in case you use “default resolution” in the model builder). This configuration can be found in the “References“ section f the table editor, e.g.:



Each property or value is accessible in the “Custom Expression“ editor.
The property name is a referenced column name

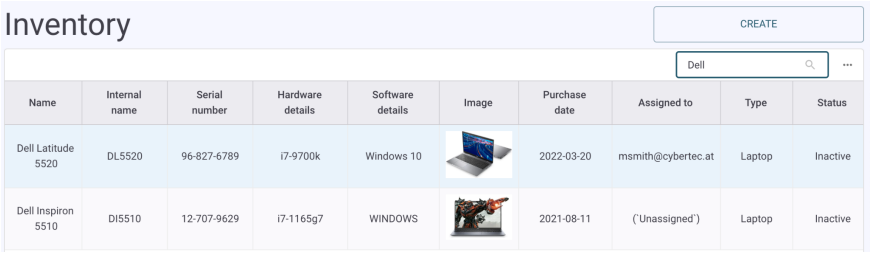


and the value is an object:

- `.viewName: string` (viewName: is the name of a joined table)
- `.identifierName: string` (identifierName: is the joined table identifier)

element.searchInputValue

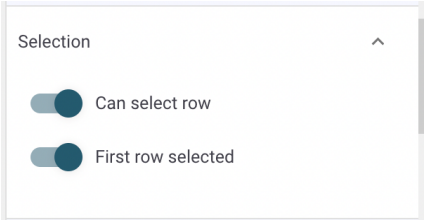
Access to the value of the table search input



Example: `element.searchInputValue = "Dell"`

element.selected

For tables with “Selection“ enabled selected value can be used as an expression:



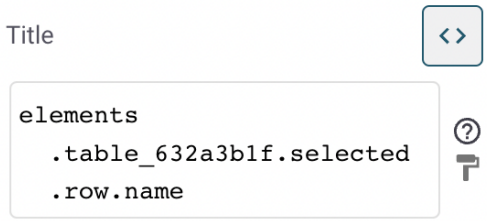
To get data of the whole table row, even to columns that aren’t displayed in the table but natively present in the “Query“, the following syntax should be used:

```
elements.<table_id>.selected.row.<column_name>
```

or for usage inside the element itself

```
element.selected.row.<column_name>
```

For example, to use the column “Name“ as a title for another element, which uses the “Custom Expression“ editor:



Note: If “First row selected“ isn't enabled and the row wasn’t clicked, it means the selected object is empty. In this situation, use “?”
which is a JavaScript operator to avoid errors if no value exists:

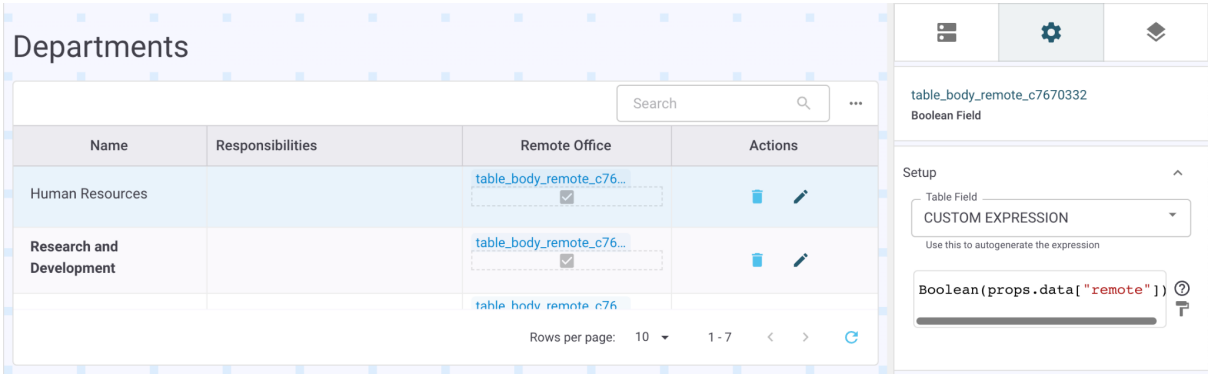
```
elements.<table_id>.selected?.row?.<column_name> ?? "Default Value"
```

“Default Value“ also can be an empty string.

Table columns

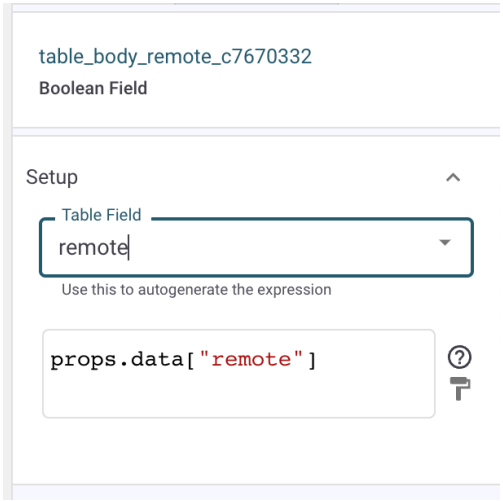
In the current version of CYPEX, table columns are elements but without external access. It's impossible to get a column value inside the “Custom Expression“ editor used by any other element on the page. However, every column type has access to the table data through the props key.

Here is an example:



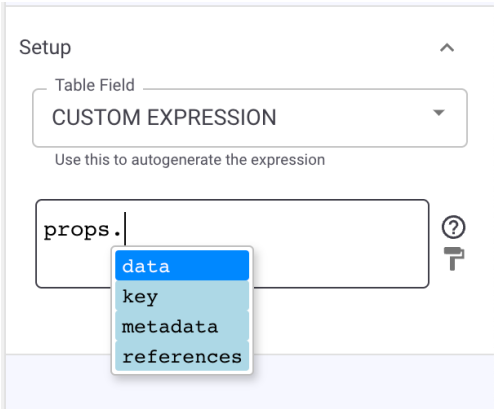
To get the data of a column inside the props objects, use the following syntax: `props.data["remote"]`. In this case we access the column called “remote” and fetch the idea. The Boolean() method will ensure that the value isn't null and not undefined.

Use “Custom expressions” to format the string or to adjust the output according to your needs. It’s also possible to pick the necessary column using autocomplete inside the “Text Field“, if adjustments aren't required.



Props object

The props object is one of the most fundamental building blocks of the “Custom expressions” machinery. It contains all object-related data, keys, metadata as well as references. It's the single most important object you must understand when working with CYPEX expressions in general:



props.data

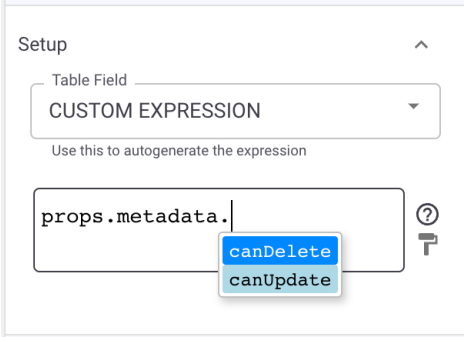
Row data object.

props.key

Row index.

props.metadata

Row metadata passed through from the table element

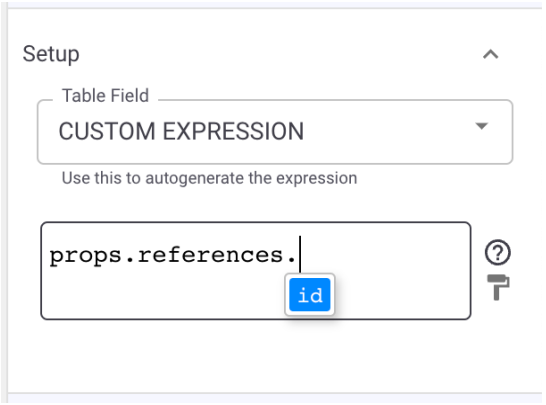


props.metadta.canDelete & props.metadata.canUpdate

Are boolean values defining whether the user has permission to delete or update?

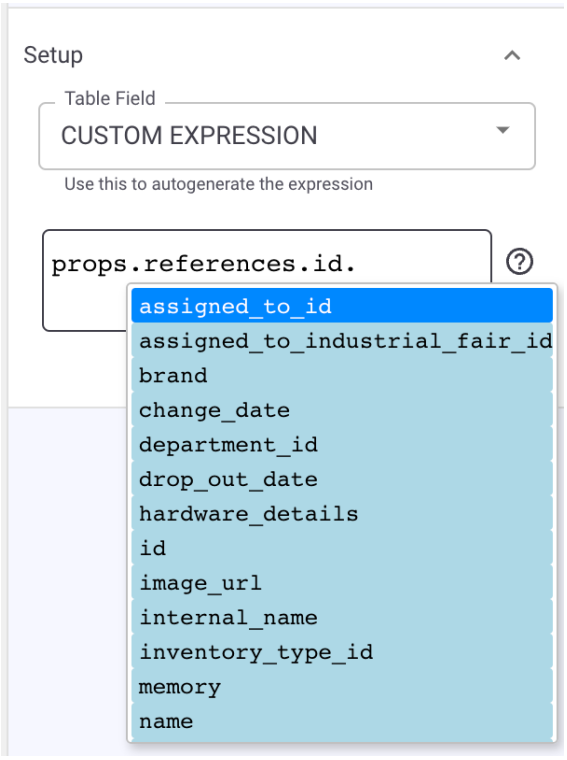
props.references

Only the parent table has any of the references(joined tables) configured; it's possible to get those references using the following method:



props.references.id

Specifies the source column in this example.
Keep in mind the whole referenced row will be returned as a value. Use autocomplete to select the desired field or use a JavaScript expression to access various fields as needed:

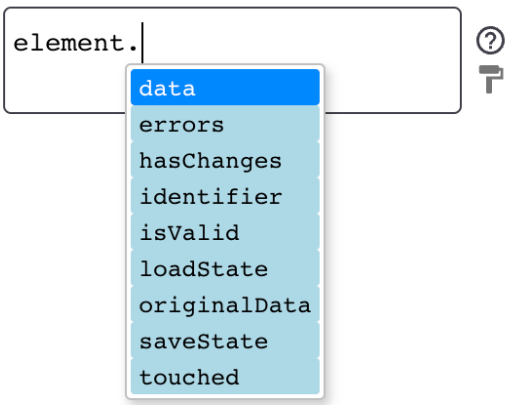


To get any value from the referenced row, just pick the desired column names.

Form

Let's focus our attention on forms which need special infrastructure to work properly. The following variables exist in this context.

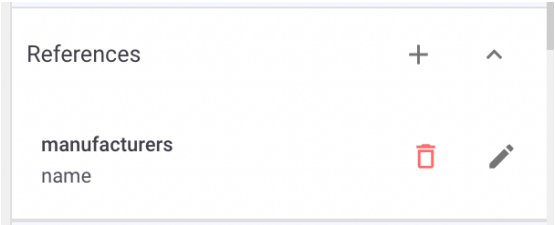
`element.` or `elements.<form_id>`



element.data

Server data, fetch by setting the proper “Query Name“.

Note: The Form data object also has access to values referenced, if the form has any joined (referenced) queries.



An example: To get the value of “`manufacturers`“, use the following syntax:

```
element.<form_id>.data.manufacturers
```

In this example, the manufacturer's column does not exist in a query that belongs to the form, but this value was joined by configuring “Form“ references.

element.errors

An object of possible server errors, available only if errors exist.

element.hasChanges

Is a boolean value. The value is "`true`" in case the form has been changed.

element.identifier

String or number required for identifying a record in the form with type "Edit" or "Detail".

element.isValid

Is a boolean value that describes if the form is valid or not.

element.loadState and elements.saveState

These are both objects which look as follows:

```
{
  inProgress: boolean;
  error: string | { message: string }
}
```

element.inProgress

Shows whether save or load action is in process. It contains "error" in case the request is failing or has failed. An error message is provided.

element.originalData

Initially fetched data. The original copy of the data is preserved until the form is submitted, so that you can always ensure that the changes can be reverted back to what was stored before.

element.touched

Boolean value shows if the user has touched any form input.

Conditional Container

`element.` or `elements.<conditional_container_id>`

element.visible

Boolean value. Since “Conditional Containers“ serve to display elements conditionally, depending on whether “visible” is set to “true” or “false”. In CYPEX the visibility of an element on a page can be turned on and off.

Note: To toggle an element’s visibility inside a “Conditional Container“ use the configuration value of other components such as “Boolean Input“ elements on the same page.

`elements.<boolean_input_id>.value`

Tabs

`element.` or `elements.<tabs_id>`

element.indexSelected

Returns a number (index) of the active tab.

Inputs

All inputs in CYPEX are accessible, like elements. or `elements.<input_id>`

element.value

Returns a value depending on the input type, e.g., “Number Input“ has an integer value, “Text Input“ a string, and so on.

element.disabled

A boolean value indicating whether the input is read-only or not.

element.touched

Shows if the user interacted with the current input.

element.errors

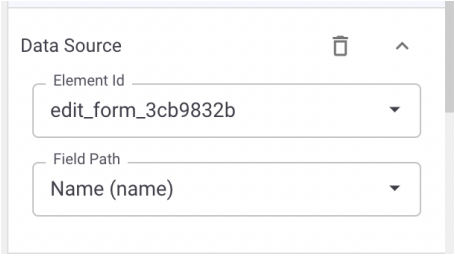
Optional key. Contains form data validation errors.

Note: Controlled Inputs

All inputs inside the form are controlled by the form they belong to.

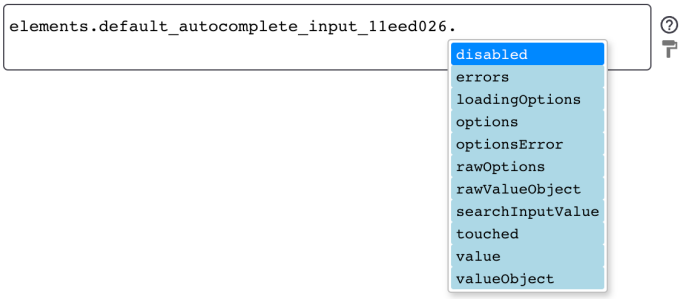
The “Data Source” section is the place to go to get the data from the parent form.

For example:



Where “Element Id“ is a parent form element ID. “Field Path“ wanted the column to be displayed.
So, the default value of the controlled input is “Form” data passed through the input props.

Autocomplete Input



element.loadingOptions

Is a boolean value that shows if option fetching is in process.

element.options

Is an array of options, each option is an object. Here's an example:

```
{
  value: string | number;
  label: string;
}
```

element.optionsError

Server error if fetching options fails.

element.rawOptions

An array. The row data fetched if “Options Source“ is a query.

element.rawValueObject

An object. The data containing the selected row.

element.searchInputValue

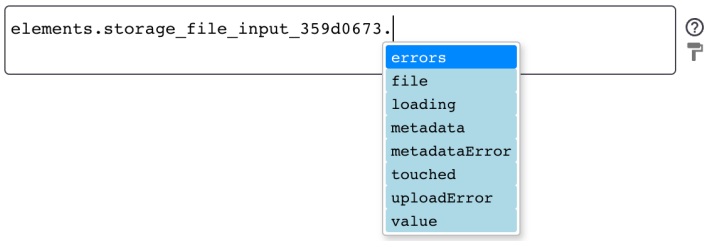
A string. The user input holding the value you are searching for..

element.valueObject

If value is selected, the value of the object is simliar to how it is in the the following example:

`{ value: string | number; label: string }` is accessible.

File Input & Multiple File Input



element.file

Access to uploaded files.

element.loading

Is a boolean value that shows if the file is currently uploading.

element.metadata

An object, uploaded file metadata

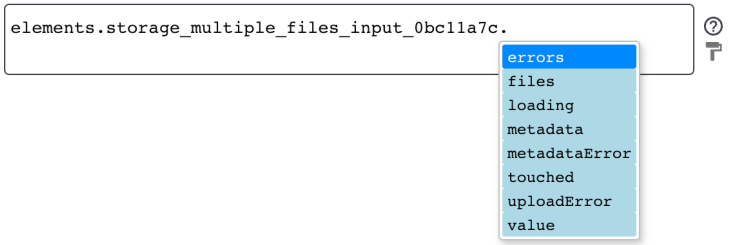
```
{
  "hash": string;
  "fileName": string;
  "realName": string;
  "fileType": string; // e.g., "image/png"
  "fileGroup": {
  "id": string;
  "name": string;
          // e.g., "public" | "private",
  "acl": string[];
          // array of strings (roles),
          // permissions for file group, e.g., ["cypex_admin"]
  },
  "typeGroup": {
    "id": string;
    "typeName": string; // e.g., "image"
  },
  "acl": string[]; // array of strings (roles), permissions for
file, e.g., ["cypex_admin"],
  "id": string;
}
```

element.metadataError

Is a string, error message.

element.uploadError

This value is a string error message, defined in case uploading fails.



element.files

Access to an array of uploaded files.

element.metadata

Same as for the single file, but an array of metadata objects.

Subform table

Subform tables serve mostly as form input, in the case of "References" are configured (at least one). So "Subform table" can update joined tables, used for 1:n relations during data editing. Subforms used as input have the same input properties mentioned above, but the value is an array of objects (joined table data). The syntax to get those values is as follows:
`elements.<sub_form_table_id>.value` returns `Array<object>`.

Fields

Various elements in CYPEX are accessible as shown in the following listing:

`element.` or `elements.<input_id>`

Google Maps

In this section we will discuss how custom expressions can help to make maps in CYPEX better.

element.data

An array of markers, markers are objects

```
{
  lat: number; // latitude
  lng: number; // longitude
  name: string;
}
```

element.loading

Is a boolean value indicating if data is loading or not.

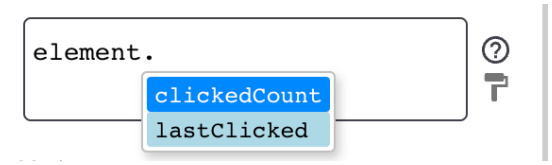
element.error

Error message in case data load fails.

element.selected

Returns marker (check the type above) object, only if any is selected by the user.

Action Button



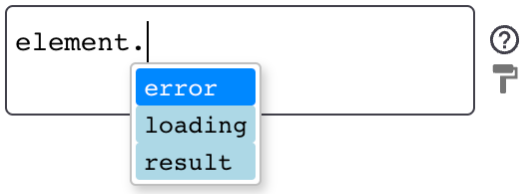
element.clickedCount

Number of times the button was clicked.

element.lastClicked

A date type, last time the button was clicked.

Call Button



element.error

An error message in case the function call fails.

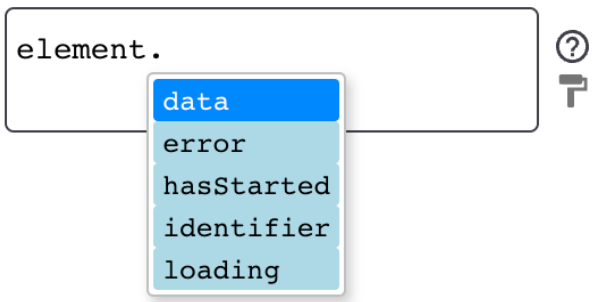
element.loading

A boolean value. Indicates if the function is being called right now.

element.result

The result of a function call.

Internal Link Field



element.data

In order to have access to data objects a “Data Source“ is required. In this case, data will be a record (table row). You have access to the data as well as to identifiers and status-related information.

element.error

Error message in case a request fails.

element.hasStarted

A boolean value, indicates if data has started loading.

element.identifier

String or number, in case the identifier was set in the “Data Source“ section.

element.loading

Boolean value, if loading currently in process.

Number Field

value: a number, field value.

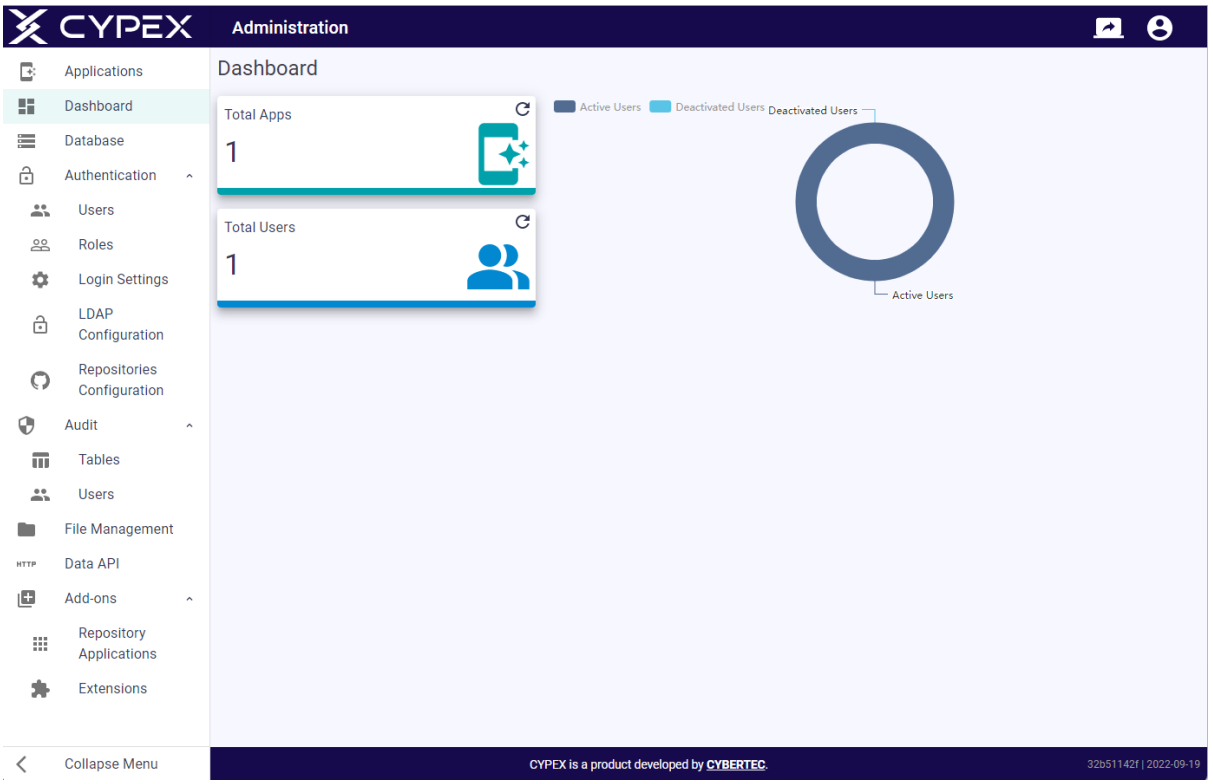
CYPEX administration panel

The CYPEX administration panel gives users an easy way to administer and manage CYPEX as a whole. Many functionalities such as ...

- The generation of applications
- Security management
- Data model definitions
- Workflow management
- Extension handling

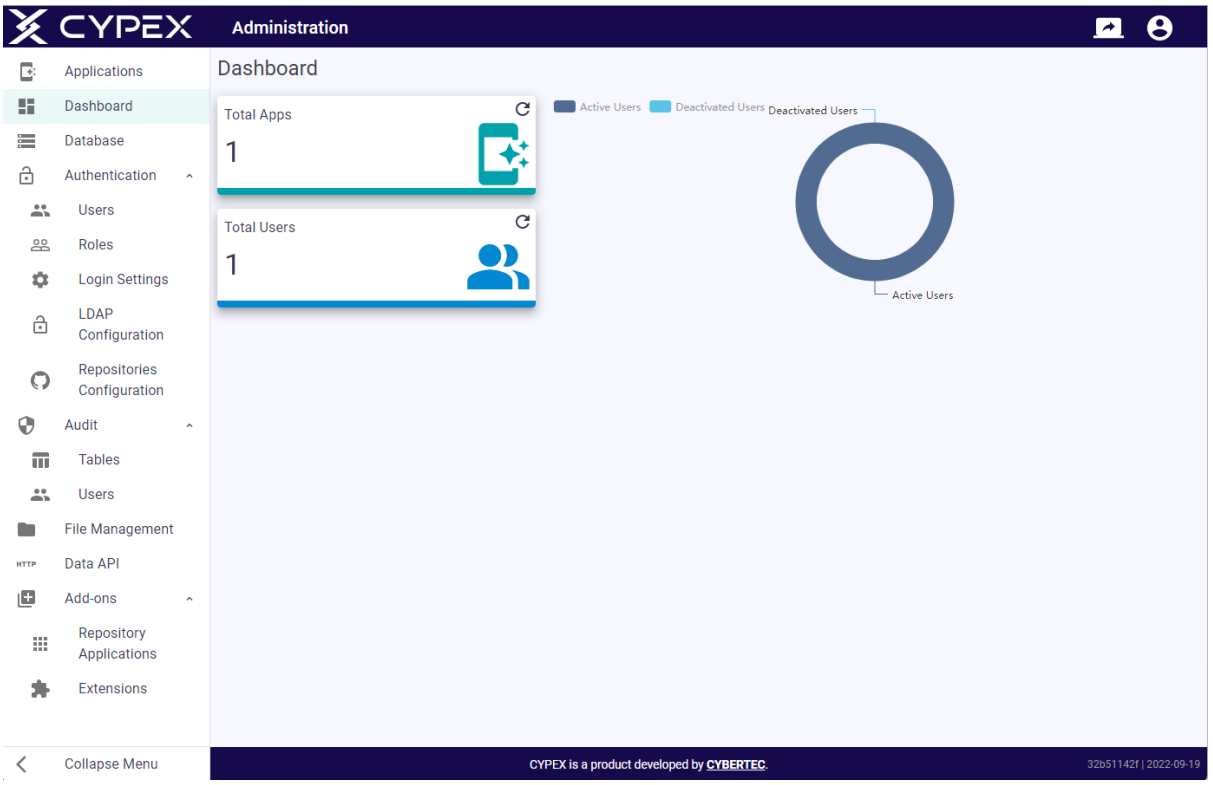
... and whole a lot more are all handled by this important and easy-to-use interface.
In this section you'll:

1) learn how to use the tool
2) understand how it works and
3) find out how to achieve your goals easily and efficiently.



CYPEX dashboard

Once you have logged into CYPEX, you'll find yourself on the dashboard. It gives you an overview of what's going on inside your CYPEX deployment:



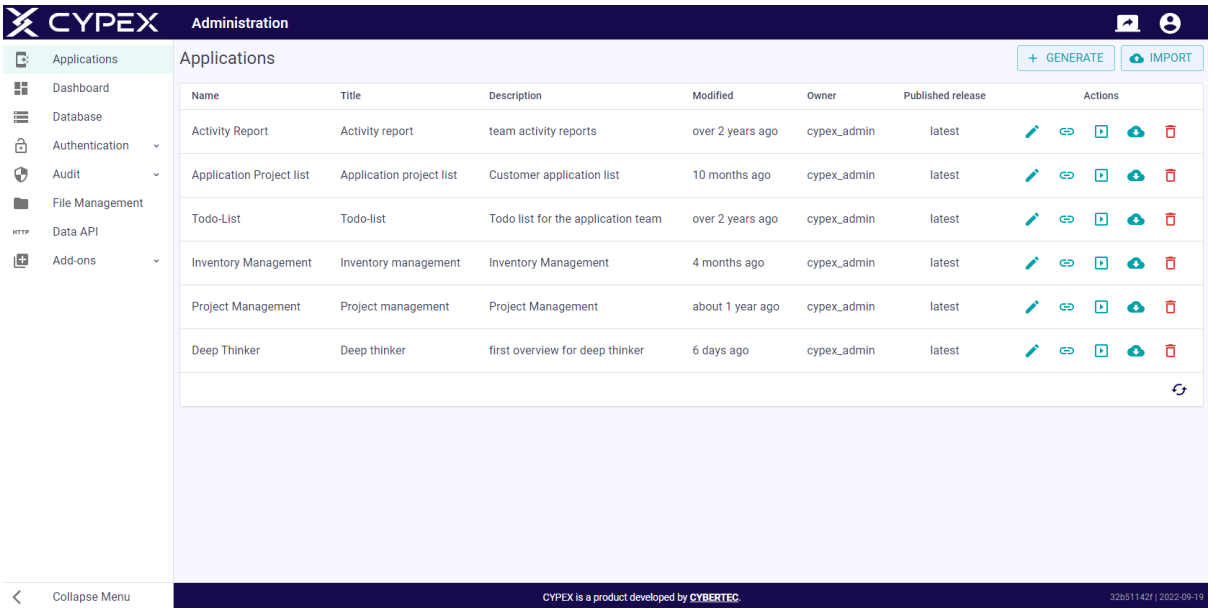
CYPEX Applications

A single CYPEX deployment can handle a large set of applications running inside the same database. If you click on “Applications” in the menu on the left hand side, CYPEX will present you with a list of all those apps currently deployed on your system.

You can easily manage your applications from this menu. This includes but isn't limited to:

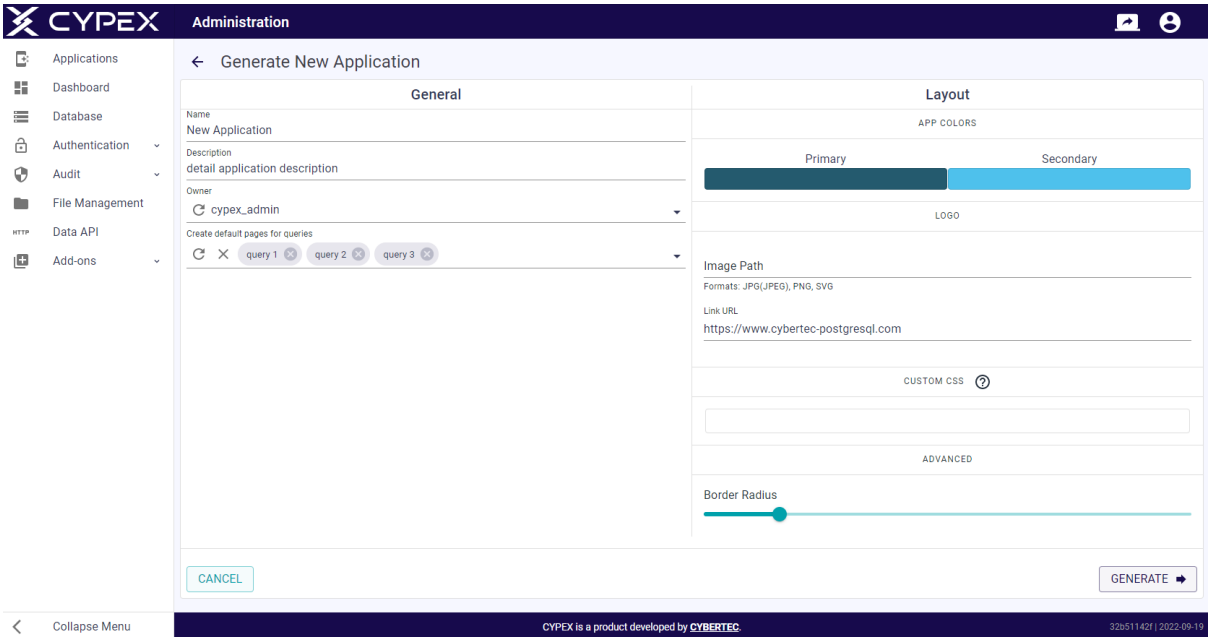
- Generating new applications
- Incremental rendering for existing applications
- Launching applications
- Deleting obsolete apps
- Importing entire applications from other systems

Let’s walk through this important page:



Create application

The most important moment in the life-cycle of a CYPEX application is its creation. CYPEX will predict the application out of the underlying data module. To start the process, click on “GENERATE”. This will open the following screen:



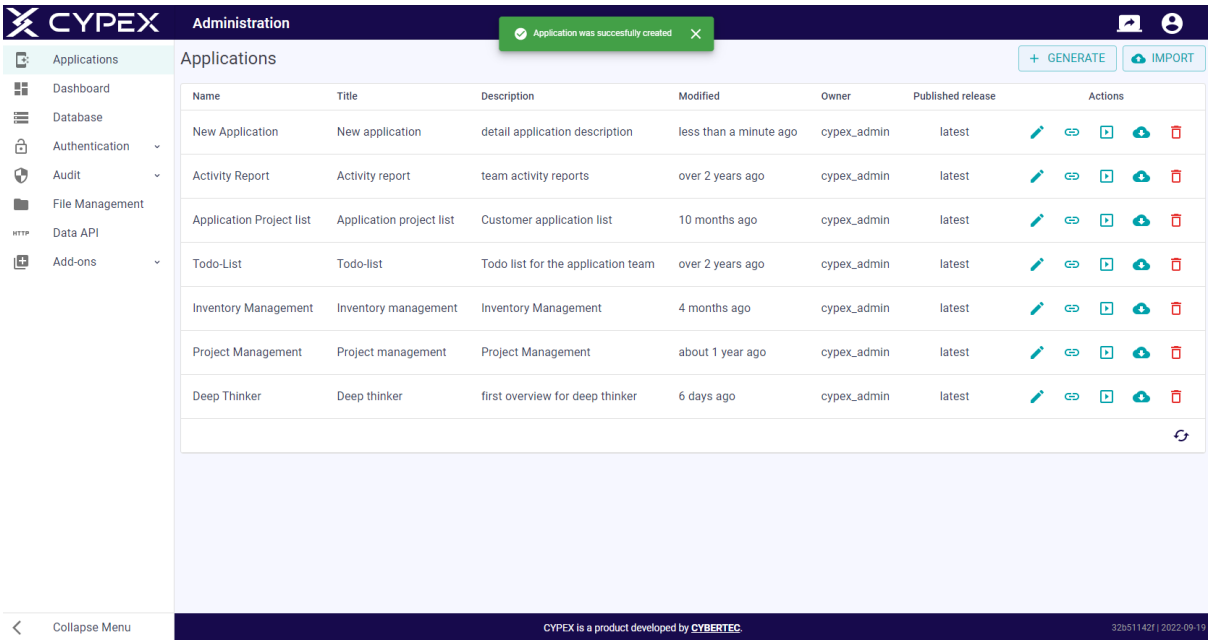
The first thing you have to define is the name as well as the description of the application. Then, you need to configure which user the app has to be predicted for. This is important because depending on who you are, you’ll end up with a different application. CYPEX will only render elements you have access to. If you aren't allowed to perform certain operations in your application (e.g. “sign contract”) CYPEX won’t generate tables, forms, buttons, etc. for that purpose. Therefore, selecting the right user is of vital importance. You also have to keep in mind that it’s often necessary to create many applications for the very same database. Just imagine a simple online shop: The backoffice application and the front app might operate on the same database, but those applications will be totally different because of permissions, requirements and so on.

Once you have decided on the user who will own the application, you need to select the “queries” you want to use in your application. Those queries that are selected will be used by the default rendering and app prediction code. Often, it’s necessary to render all existing objects. However, this is far from certain, which is why you have the option to selectively decide about what you want to render.

Finally, you can choose a layout, which is a vital part of the process: In large companies, the style of an app is of critical importance.

Application list

Once the new application has been generated, it will show up in the list as displayed below:

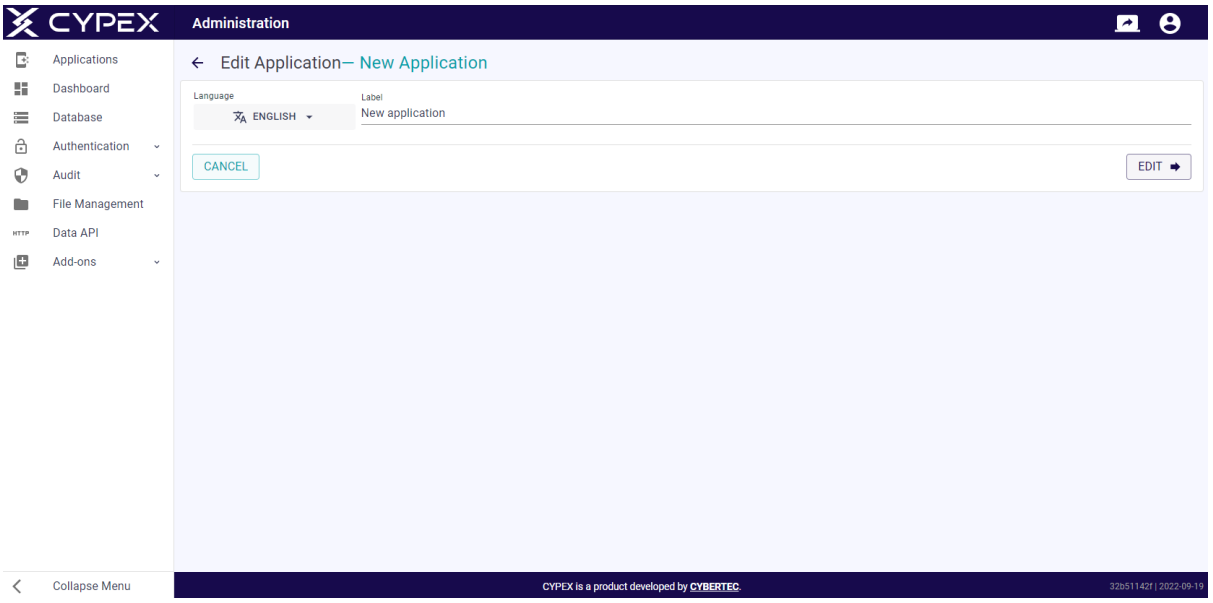


Application List icons

Next to each application in the list, you'll find a couple of icons which are needed to manage the application and handle its life-cycle. Let's walk through those icons step by step, and see how to make use of their functionality:

Edit icon

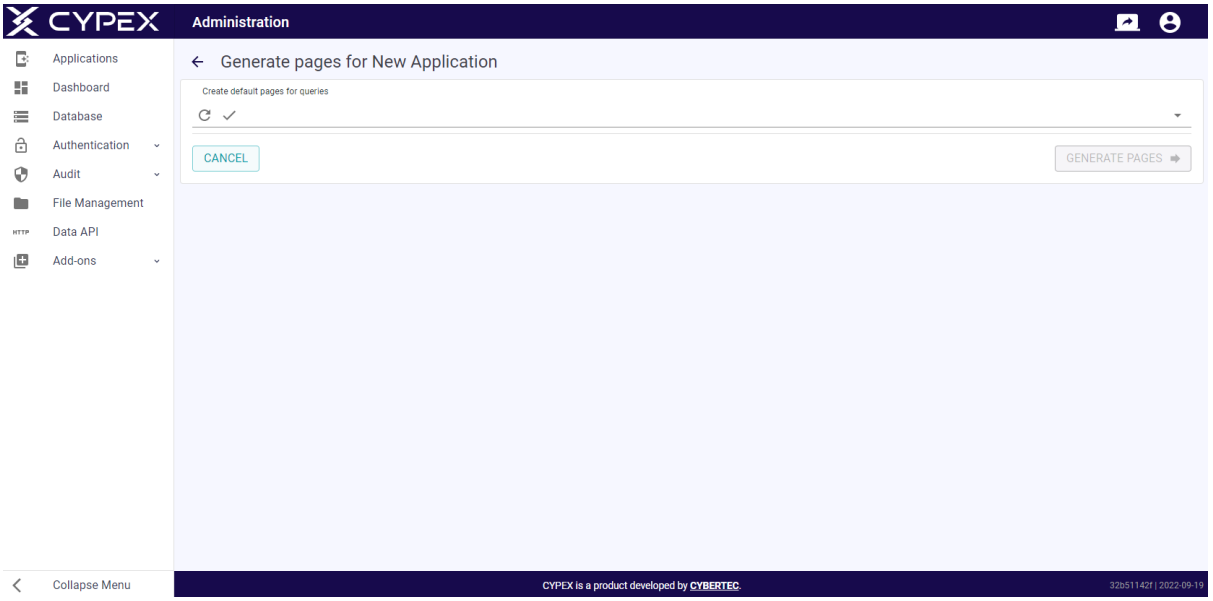
The name of your application can be easily added after its creation. You can also define the default language of the application:



Generate and add new page icon

Applications aren't static. During a project, the first incarnation of an application might not be the final version. This is true for the graphical user interface as well as for the underlying data structure. It happens more often than not that after an application has been used for a while, new tables are added and it's necessary to work incrementally on the application. The "generate and add new pages" icon can help you to predict pages and add them to your existing application. It's possible to use the icon after the app has been used, or after it has already been heavily modified. Incremental rendering is an important method to reduce the effort needed to add new components to your solution. Incremental rendering will dramatically improve the life-cycle of your application. No application is ever

static and therefore it's vital to have the capability to add data sources after an app has become productive:



Note that you can only render pages which aren't part of your application as yet. CYPEX will only provide you with those queries which have not been used, reducing the manual input needed to an absolute minimum.

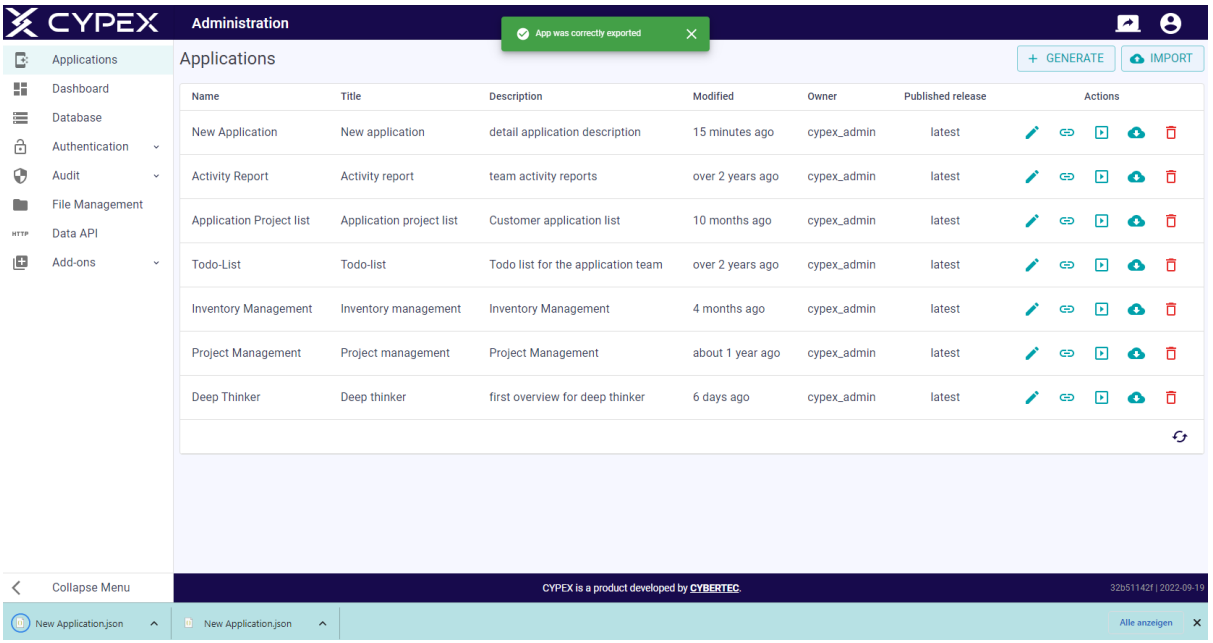
Start application icon

The “play button” will launch the application you’ve just created. You can use this button to ...

- Use the application
- Open the application to run the WYSIWYG editor

Export icon

CYPEX allows you to export an application. Why is this important? Development is generally not carried out on a production system. Therefore apps have to be transported from one CYPEX deployment to other systems. Import and export are needed to achieve exactly that.

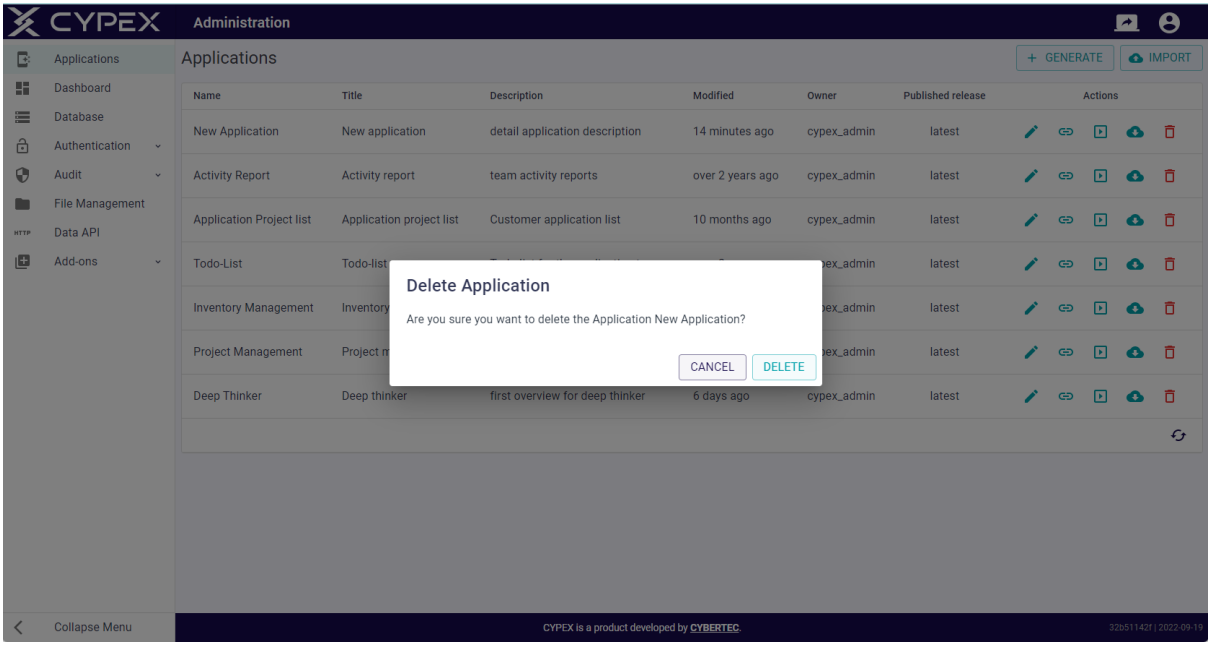


Export format in JSON

Note that the entire definition of the graphical user interface (GUI) is a single JSON document which can easily be stored, read and modified as needed. You can also put this under version control (for Git) to ensure that changes are tracked.

Delete icon

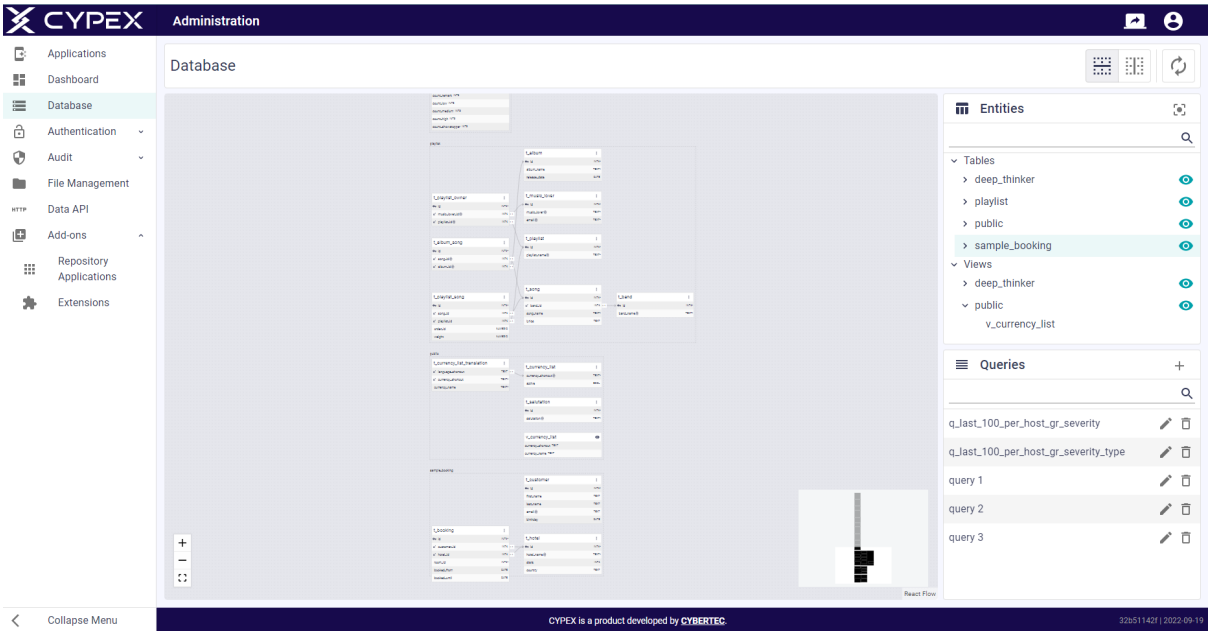
If you don't need your application anymore you can simply delete it:



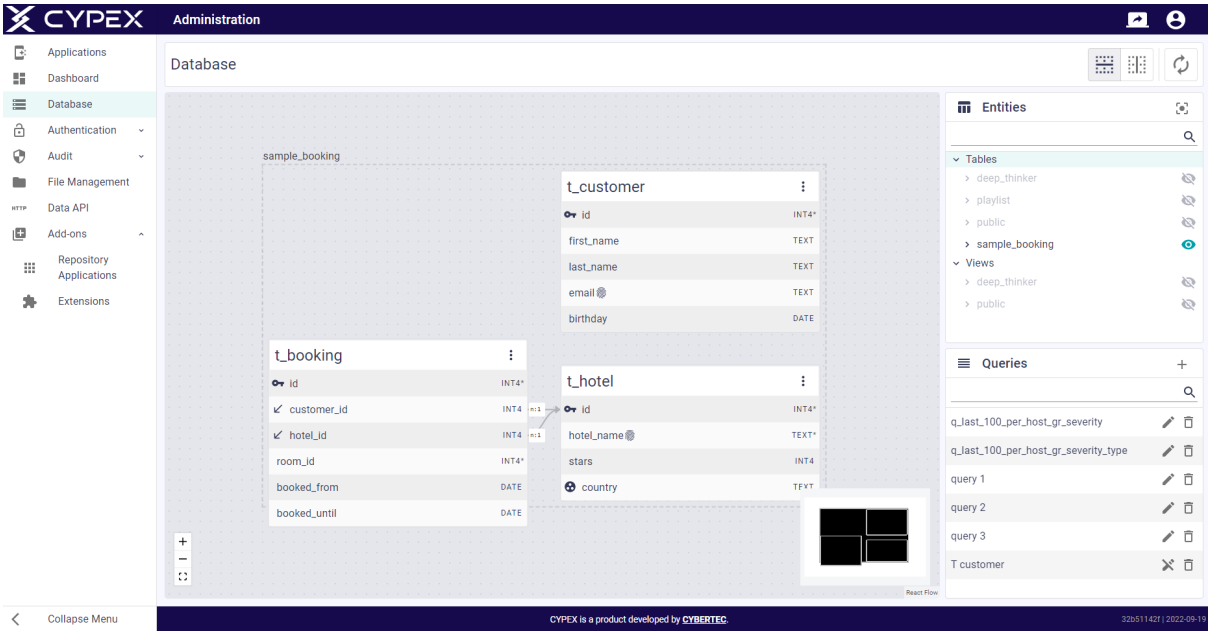
What this does is to remove the JSON documents from the database: Note that it does NOT delete queries, tables, constraints, workflows and alike - all we delete here is the JSON definition representing the graphical user interface. This is important to understand because CYPEX will not put your data at risk.

Database

The next important feature of the admin panel is the ER editor. It allows you to check your ER model, define queries, handle workflows and a lot more. It will be the backbone to handle the data side of your database infrastructure:

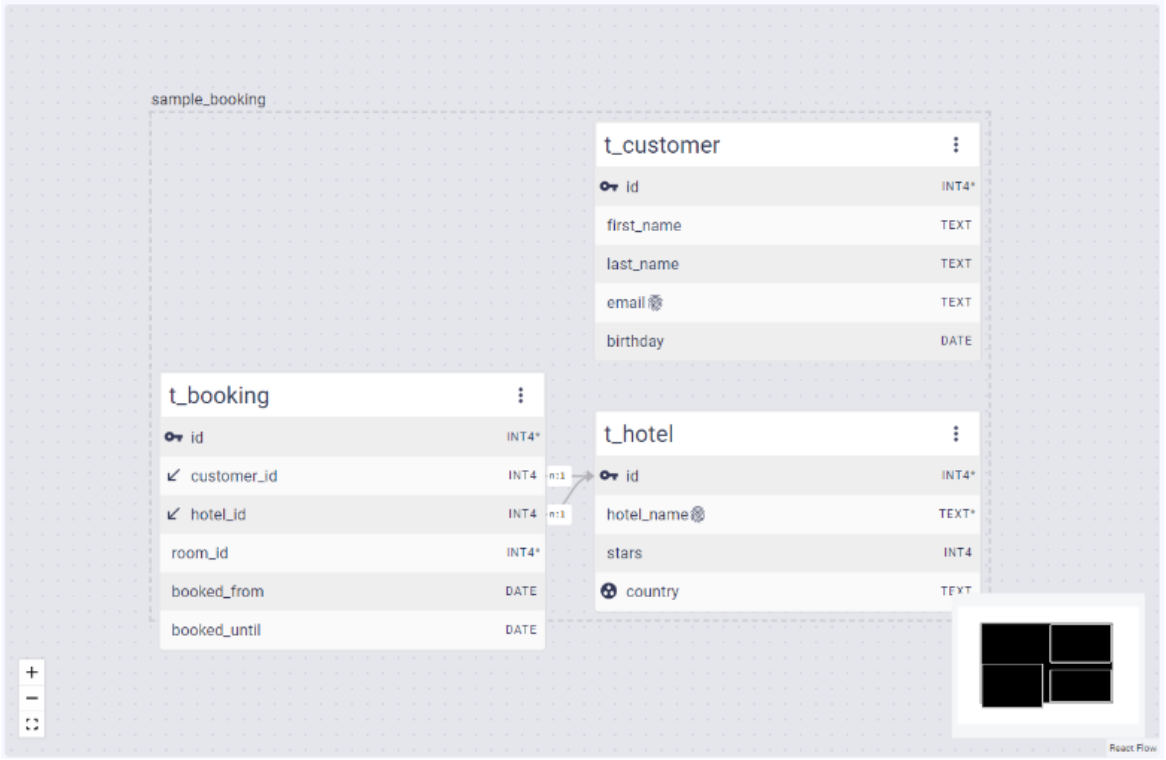


You can zoom in and out of your ER model easily. Tables residing within the same schema will be grouped together within a box. Also: You can click on tables and views on the right hand side to navigate through the ER model quickly and easily:



Schema overview

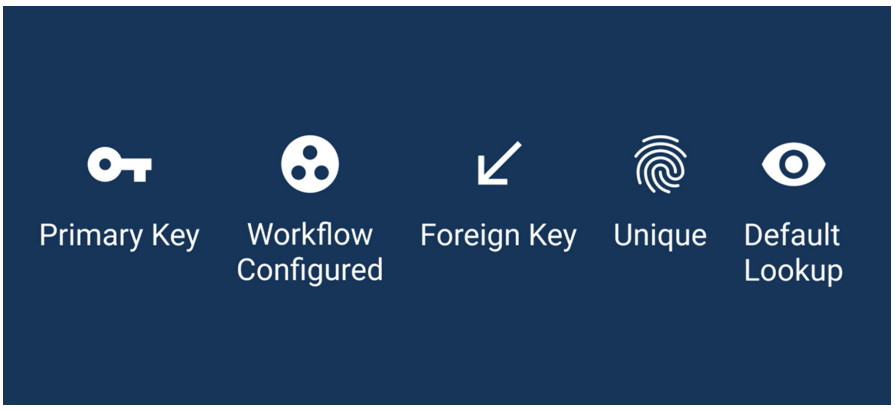
When looking at a schema there are a couple of things to consider: First of all each relation has “three dots” on the right side of the box. Click on those dots to configure the relation (create workflows, audit the table and so on). All those features will be explained later in this document.



You can also see that CYPEX displays the relations between those tables. But there is more. Let's take a look at those other icons in more detail:

Available table detail icons

The following symbols are used by the ER tool:



A primary key represents a unique-constraint which prevents NULL entries inside the table.

The next symbol indicates that a certain column is used by a CYPEX workflow. Remember, workflows are always defined on a column. The existence of the workflow is represented by the round symbol.

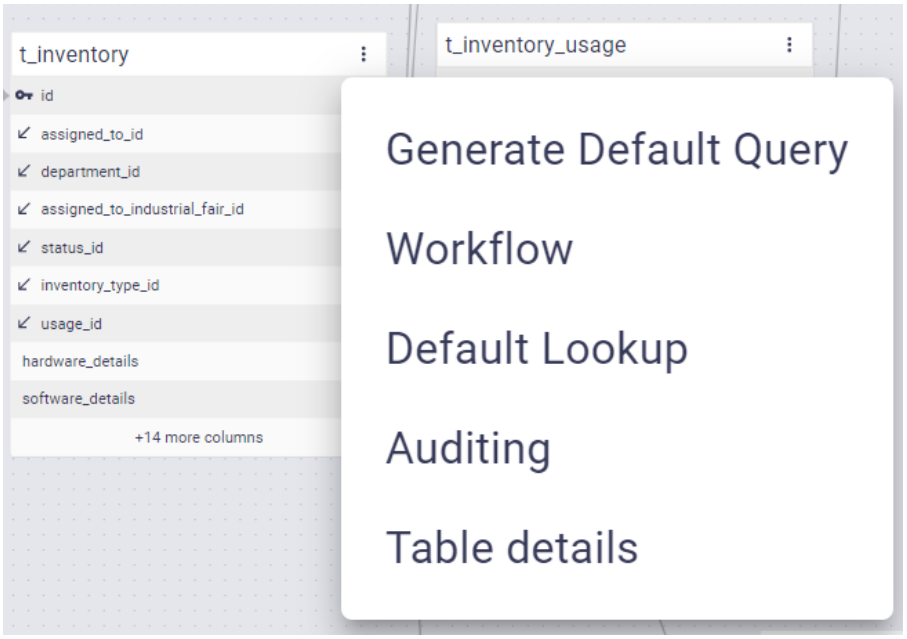
The “foreign key” symbol is on the “n” side of a “1:n” relationship. Please keep in mind that you should always index both sides of a foreign key relation to maintain efficiency.

The “finger print” column represents a unique field. In case a field is unique it can be used as an identification column.

Finally there is the “default lookup” symbol. It defines that the column in question will be used as the default text representation of the entity in question.

Context Menu Table

The core of every relational database is the concept of a table. If you have created a table, you'll be able to see it in your ER window. Here you see that you can configure various aspects of a table:



The following entries are available (in case you're dealing with a local table:

- Query generation
- Workflow management
- Default lookups
- Auditing
- Table details

Let's walk through those components one by one.

Generate Default Query

The first thing to focus on is the idea of a default query: Often you simply want to see and edit a table. A default query is the best way to make that happen quickly. Fill out the form and assign the following permissions:

Generate Default Query

inventory.t_inventory

Title
Inventory

Name
inventory

Permission Table

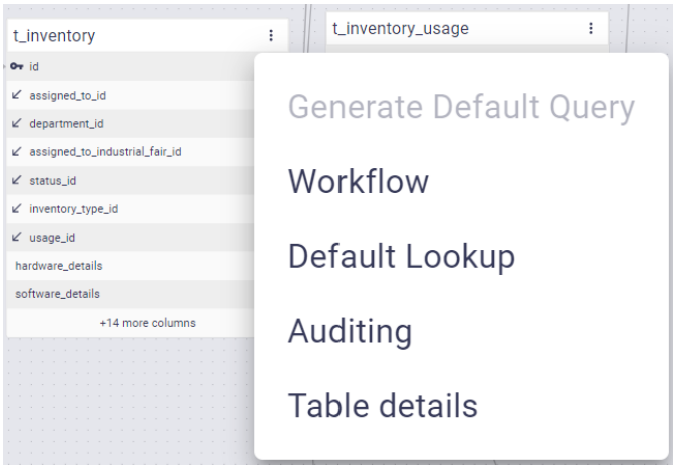
Role Search

User	SELECT	INSERT	UPDATE	DELETE
cypex_admin	☑	☑	☑	☑
cypex_user	☑	☑	☑	☐
inventory_owner	☑	☑	☑	☑
inventory_user	☑	☑	☑	☐
inventory_admin	☑	☑	☑	☑

CLOSE SAVE

The list of permissions is important because it will provide vital information used during default rendering. If a user isn't allowed to perform certain actions, the rendering process knows that those elements should not be generated in the first place. Without “INSERT” permissions, you won’t see input forms.

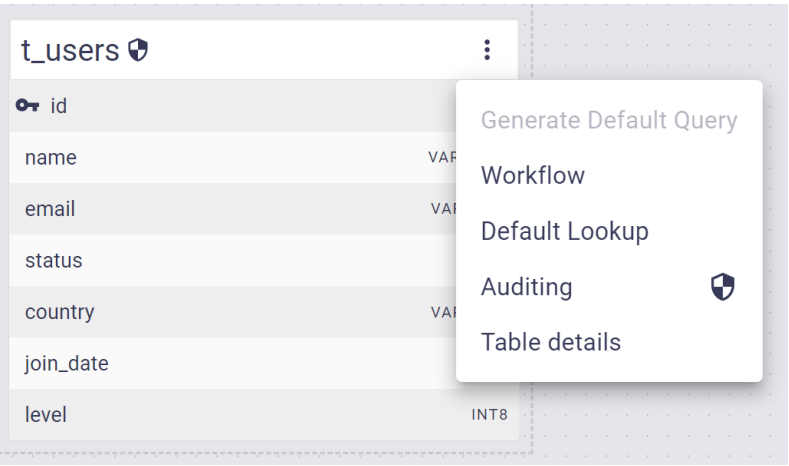
In CYPEX, you can have exactly one default query per relation. In case one exists already, the element will be disabled:



Understanding CYPEX workflows

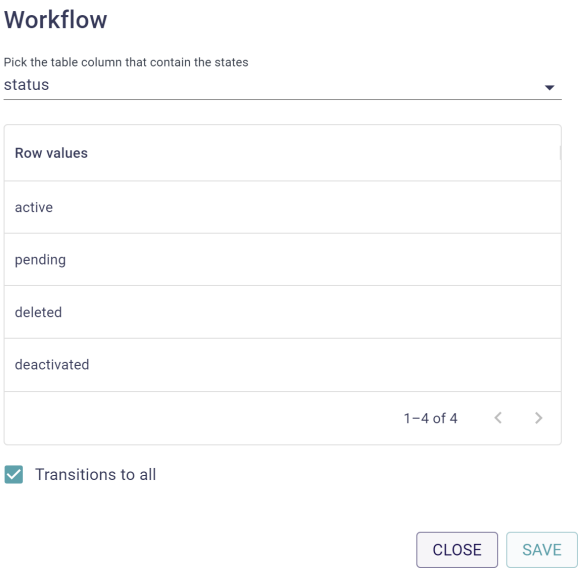
The next important menu entry is used to open the workflow editor. As previously stated, workflows are a core component of CYPEX. In a simplified world, tables translate to forms and tables - workflows are in charge of buttons, etc. One could argue that workflows actually add “life” to your otherwise pretty static application.

In CYPEX, workflows are associated with tables and are defined for a state column. Go to the ER model and click on the three dots. There you'll find a menu entry allowing you to define a workflow:



Creating a new workflow

To create a new workflow you need to choose the column which will contain the status information of the workflow. How does it work? The first important thing to understand is that a workflow basically (but not only) consists of states as well as state changes. States are valid entries inside the state column, while state changes are UPDATEs moving states from one value to another. The validity of those changes are guaranteed by the database engine.

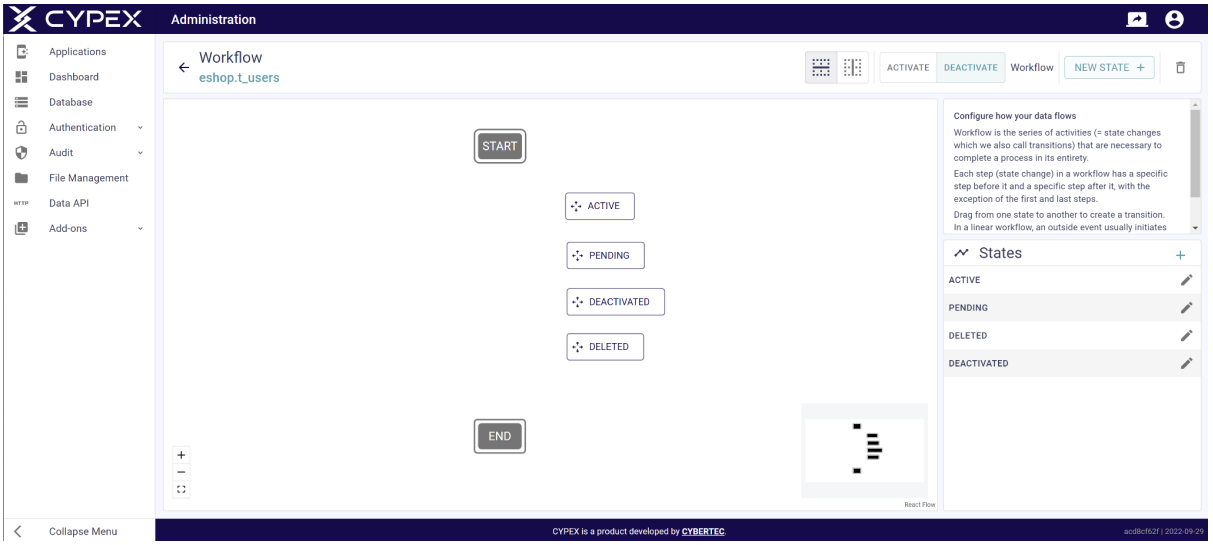


First select the column which is going to be used to store states. CYPEX will provide you with values that are currently in the table and offer them as valid state entries.

Using the workflow editor

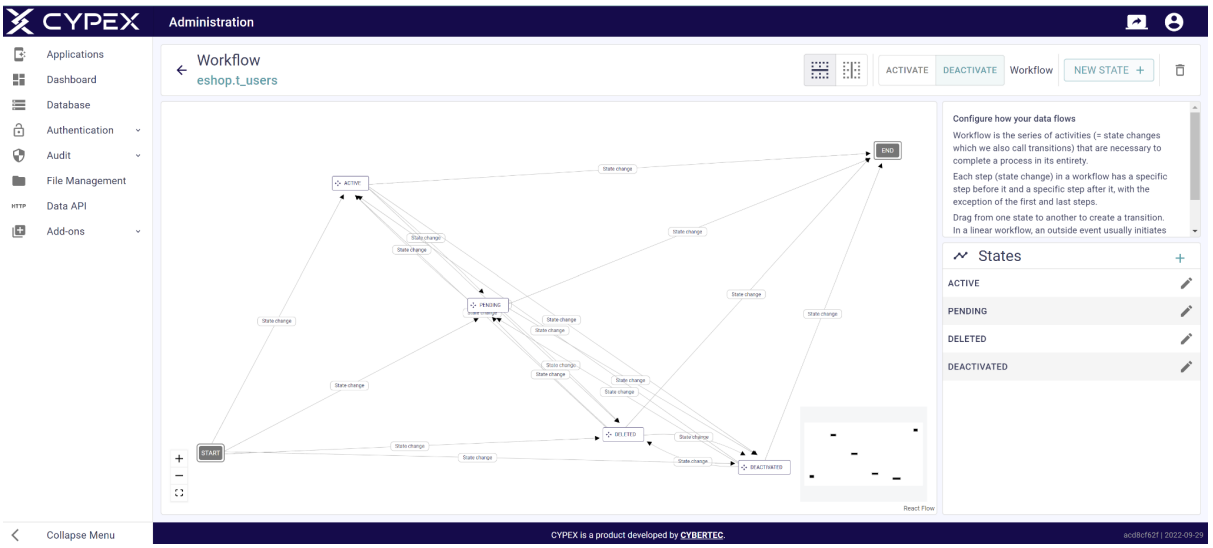
Once you have selected the state field you can create the workflow without any additional precautions.

Let's take a look and see what the workflow looks like. Note that if you didn't click the "transitions to all" you'll see the following picture:



States are listed but aren't connected yet. You need to do that by hand.

However, you might want all states to be connected with each other. In that case you click the "transitions to all" checkbox. CYPEX will then automatically produce state changes for you.



As you can see, the flow chart is more complicated in this case, since all those state changes will be represented using directional arrows. The arrows can be modified easily.

verview of the workflow editor action items

Let's get an overview of all action items available in the editor:

The headline section shows the name of the relation the workflow is being defined for.

The alignment icons are arranged as workflow elements horizontally or vertically on the playground

"ACTIVATE" & "DEACTIVATE" are of key importance: Suppose you are changing a workflow as part of a design process. As long as a workflow is deactivated, you make all the changes to the CYPEX configuration tables (meta data) - the workflow isn't enforced on the core relations. Only by pressing "ACTIVATE" will you enable PostgreSQL to apply CHECK constraints and triggers to actually ensure that what you see is actually in the data table. Keep in mind that activating the workflow can fail, in case invalid entries are included in the table. In that case, the data has to be cleaned first before the workflow can be enforced.

We have two ways to add new states to the workflow: The "NEW STATE" button and the plus icon + shown in the state overview section.

The delete icon allows us to delete a state from the workflow.

Symbolizes the start of a workflow (preliminary state). This element isn't a real state but shows the point when the non-existing element comes into existence

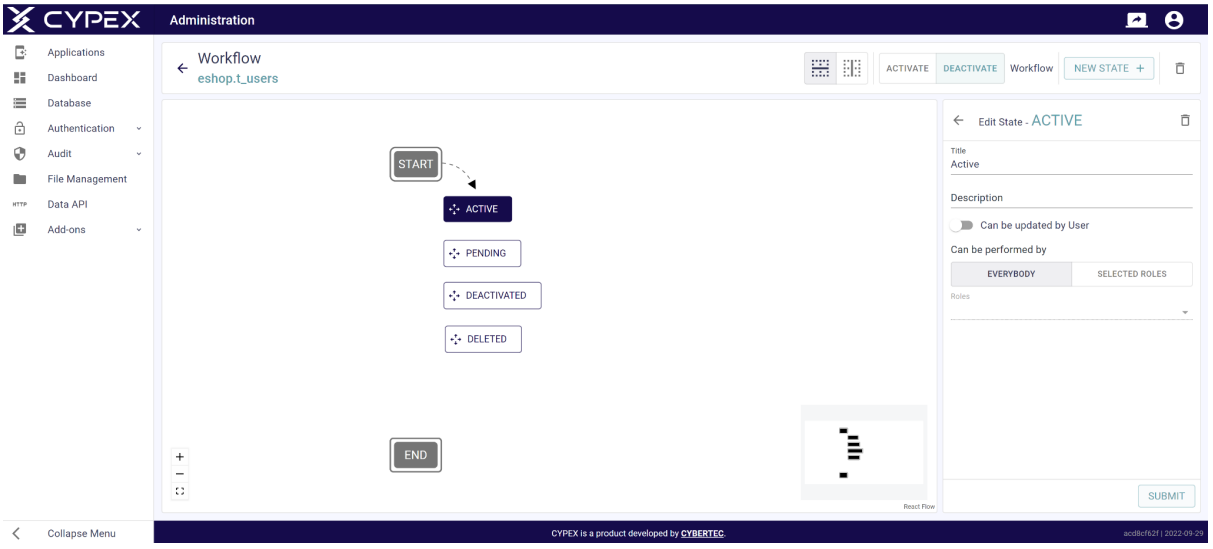
Shows the end of the workflow
(basically when the object is deleted)

Workflow states

Edit the state inside a workflow

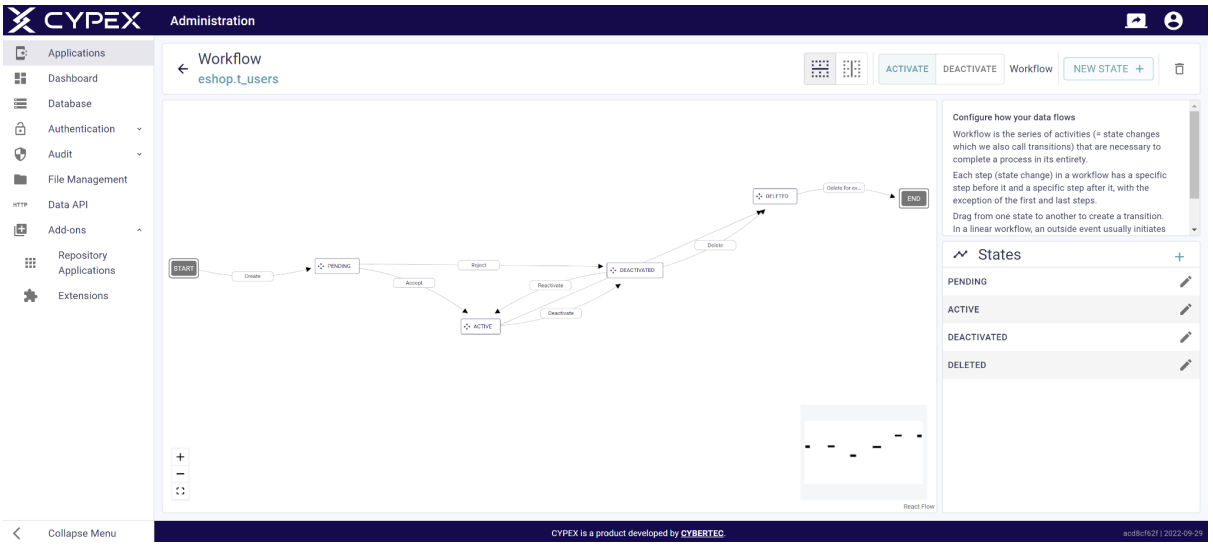
State changes

To create state changes, you have to draw an arrow from one state to your desired target state. Make your changes visually, as shown in the image below:



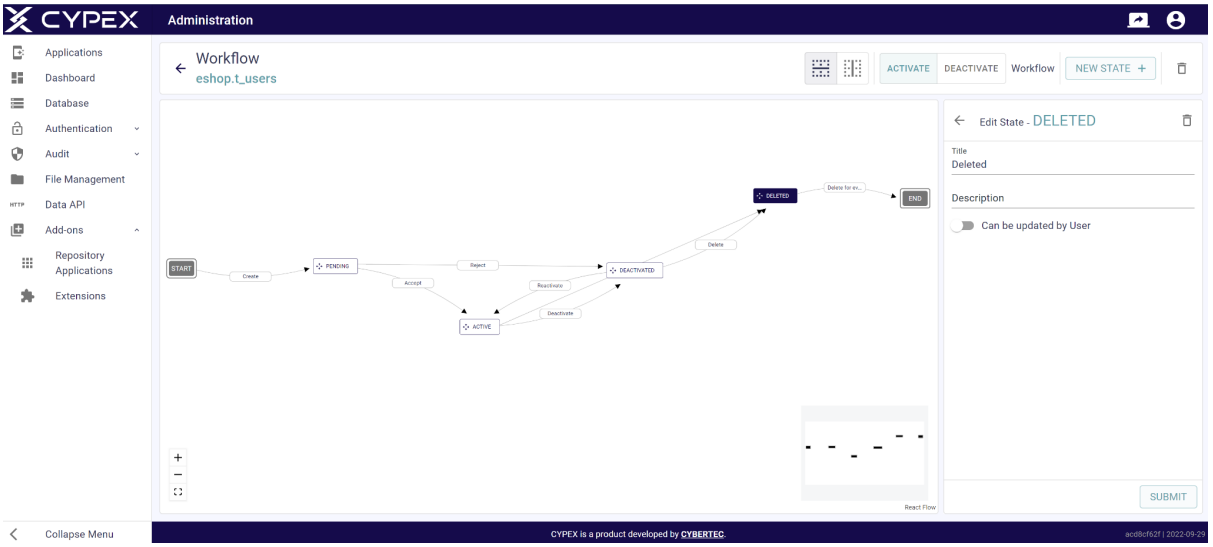
Make sure that your state changes are documented and configured properly. By adding texts to your state, the system will know how to label your buttons and so on. Therefore it makes a lot of sense to put effort into properly describing your workflow. Of course changes can be made later. However, it's good practice to use proper texts early in the process.

Once the work has been done it will look something like this: As you can see the business process as been properly modeled and can already be feedbacked by the end customer:



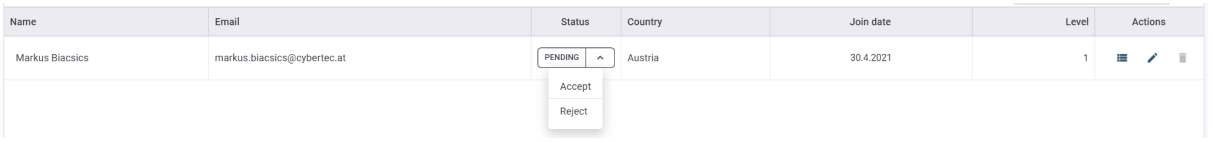
Edit a State

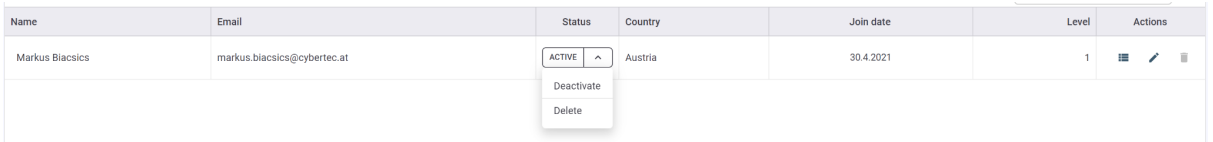
Often work has to be modified later. In such cases, use the “Edit state” machinery. It allows you to change texts later. Note that while it's easy to change the texts, PostgreSQL enforces these workflows and therefore making changes on excessively long tables can be quite time-consuming (of course changing the text itself does not matter):

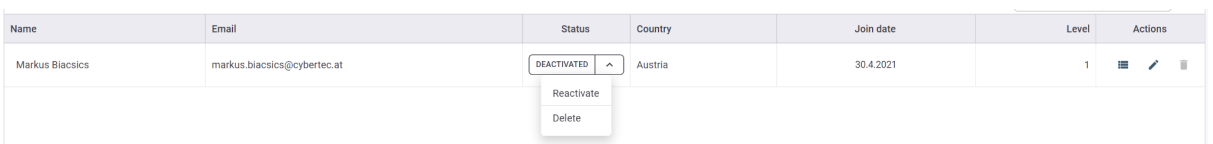


Inside the application

Workflows serve a purpose. They are important to ensure that the application actually does what it's supposed to do by limiting possible changes of values along the way. In the listing below, “pending” can result in “accept” or “reject”. Active can only be deactivated and deleted:





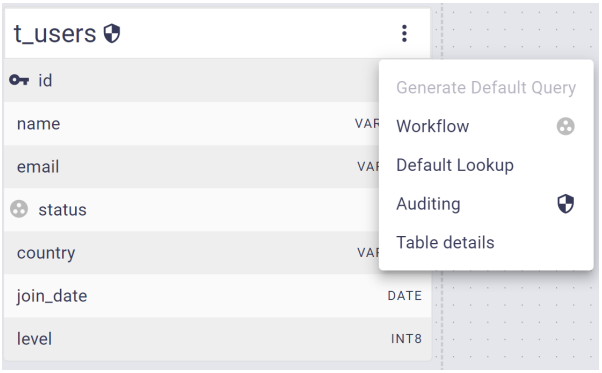


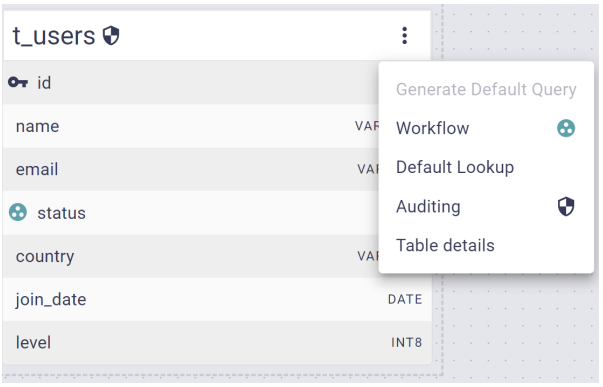
As you can see the GUI does not allow for changes that aren't supported by the workflow. The way data is changed is therefore rigorously restricted to what may happen.

Workflow symbols inside the table context menu

The existence of a workflow for a certain relation can easily be determined by looking at the symbols associated with the relation. The green symbol will show us the way:

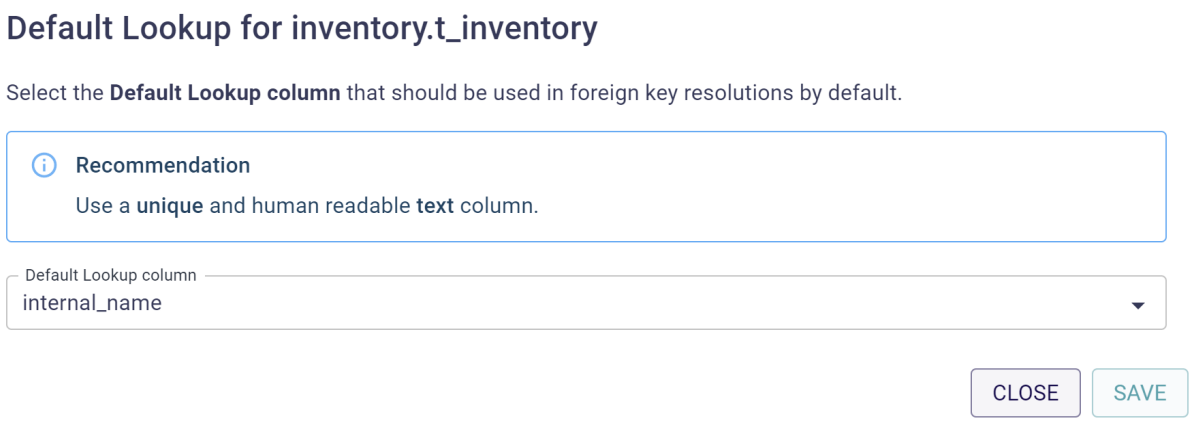
- Existing inactive workflow on column “status”
- Existing active workflow on column “status”





Default Lookup

Every relational model will contain a significant amount of “id” columns. The trouble is, nobody wants to see those numbers in the GUI. To ameliorate the situation, we enriched the data model by introducing the concept of “default lookups”:

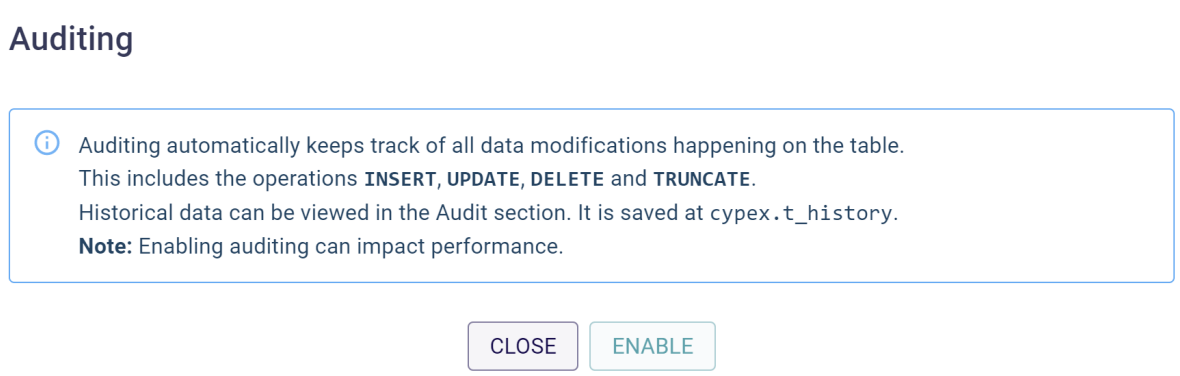


CYPEX relies heavily on foreign keys and constraints. Select the column you want to see instead of ID’s. CYPEX will inspect your ER model and key relations so that the default renderer can automatically generate the app the way you want things to be.

Auditing

The CYPEX development team has put a heavy emphasis on security as well as tracking. Our database experience tells us that security is a key concern for most enterprise customers. To reflect this need, CYPEX offers an easy way to audit tables and track all changes.

Enable the audit trail:



CYPEX will automatically deploy all the infrastructure to track changes made to your table. Those changes aren't only tracked when the GUI or the GUI is used - even changes made to the underlying tables directly will be tracked to guarantee that no changes are lost.

Turning off this kind of tracking is equally easy:

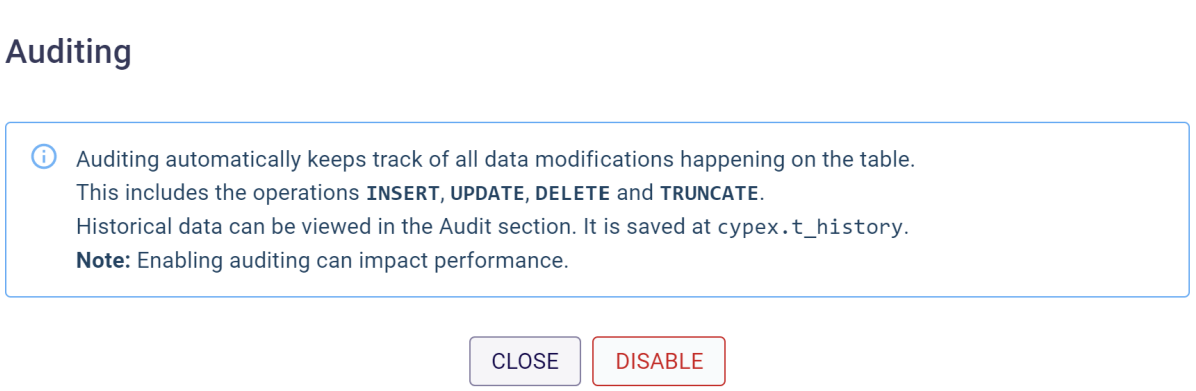
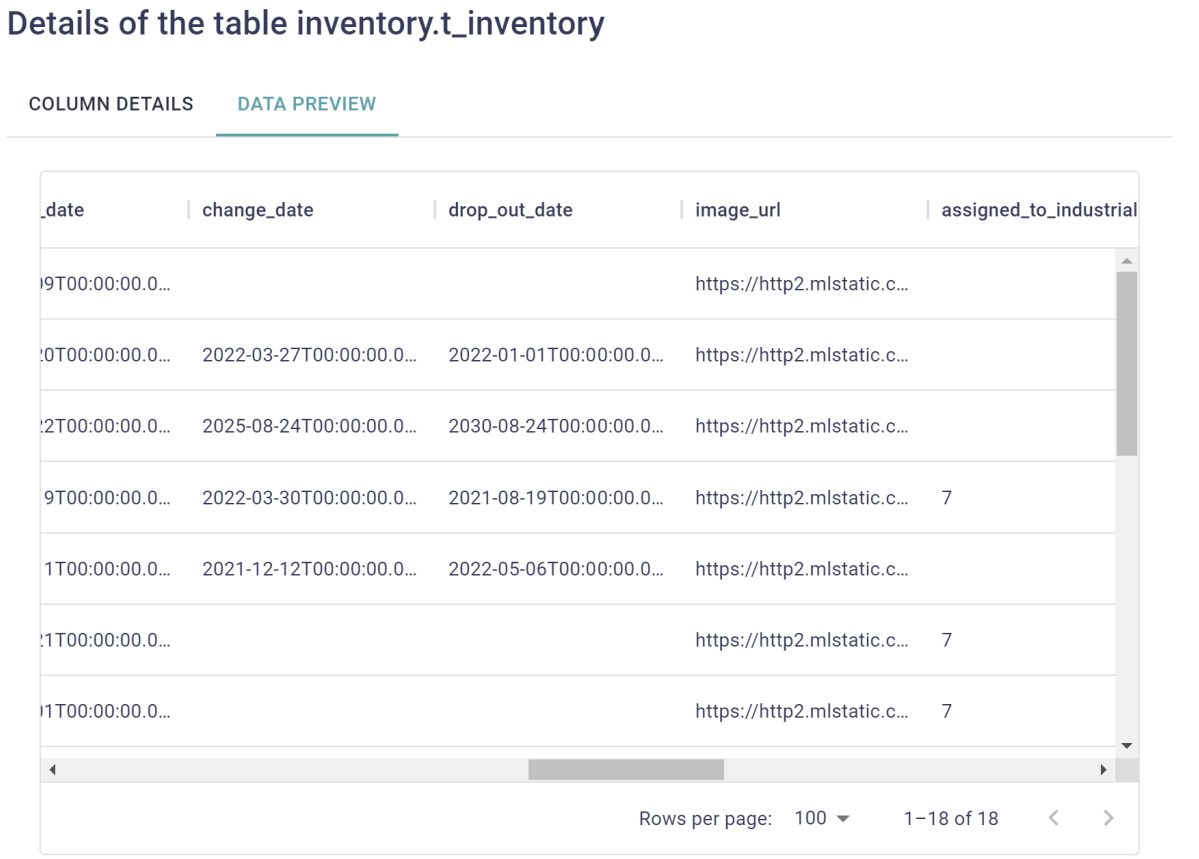


Table details

Just looking at an ER model might not provide you with all the information you need. The “table details” features allows you to take a look at your data and inspect the relation in more detail:

In addition to the data model, you can also take a look at the data inside your table. Note that only a subset of data is displayed, to ensure good performance. The CYPEX model builder isn't a replacement for a normal database client. If you want to search the table, modify it, etc., a standard database client such as DBeaver should be utilized.



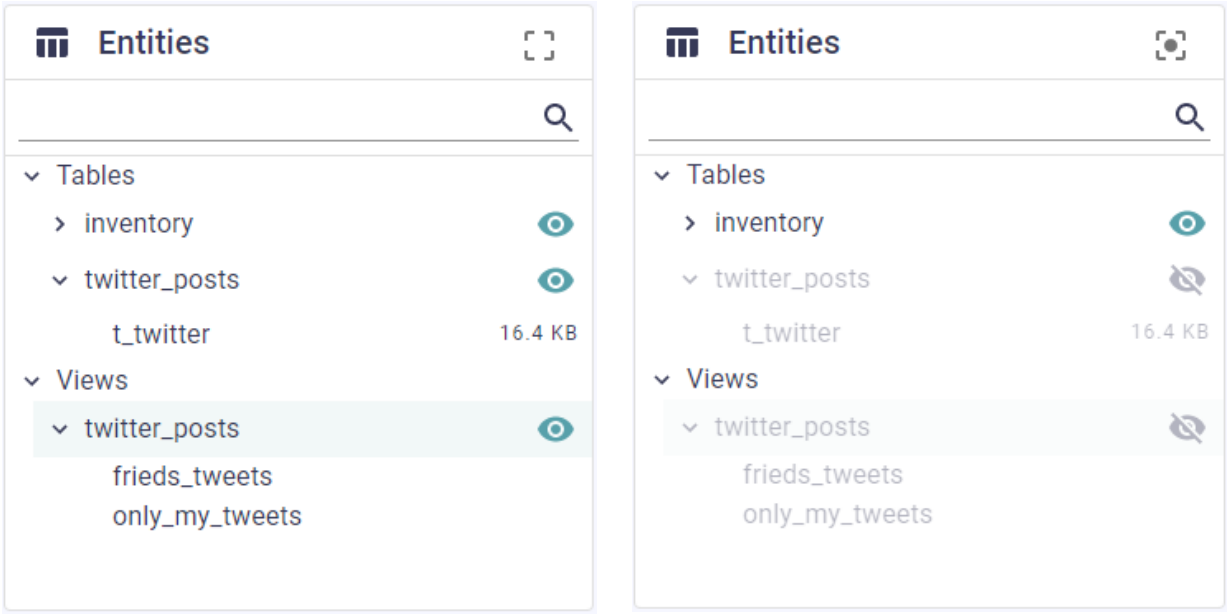
The “data preview” section allows you to inspect the data and get a feeling of what is inside the database. Often this is needed to get a handle on things before writing a query.

Entities

Now we'll focus on the entity section and discuss which icons are available and what those icons can be used for:

- Focus on selected table in schema overview
- Do not focus on the selected table in schema overview
- Filter table overview
- Schema is visible in schema overview
- Schema isn't visible in schema overview

If we don't want the twitter_posts schema to be visible inside the schema overview, click on the toggle icon on the right side. The schema with the tables and views will disappear from the schema overview. Both tables and views will always appear and disappear when the toggle icon is used.



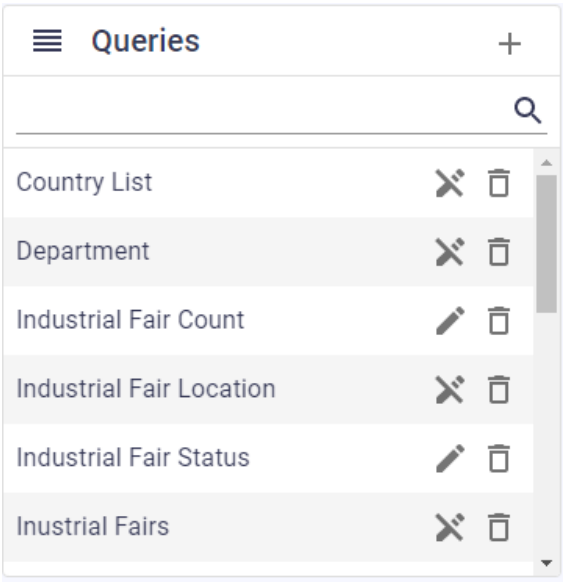
..

Queries

The next important section on the screen is related to queries. Raw tables are often what we need to build applications. In most cases we have to pre-process data before we can feed it to the graphical user interface. This is done using queries. You will find a handful of icons within the editor which can be used to control this feature:

- Add new query
- Filter queries
- Default queries are not editable
- Edit custom query
- Delete query

The following image contains an example showing what those menu entries might look like:

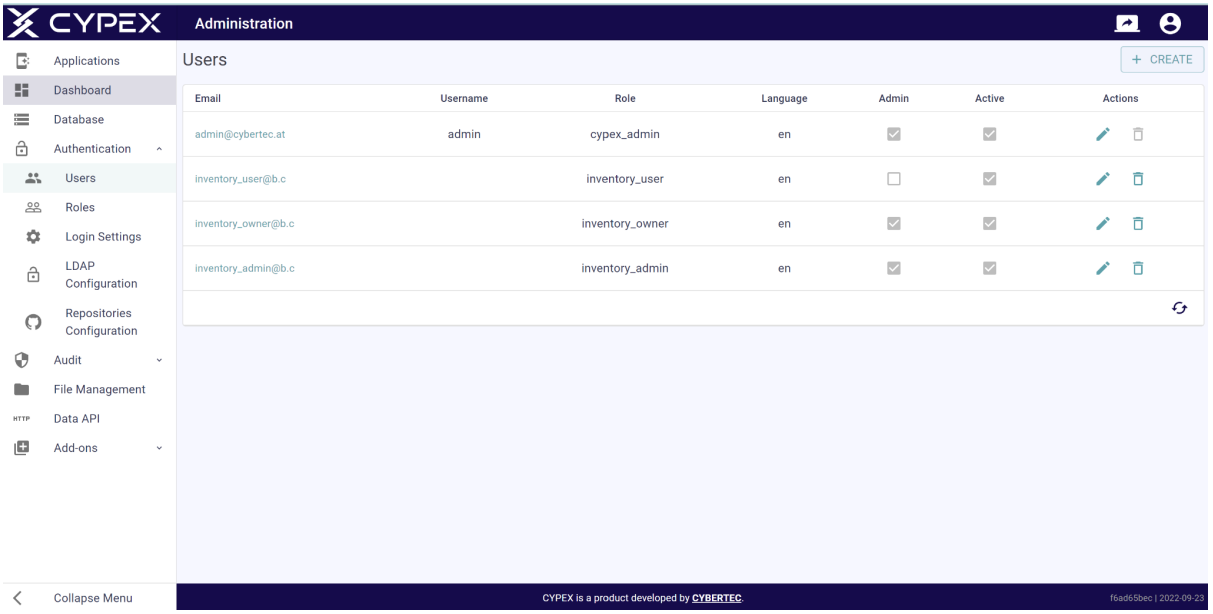


Authentication

Let's move on to a crucial topic: authentication. Security is of the utmost importance, which is why the CYPEX team has put great emphasis on protecting your data and applications.

Users

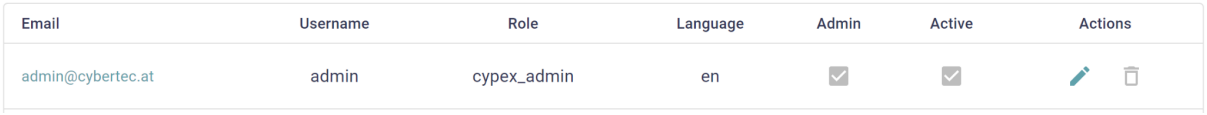
On the CYPEX side users and roles are mapped to “login names”. This can be done in the “Users” section of the admin panel:



It’s important to mention that **if a user is marked as “admin” it's possible to use the application designer (WYSIWYG editor) to modify applications**. In production this isn't desirable and therefore **you need to be careful with this setting**.

A default language can be assigned to a user. Usually this language is English, but almost every language is possible, assuming that translations are provided. Also note that users are mapped to a database role. This is important as the database role is what controls access to data. The login name (= email) is merely to handle CYPEX logins - permissions to data are managed on the lowest possible level (= PostgreSQL) to ensure consistency between the API, the app and of course with direct database access.

Users are listed in a table. What you see below is the email address you can use to log in. Then you see the underlying username as well as the PostgreSQL role assigned to this specific CYPEX user. Finally, you see the default language of a user and can figure out if the user is active or marked as an admin user. Users can be edited using the "pen" symbol:



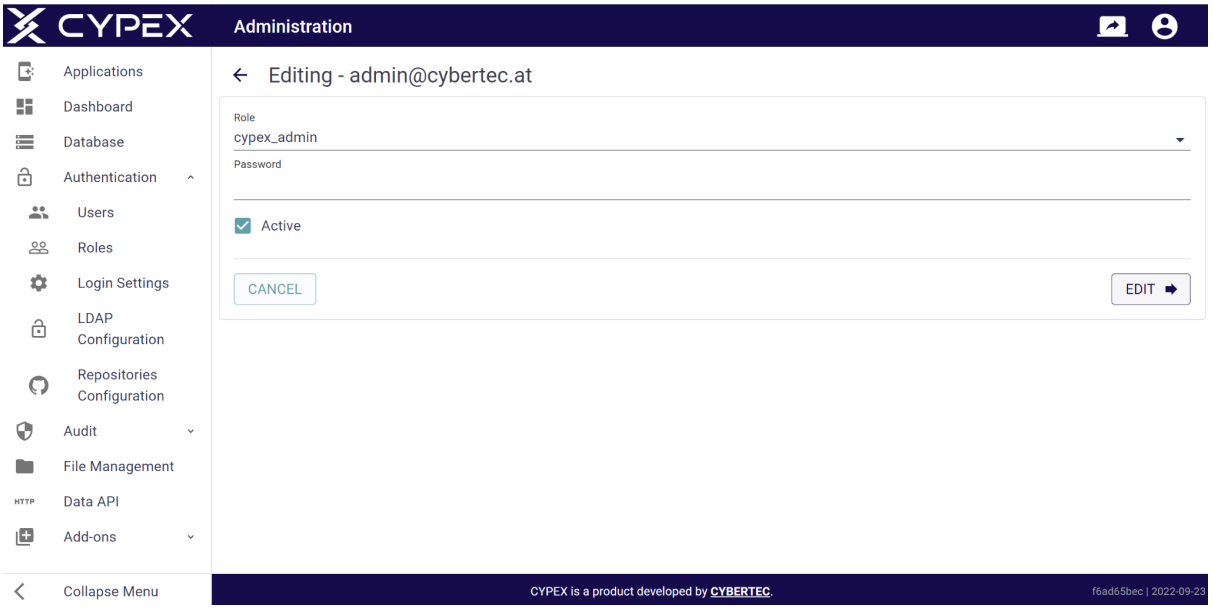
Create user

Users can easily be created in the GUI. Make sure that your user is properly mapped to the database role of your choice:

Users are available instantly - there is no synchronization of any sort needed to make this work.

Edit User

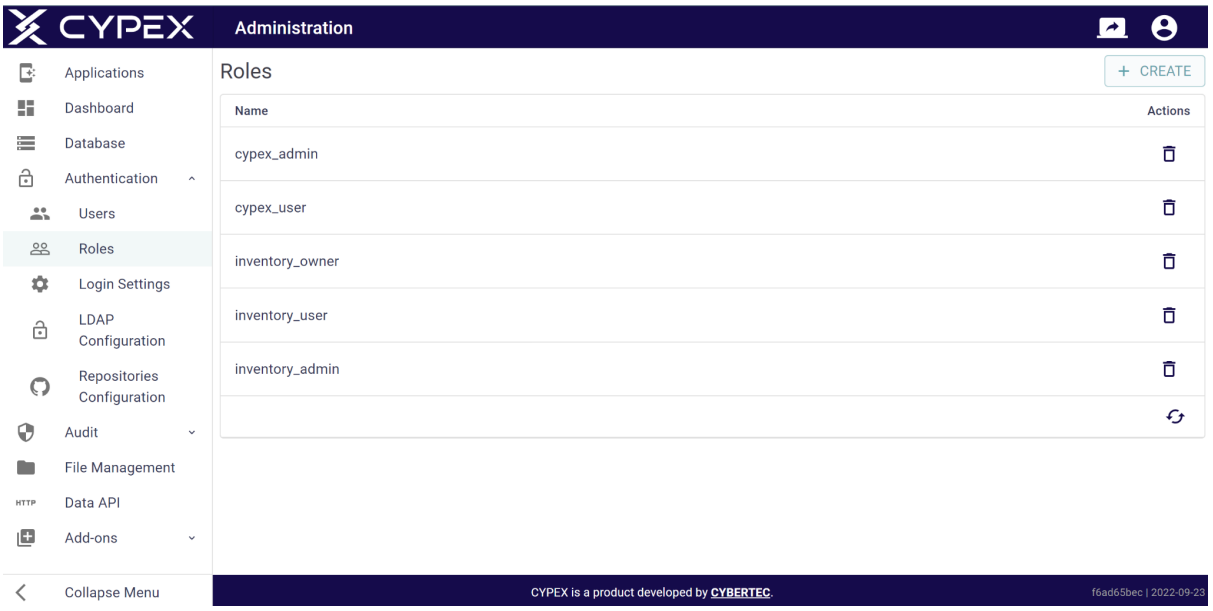
Editing an existing user is equally simple. Click the edit icon and make the changes:



You can change the password and quickly set the user as active / inactive.

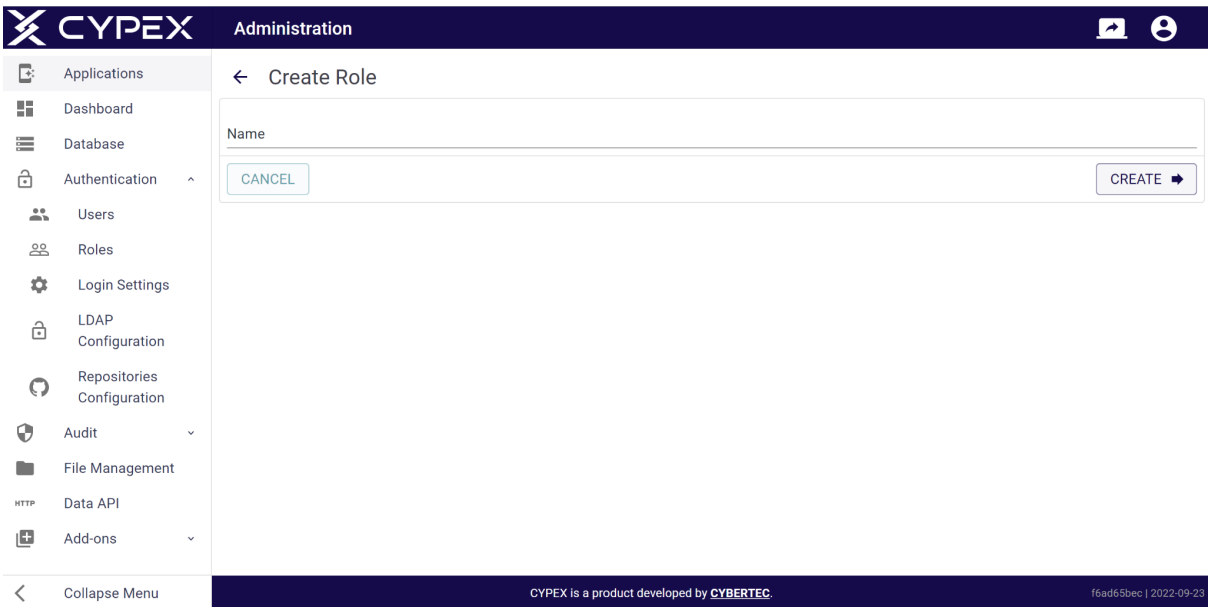
Roles

The next important step is to define roles. Remember, roles are connected to CYPEX users and represent real database side users:



Create Role

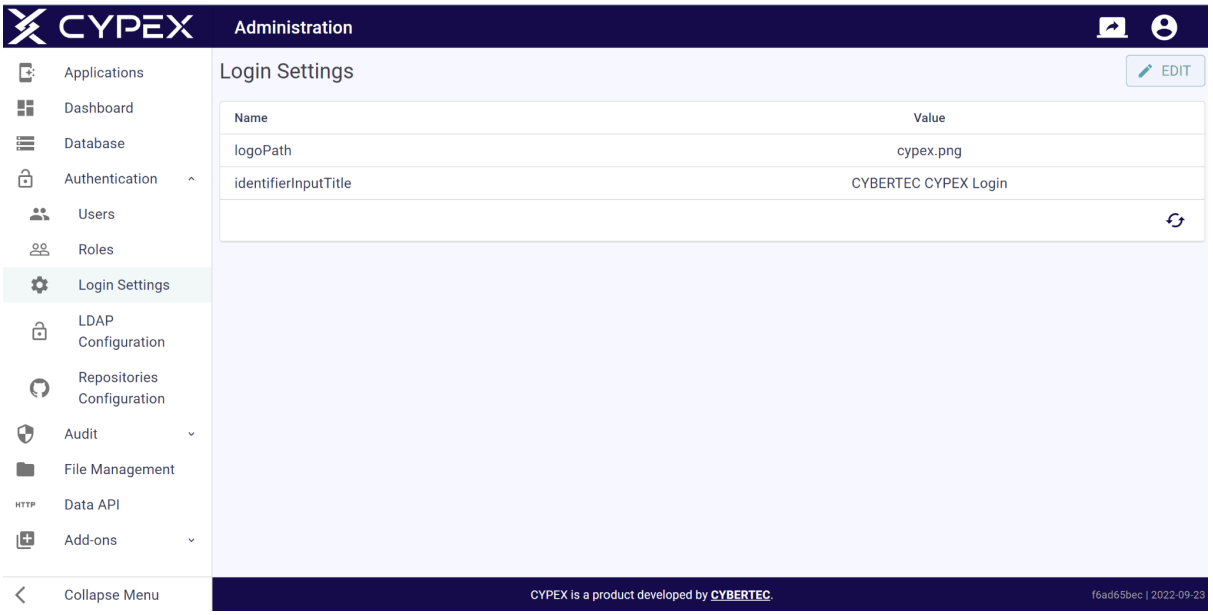
Roles can easily be created in CYPEX:



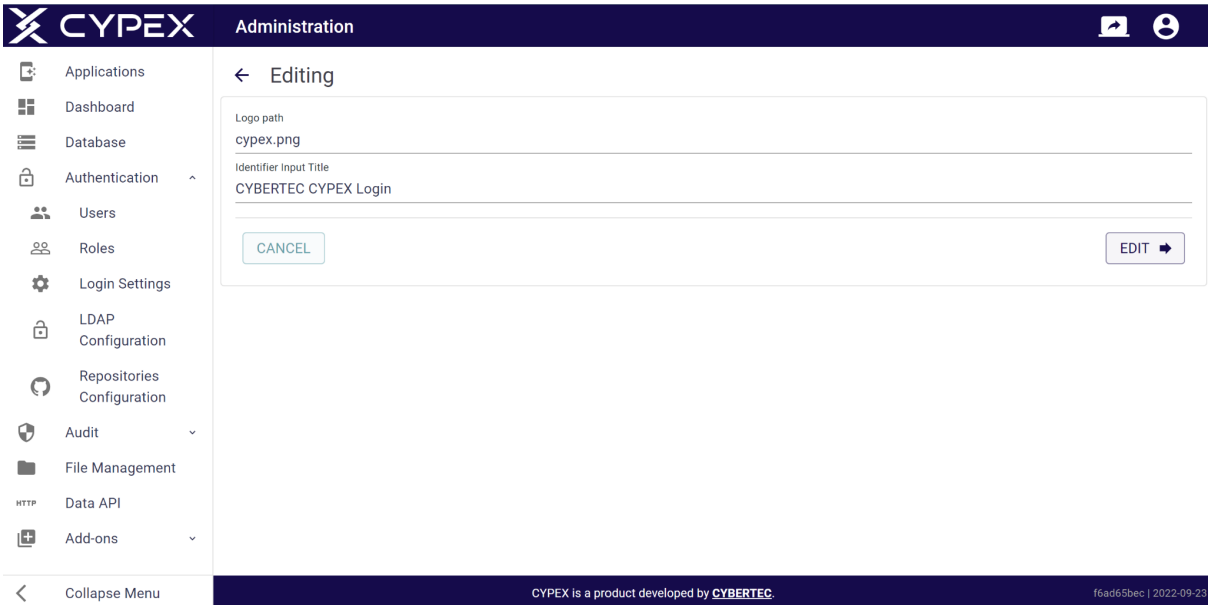
While it's possible to create roles quickly, it's the task of the DBA to assign actual permissions to those roles. At the moment, this is done at the “query” level. Once a query is created, you can assign permissions to roles.

Login Settings

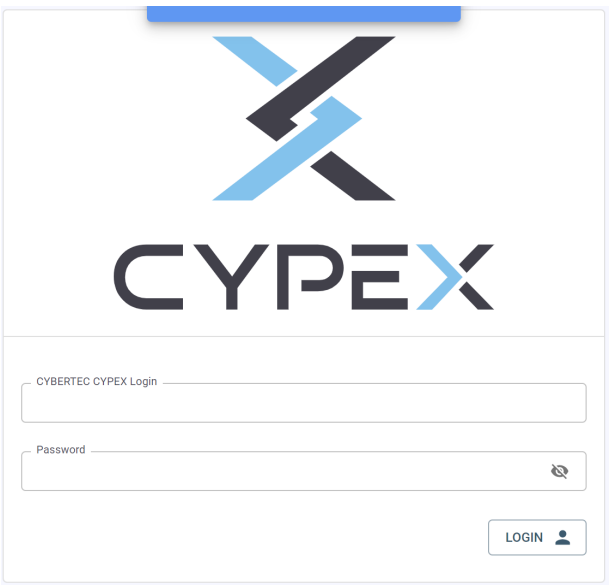
In the “Login Settings” you can define the logo shown during login:



You can upload any suitable logo and easily change the name of the page to adjust CYPEX to your company CI’s needs. Note that the logo you are pointing to has to be in the “public” folder of your webserver:



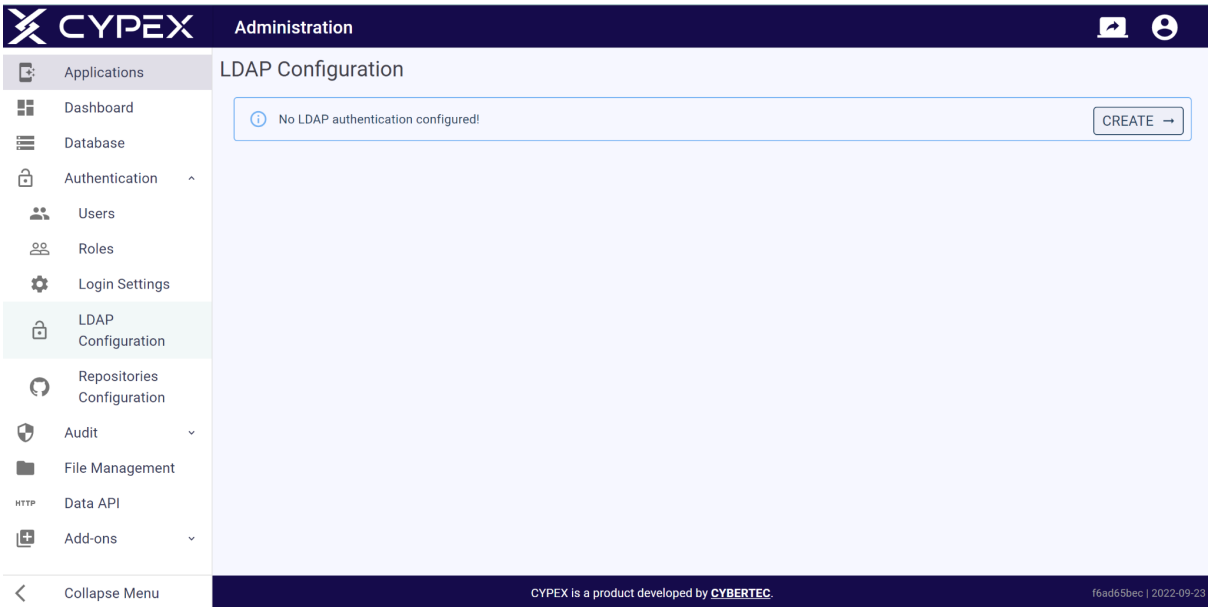
These settings will directly translate to the way the start page looks like. The following screenshot shows what the default layout is like in the standard configuration shipped to customers:


CYPEX
CYBERTEC CYPEX Login
Password
LOGIN

LDAP Configuration

So far you have used local users and local authentication. While this is perfect for small scale deployment, it's not viable for large companies featuring hundreds and maybe thousands of users.

The solution to this problem is “Single-Sign-On”. In CYPEX you can achieve this functionality using LDAP:



Create a new LDAP configuration and connect CYPEX to your LDAP infrastructure. Here is how it works:

You need to fill out a form containing the settings. The LDAP configuration requires the following settings to establish a connection (The list also contains examples of each setting):

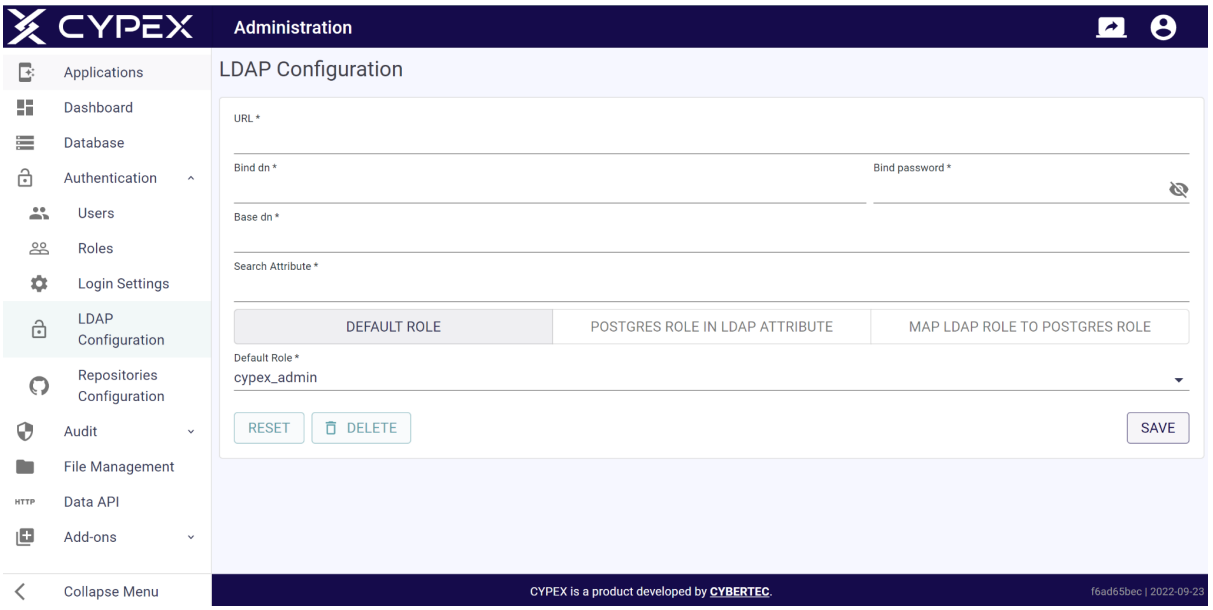
- URL ldap://ldap:10389
- Bind dn cn=admin,dc=cybertec,dc=at
- Bind Password **********
- Base dn ou=people.cd=cybertec, dc=at
- Search Attribute uid

To use LDAP in order to authenticate users and manage rights inside the application, you have three options:

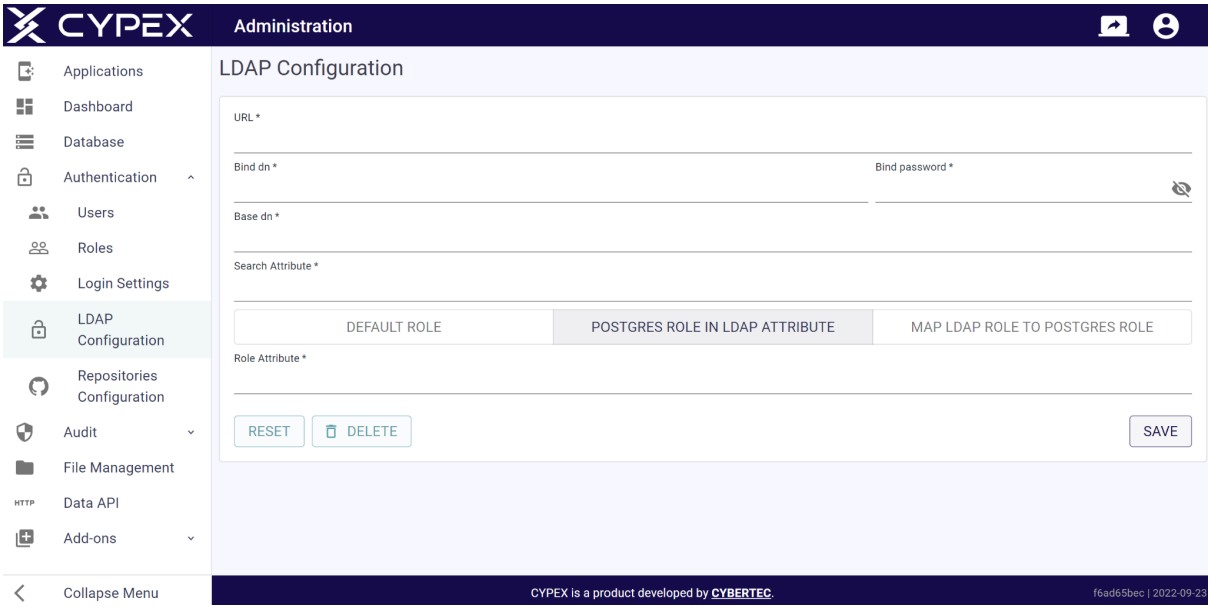
DEFAULT ROLE:
means only one role on the CYPEX side for all LDAP users. If the username and password are correct, the user can log in, and gets assigned this default role.

POSTGRES ROLE IN LDAP ATTRIBUTE:
is the LDAP entity attribute name for the CYPEX role name. If the user logs in and is defined as a cypex_admin in LDAP, the user is set for the attribute cypex_admin in CYPEX

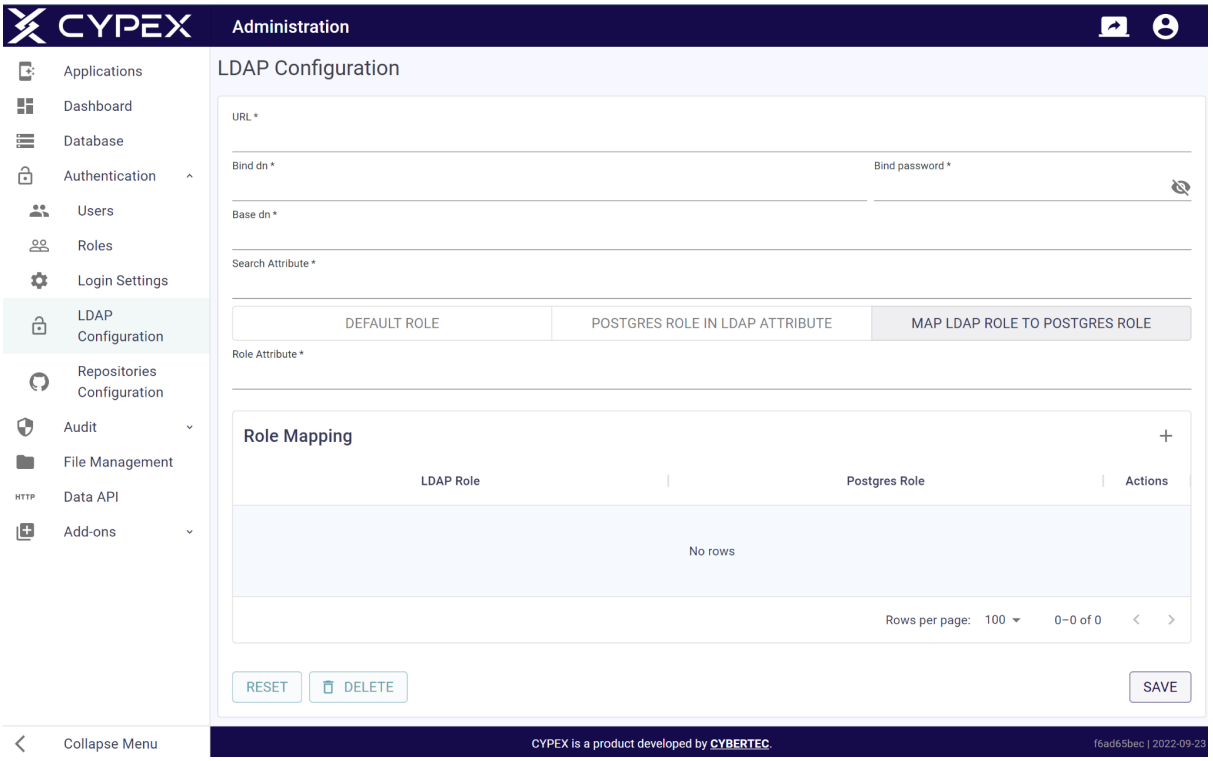
MAP LDAP ROLE TO POSTGRES ROLE:
If the username and password are correct, then the user gets the first match from the mapped roles. In case the user has the LDAP user role and this is mapped to cypex_user, the user has the role cypex_user in CYPEX.



LDAP-Configuration option POSTGRES ROLE In LDAP ATTRIBUTE_



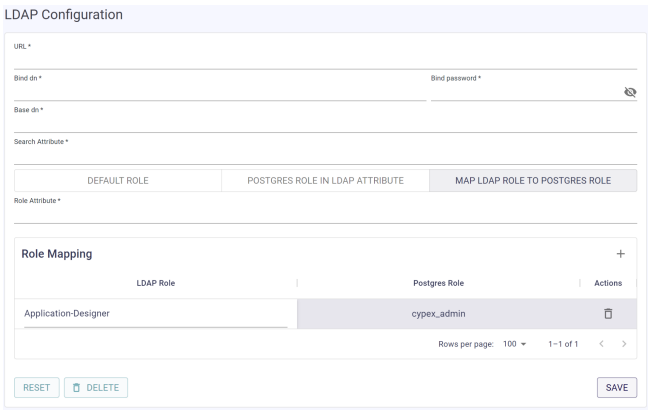
The role attribute where the role is saved on the LDAP side.

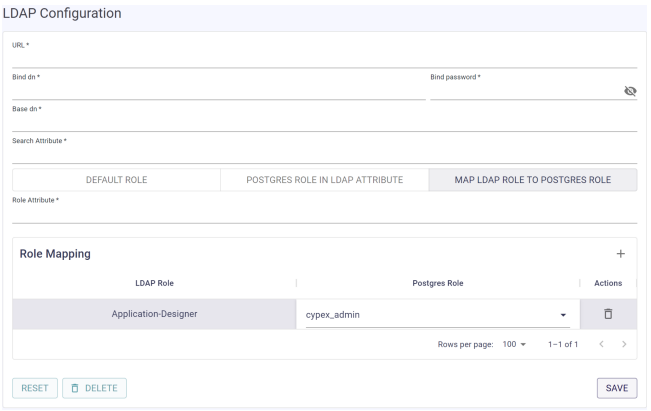


You can map the CYPEX role to the LDAP role.
If the LDAP user is assigned to the “Application Designer” role in LDAP, map this role to the CYPEX cypex_admin role. Check the LDAP group mapping on login to make sure that mapping exists and the CYPEX role is allowed to login.

Mapping also has an effect on the way CYPEX handles things: If the CYPEX role mapped is an admin role, the user is considered a CYPEX admin.

This same mapping and authentication process works for the API as well:

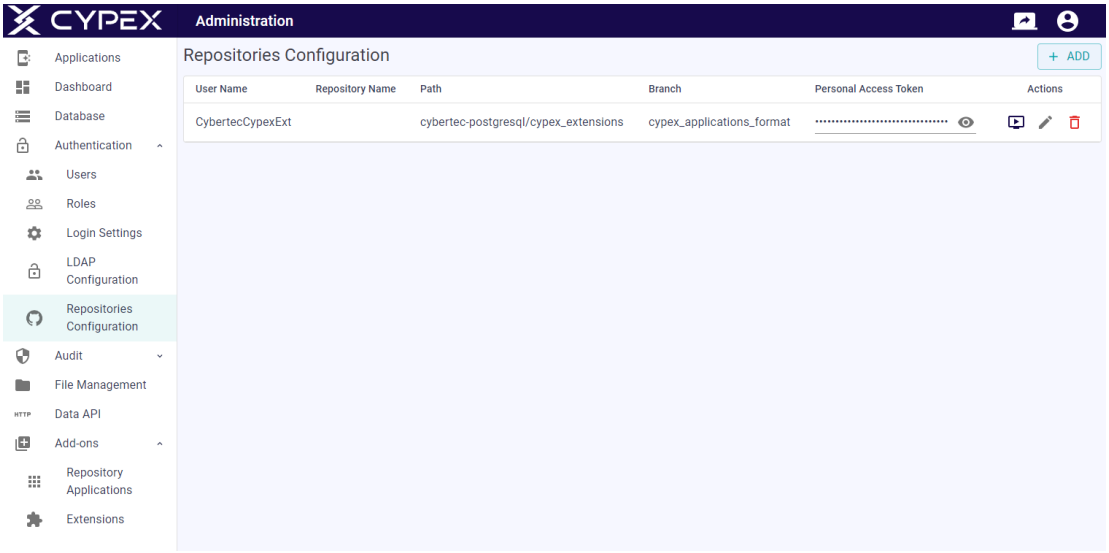




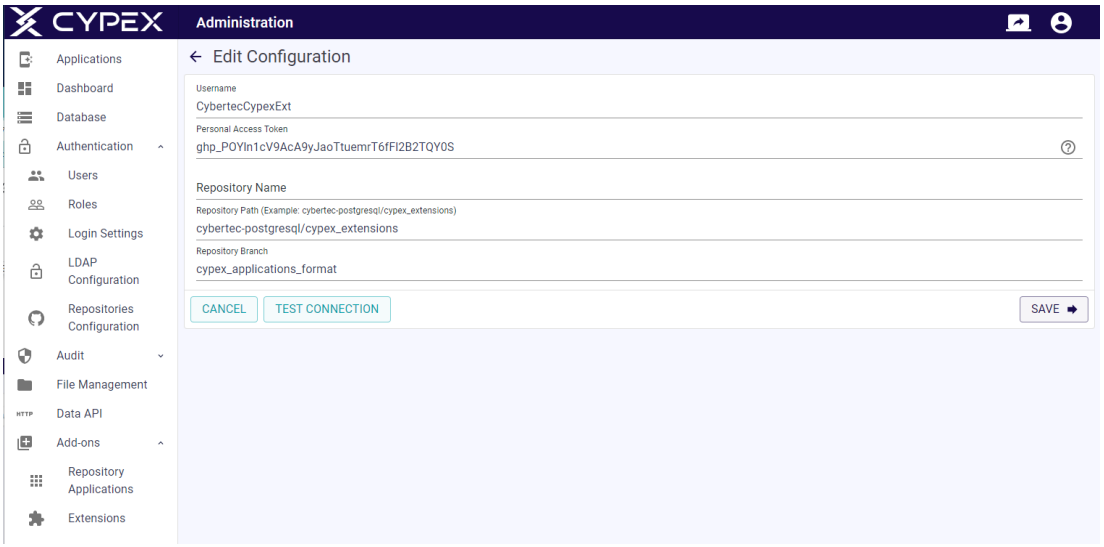
“LDAP Role” is a text input field, and “Postgres Role” is a drop-down containing CYPEX user roles.

Repository Configuration

Finally, you can define the repositories you want to use to handle CYPEX extensions. Basically, you give CYPEX access to a Git repository which contains all of the extensions in a format accessible to CYPEX:



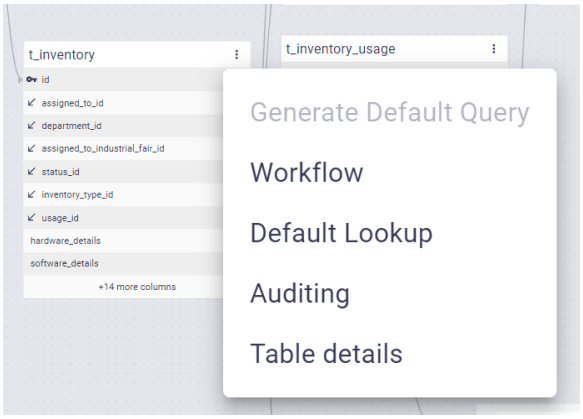
The configuration is straightforward - add the Git data and test the connection. CYPEX is then fully connected and you can easily add extensions to the system as needed:



What is of vital importance here is the use of the personal access token. Github has recently added some security precautions which make this feature necessary.

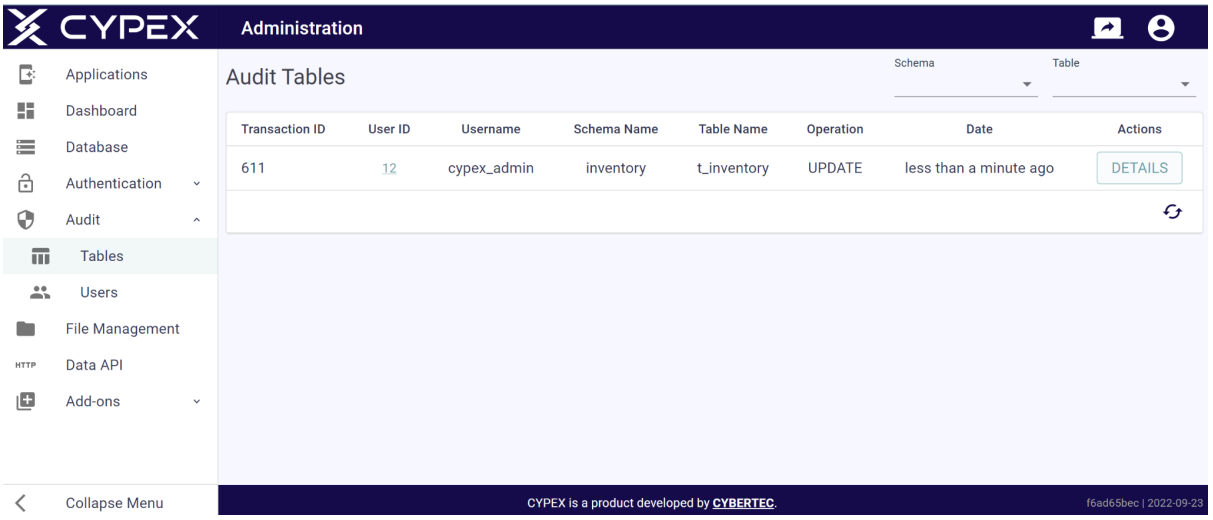
Audit

CYPEX allows users to audit tables. This menu entry will facilitate exactly that:

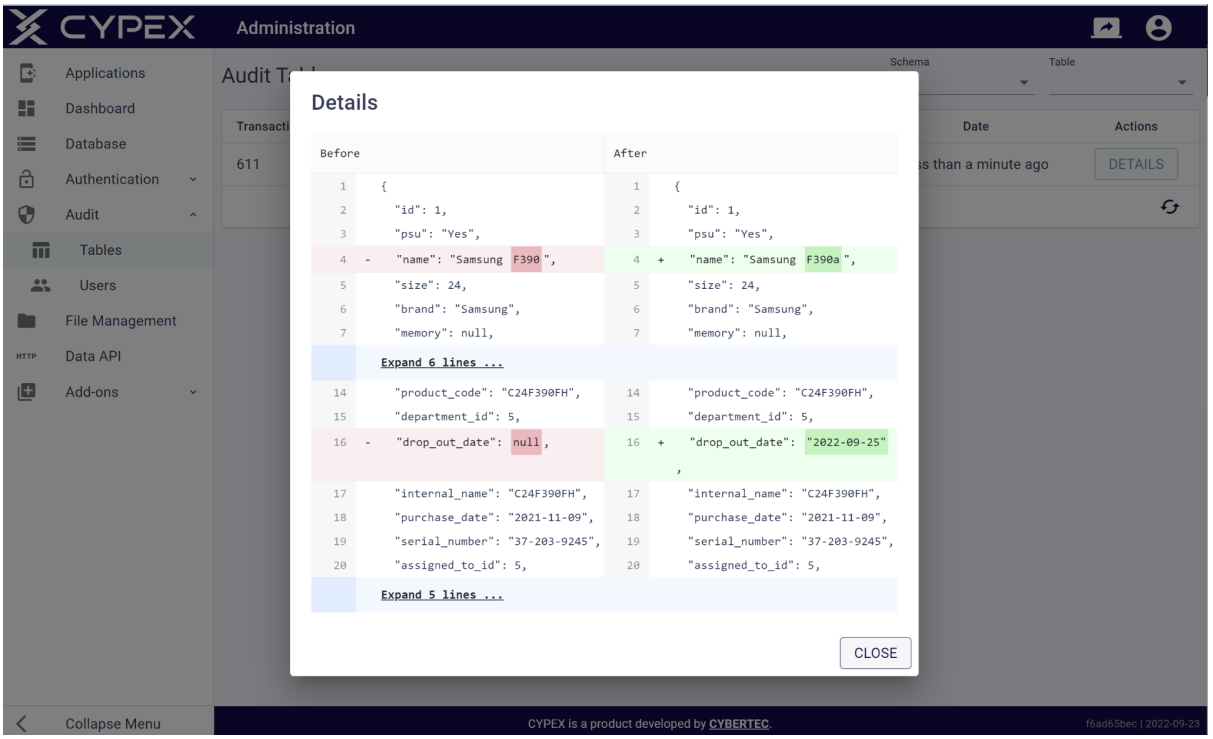


Tables

When a table is audited, the audit trail will be visible in the “Audit -> Tables” section. The following screenshot shows what that might look like:



PostgreSQL will capture all changes and display them in an easy-to-read format. One can see a “diff” of what has changed which allows you to gain an overview quickly:

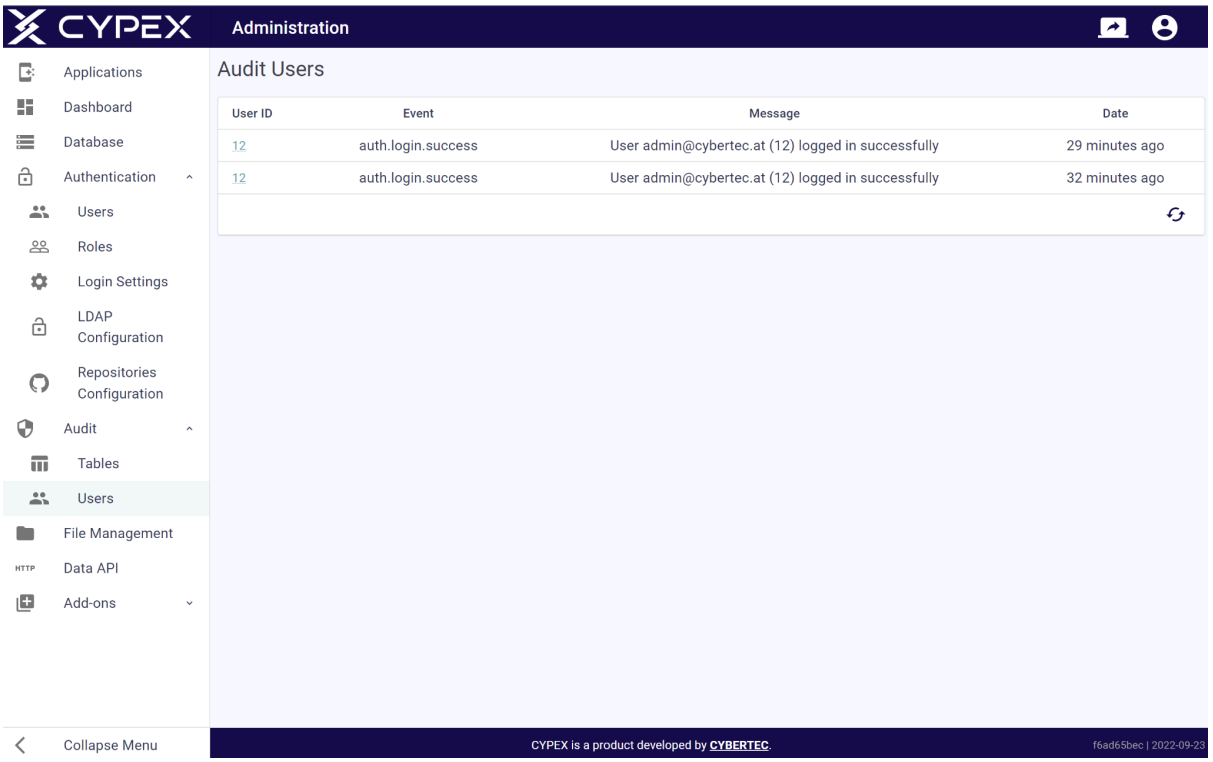


Keep in mind that the audit trail can accumulate large amounts of data, and thus keeping an eye on storage usage is of vital importance to the system.

Users

However, CYPEX does not only audit the changes made to tables. It's also important to keep an eye on how users behave and which login activity can be observed. The “Users” section does exactly that. It contains vital information about who has logged in successfully, and who has failed.

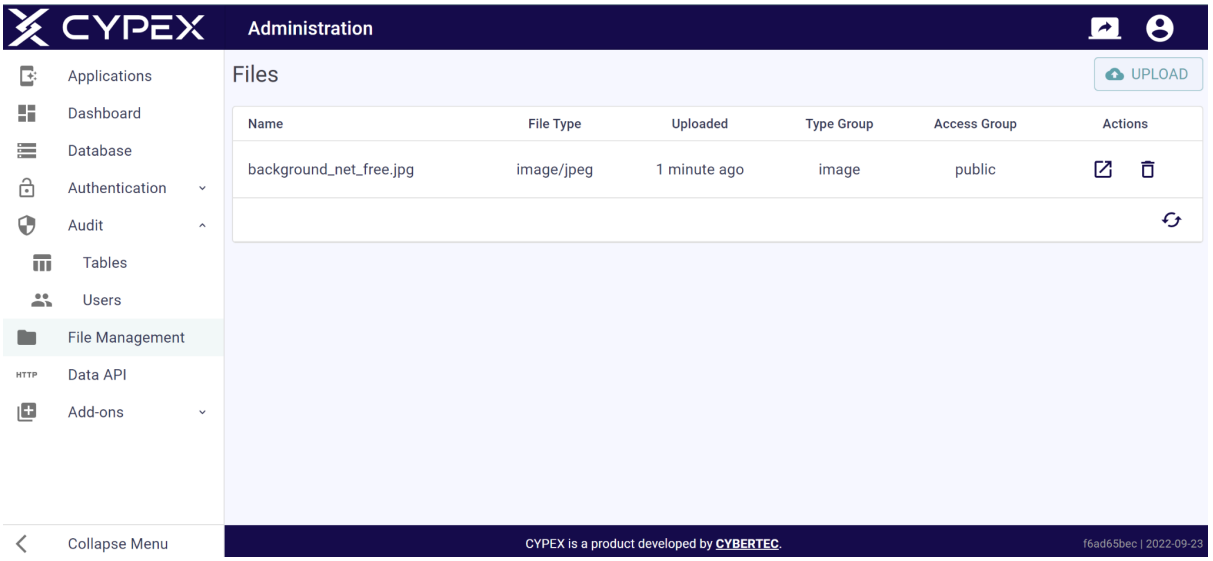
The goal is to give you a quick yet comprehensive overview of the login activity related to your application:



File Management

The next big feature of CYPEX we want to focus on is the ability to upload files. Files are stored in the database. Storing files in the database has always been controversial. However, in this case it's done to ensure that all data including the application itself can be saved using standard PostgreSQL backups. There is no need to back up the database, the application and those files separately - everything is in the same backup. In addition to that, files are handled in a transparent manner which brings countless advantages if you are dealing with workflows.

The file upload screen is easy to understand:

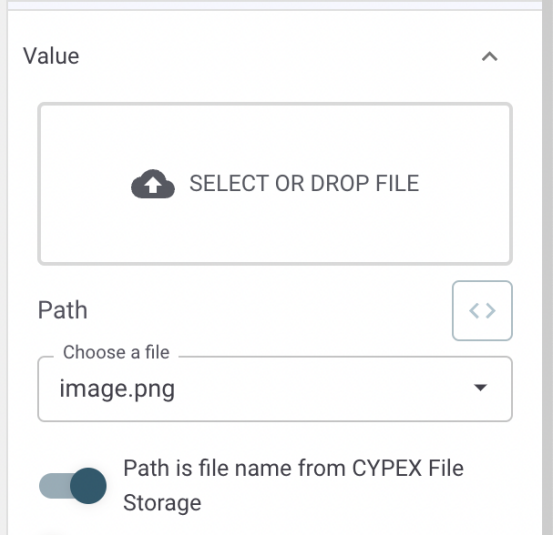


The core question is: Why would anybody use this feature? What are the benefits of such infrastructure? Here are some typical use cases:

- Display images
- Offer downloads
- Send as email attachments

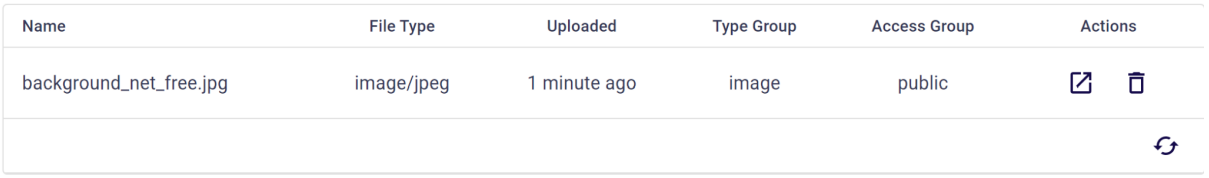
Once a file has been uploaded, you can define user permissions to define who is allowed to access the file. As with all other data, permissions are handled by the database directly and are therefore identical within the entire stack (API, GUI, etc.).

Inside the WYSIWYG editor the application designer can use of such files:



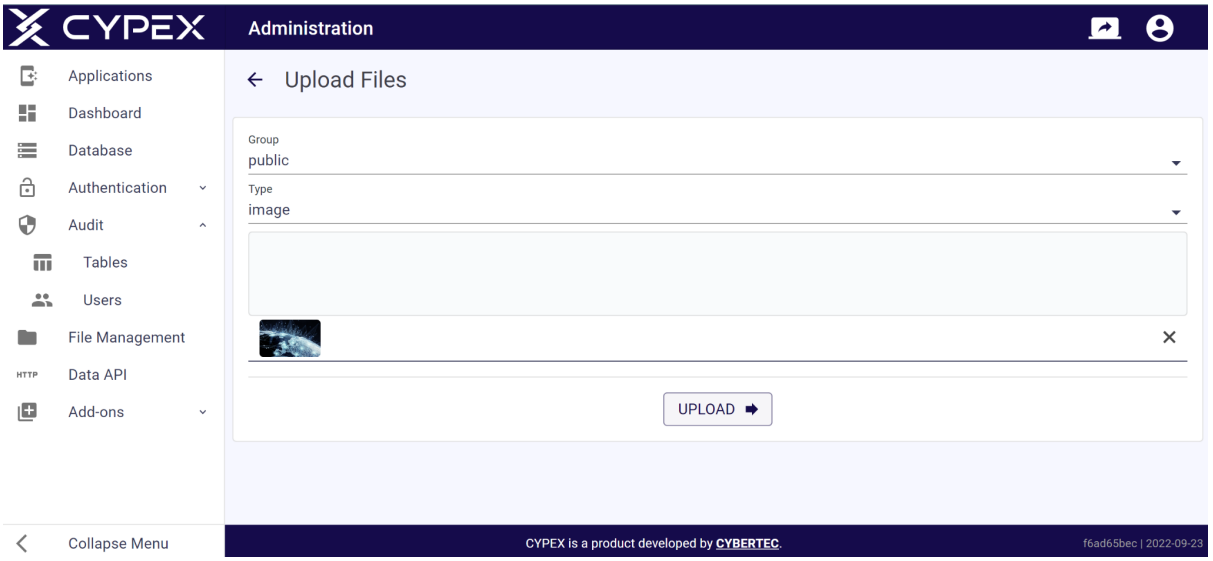
Developers have direct access to CYPEX storage and files can be taken from there.

Here is what the list of files might look like in your admin panel:



Upload Files

The upload facility is capable of handling reasonably sized files. In general the infrastructure is usually used for pictures as well as documents (PDFs, etc.) which are static in nature:



Various types of binary files are supported. The following list contains an overview of what is possible:

- audio
- document
- image
- other
- text

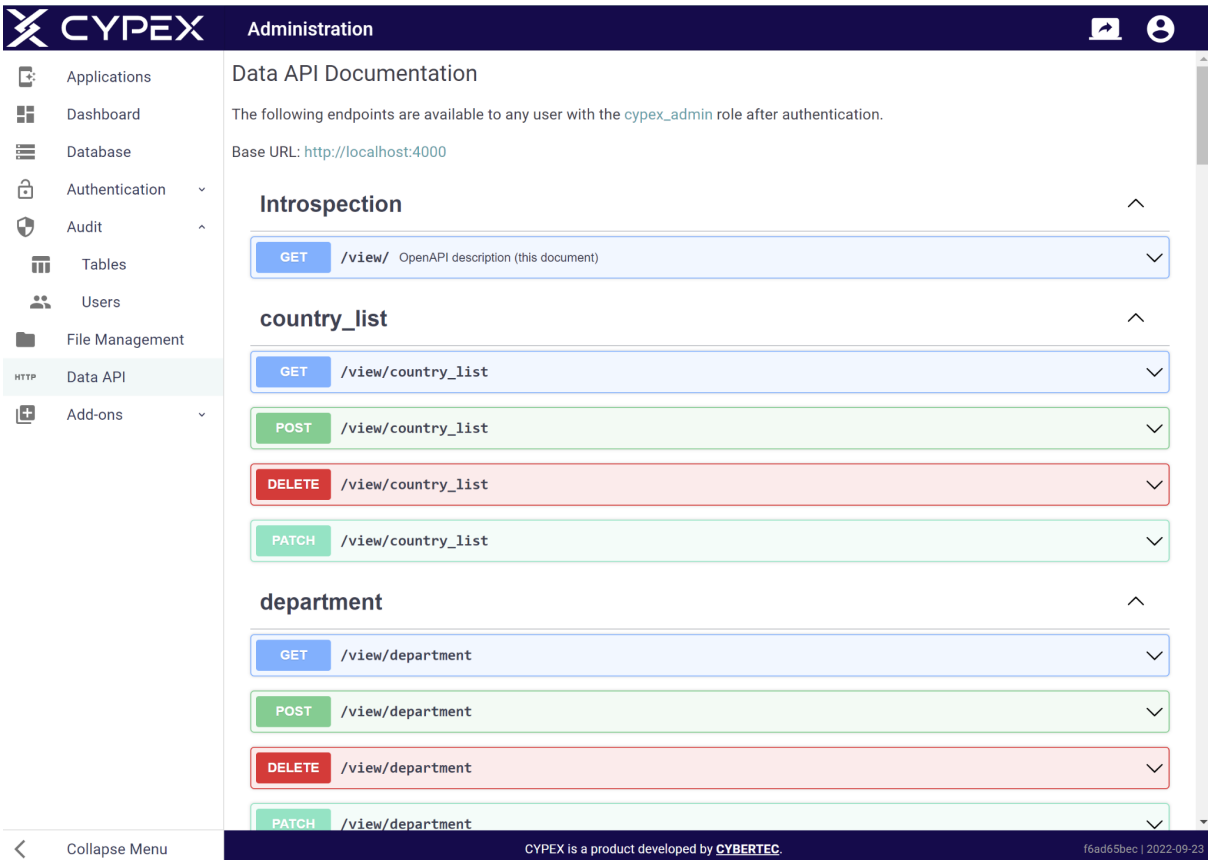
In the future, additional file types will most likely be added, in order to make this feature even more powerful.

Data API

The CYPEX data API is an integral part of the system. Every query is automatically exposed as an API endpoint. The infrastructure will honor access privileges and automatically keep the API up to date.

The general purpose of the API is to allow users to build custom apps which are hard to create with the builtin-WYSIWYG editor. In addition to that it allows for easier integration with other infrastructure components. It's important to understand in this context that CYPEX isn't "all or nothing" - it's perfectly feasible to only use the API.

The purpose of the "Data API" section is to give users a simple method to test the API generated by CYPEX. As you can see in the screenshot below, a list of all endpoints is generated automatically:



These endpoints can be tested directly.

Note: When you're testing the API, keep in mind that you aren't working in a sandboxed environment. This is the real thing and changes will make it to the

underlying database (assuming the API call is successful). Therefore caution is advised.

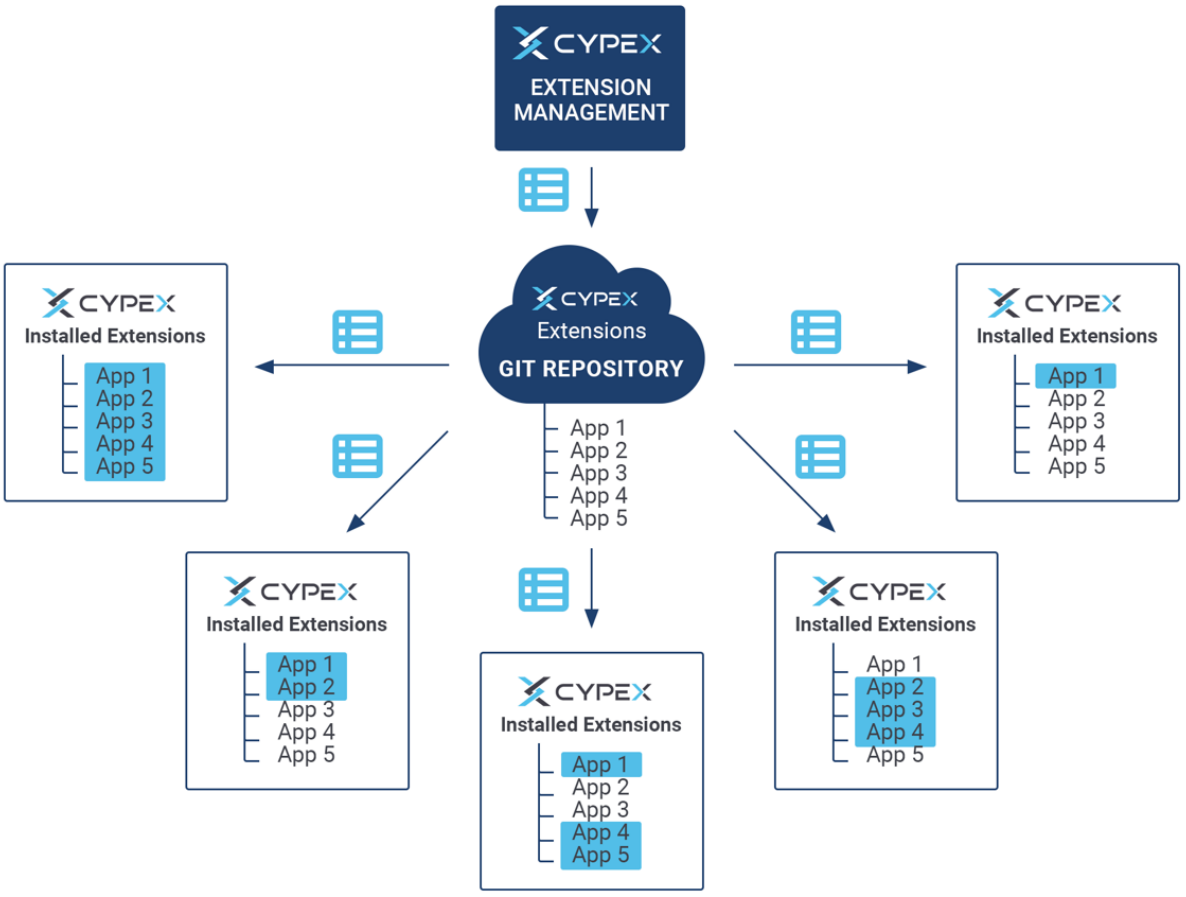
Add-Ons

In this section we'll dig into CYPEX extensibility and learn what can be done to make CYPEX even more powerful by adding code from external sources.

Repository applications

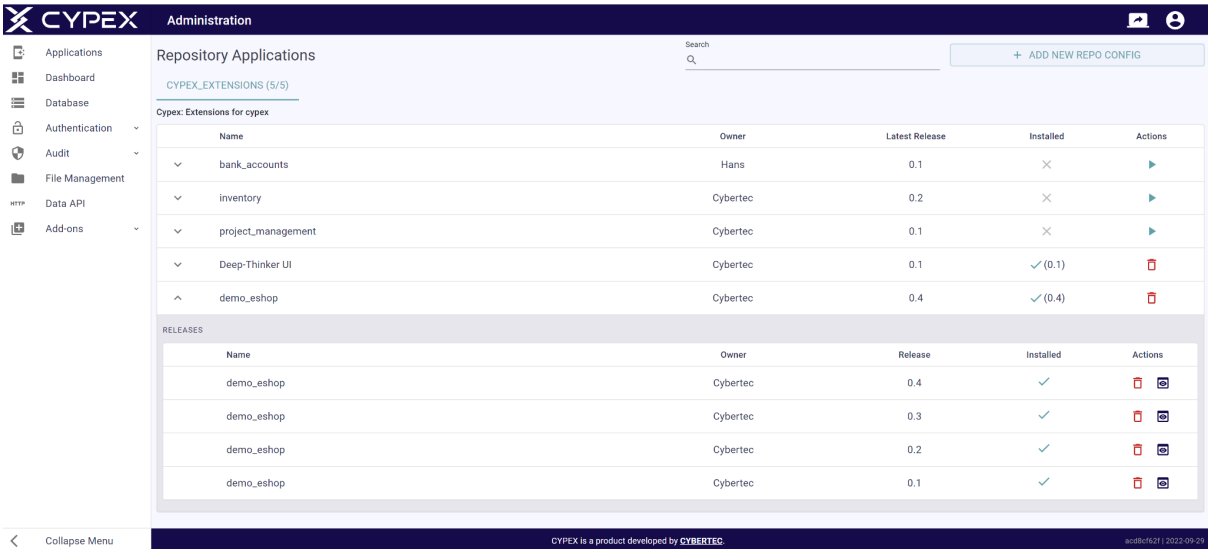
CYPEX allows users to define ready-to-use PostgreSQL extensions. Those extensions can be integrated into existing applications to simplify the model creation process and to automate as many steps as possible.

The way external extensions are supported is as follows:

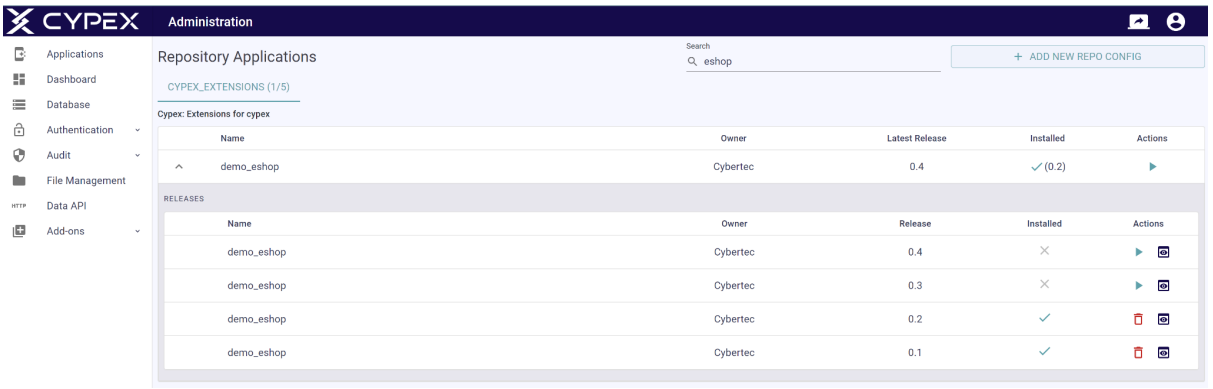


CYPEX allows users to define a set of Git repositories which can be used to fetch extensions. It allows you to deploy everything from small SQL fragments and simple procedures all the way up to full-fledged complex data models.

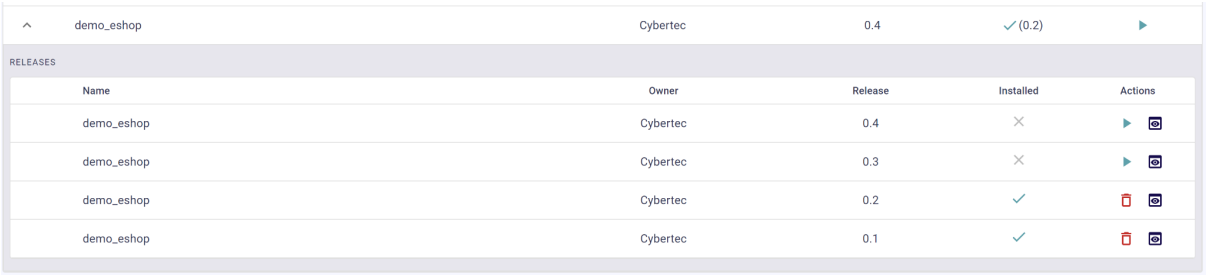
The “Repository Applications” menu entry allows you to quickly load entire applications from the Git repository.



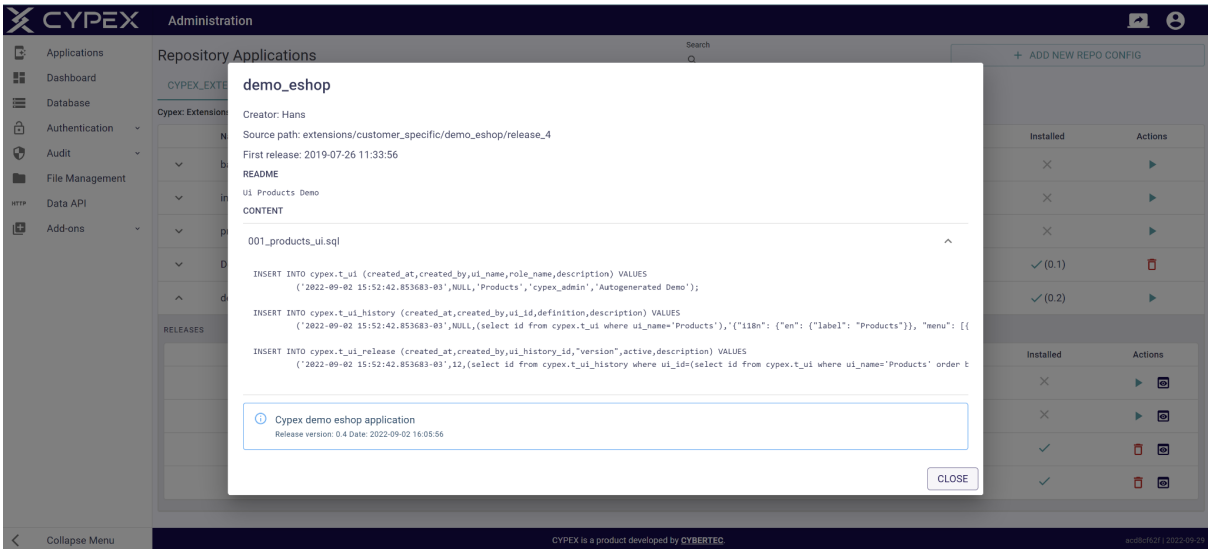
The search field allows you to search simultaneously in all available repositories. The numbers beside the repository name show how many applications are found, out of the total number of applications.



For each application, several different releases can be installed and uninstalled. That means you may decide whether to use the latest release of the application or not.



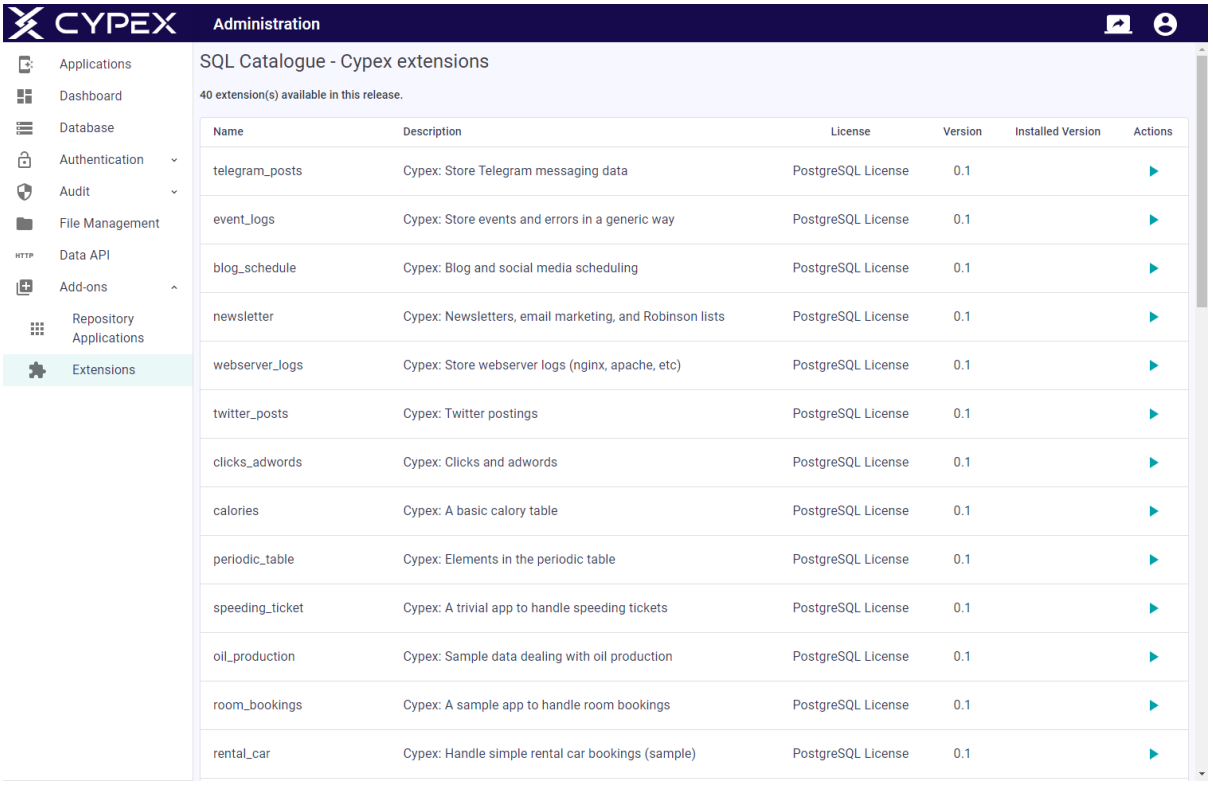
It's easy to learn more about the application which has been deployed. CYPEX provides you with metadata concerning your application, as shown in the next image:



Extensions

To install SQL fragments you can use the “Extensions” entry. Click on the “action” button and to easily deploy extensions.

There are countless extensions inside the default repository which can be used:



By default, the CYBERTEC repository for CYPEX is enabled. However, you can easily add more repositories as needed. For more information, see the section on Repository Configuration.

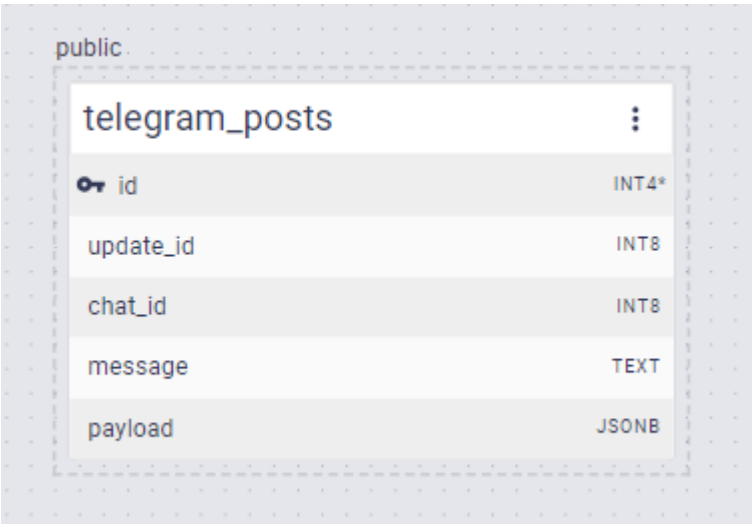
Available CYPEX extensions

CYBERTEC provides a set of extensions which can be used to simplify the process of building applications. In this section you'll learn which extensions exist and what they are capable of doing.

Extension: telegram_posts

Purpose:
Store telegram posts

ER model:

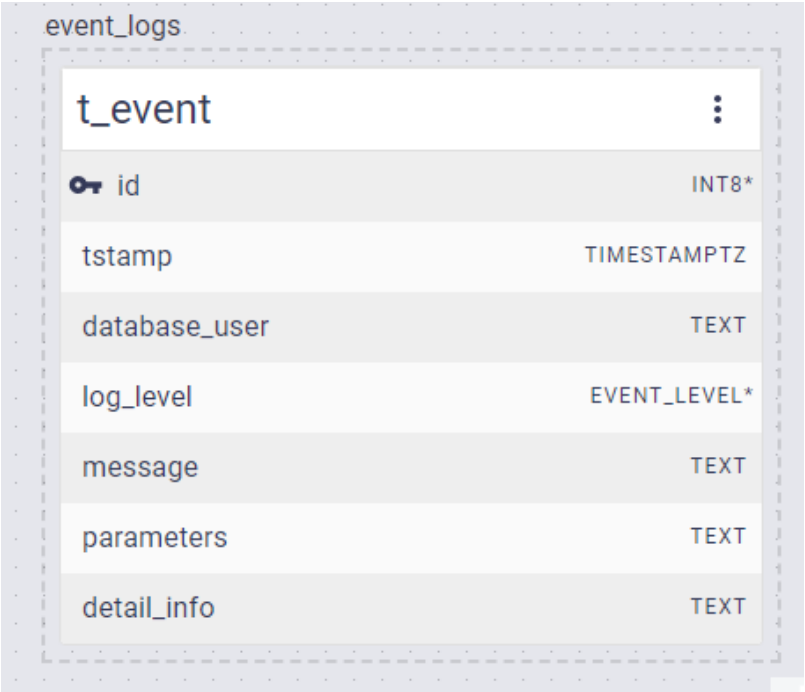


Description:
The extension consists of just one table. The content of the message is stored in the “payload” column.

Extension: event_logs

Purpose:
This extension provides a generic extension to store log entries and events. The idea is to generalize event messages.

ER model:



Description:
The database user is stored as text and not as an object id. The reason is that we want to support “DROP USER” in PostgreSQL and allow for more generic usage.

The log_level is represented as enum type in PostgreSQL which allows for sorting. The following sort order is used:

- INFO
- NOTICE
- LOG
- WARNING
- ERROR
- FATAL

Note that PostgreSQL will provide this order automatically. It's also possible to filter easily.

Extension: blog_schedule

Purpose:
This module offers a simple way to manage blogs, posts and authors.

ER model:

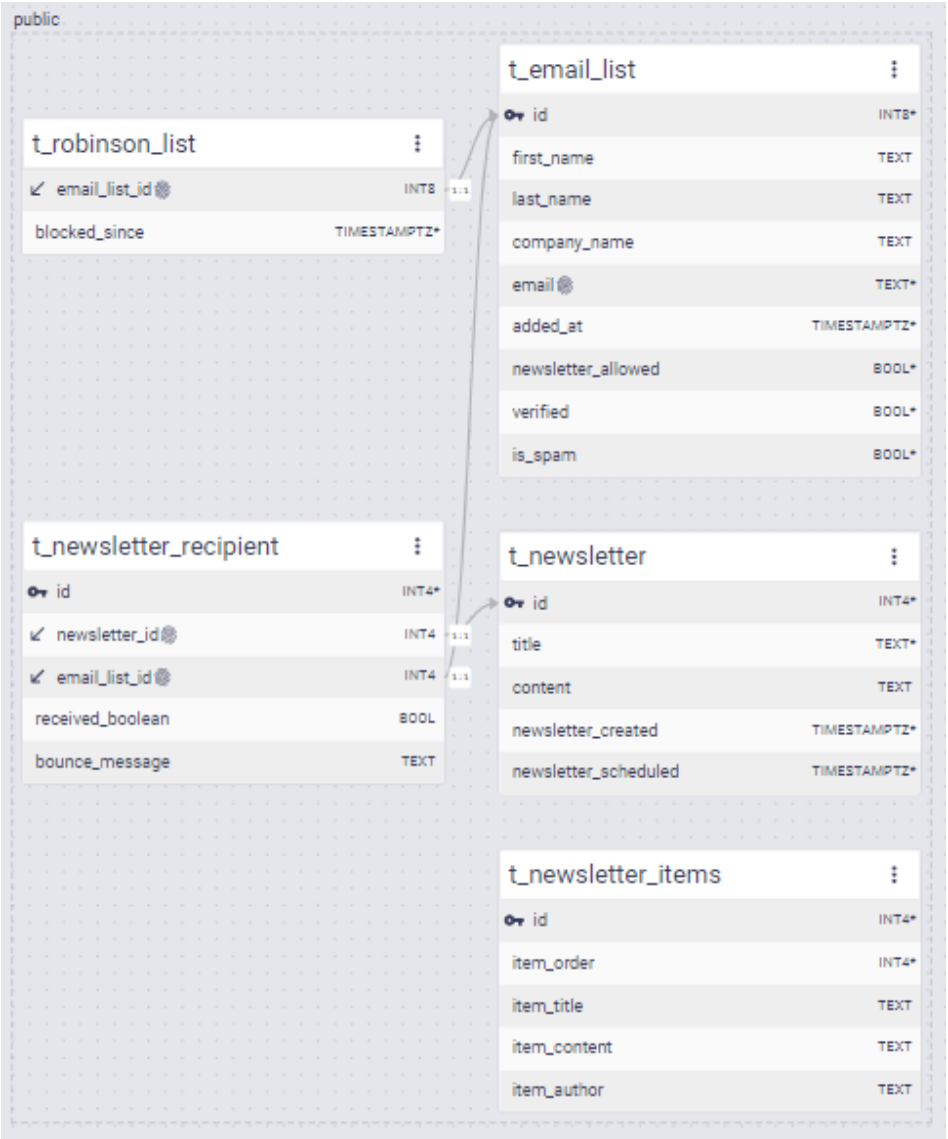
Description:
Authors are identified by email address (unique field). The table is structured in a way it can be extended easily (fields for phone, etc.). Posts contain boolean field to identify the status (proofreading yes / no). Blogs have a title. The payload is intentionally not part of the table as a blog might need various fields to store the content (payload, images, etc.) - those are supposed to be added by the ER design person.

Sample data isn't included.

Extension: newsletter

Purpose:
This module can handle newsletter as well as blacklists.

ER model:



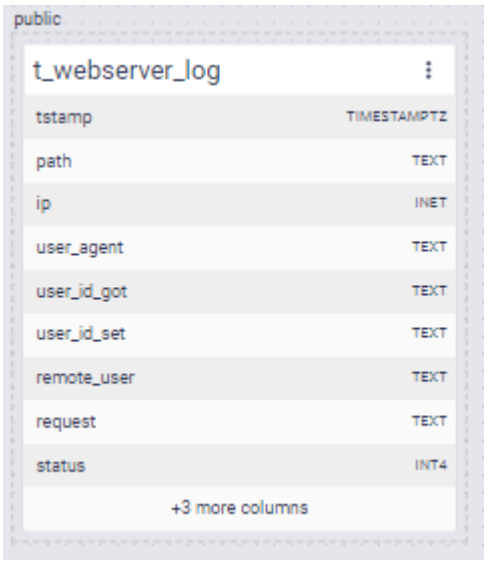
Description:
In email marketing a “Robinson list” is a list indicating who isn't supposed to receive messages. It's basically a “blacklist”. In the “email list” itself, we store if newsletters are allowed or if an address is marked as spam. However, it can still happen that emails bounce. In this case the bounce message is stored in the newsletter recipient table.

Sample data isn't included.

Extension: webserver_logs

Purpose:
This is a basic module to store web server logs in a table.

ER model:



Description:
The module consists of just one table. It stores the typical data one would find in a web server log as a database entry. We are using PostgreSQL optimized data types to handle IPs. The http status is stored as a simple integer value. Note that the “tstamp” column represents the insert-time into PostgreSQL (default value = clock_timestamp()). The timestamp as observed by the webserver is stored in “request_time”.

Sample data isn't included.

Extension: twitter_posts

Purpose:
An extension to store twitter posts.

ER model:

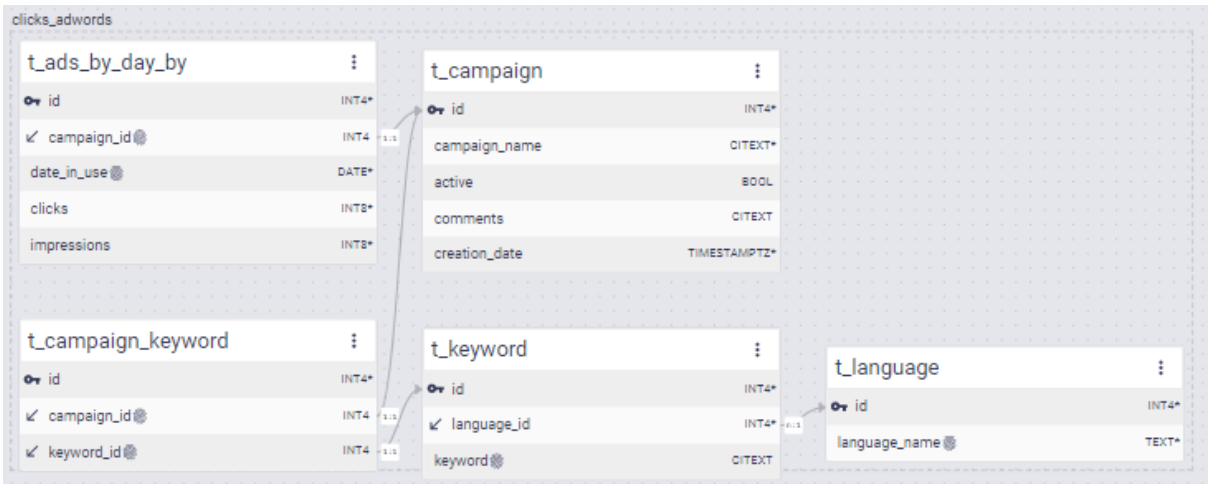
Description:
The module consists of just one table capable of storing twitter messages. This is basically a 1:1 copy of the Twitter API (which is also the foundation of the twitter_fdw).

Sample data isn't included.

Extension: clicks_adwords

Purpose:
Handling extensions, campaigns, keywords and clicks.

ER model:

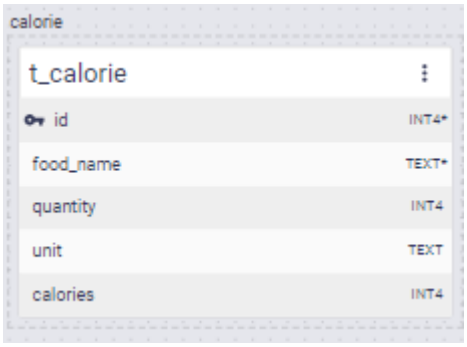


Description:
The model allows storing keywords (t_keyword) in various languages (t_language). Those keywords are associated with campaigns. For each campaign the extension analyzes how many keywords were clicked and how often on each day.

Extension: calories

Purpose:
Storing the energy content of food.

ER model:



Description:
We store the energy content of food as measured in calories. By default the table is empty. However, when looking at the content of the extension in Git you'll notice that sample data is available but it's commented out. It should be easy to load this information if needed.

Extension: periodic_table

Purpose:
Storing elements in the periodic table.

ER model:

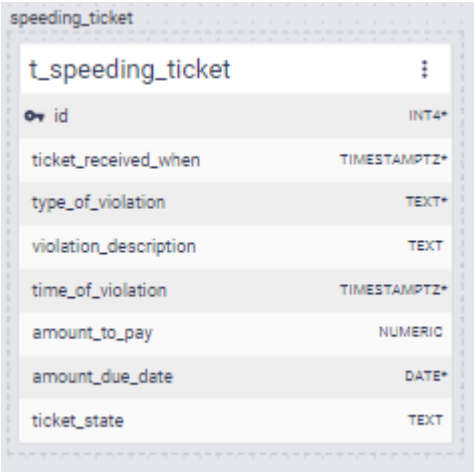
Description:
We store elements in the periodic table including a complete list. The name of the element is stored in English.

Extension: speeding_ticket

Purpose:

A sample app to store speeding tickets.

ER model:



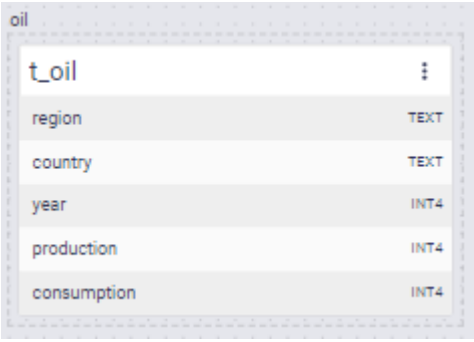
Description:
We store information about speeding tickets. The purpose of the extension is more for educational purposes.

Extension: oil_production

Purpose:

Sample data taken from the oil industry.

ER model:



Description:

This table contains data sets (644 records) from the oil industry. The purpose of this extension is mostly educational. It's ideal to teach windowing functions, analytics and time series analysis.

Extension: room_bookings

Purpose:

Handling basic room reservations.

ER model:

Description:

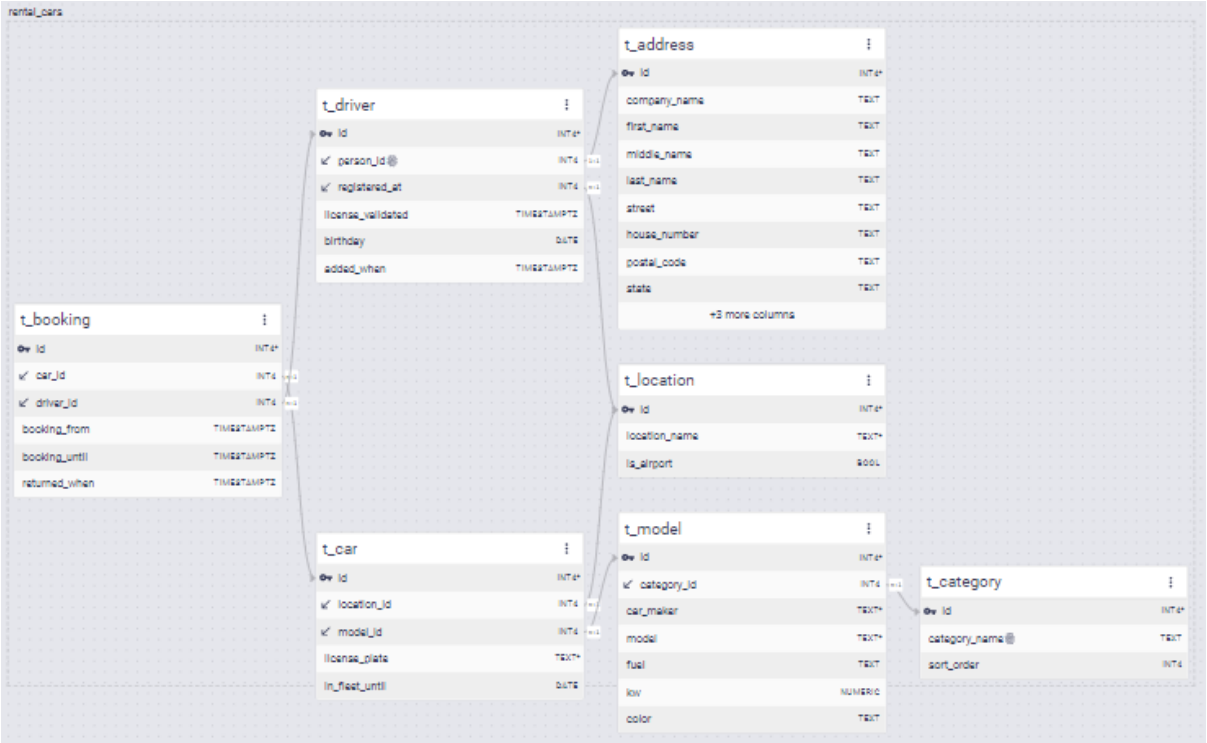
This module stores hotel bookings. Hotels are associated with bookings. Note that there is no foreign key relation between customers and bookings. We do so to ensure that customers can be deleted without destroying historic bookings.

Extension: rental_car

Purpose:

An ER model to handle rental cars

ER model:



Description:

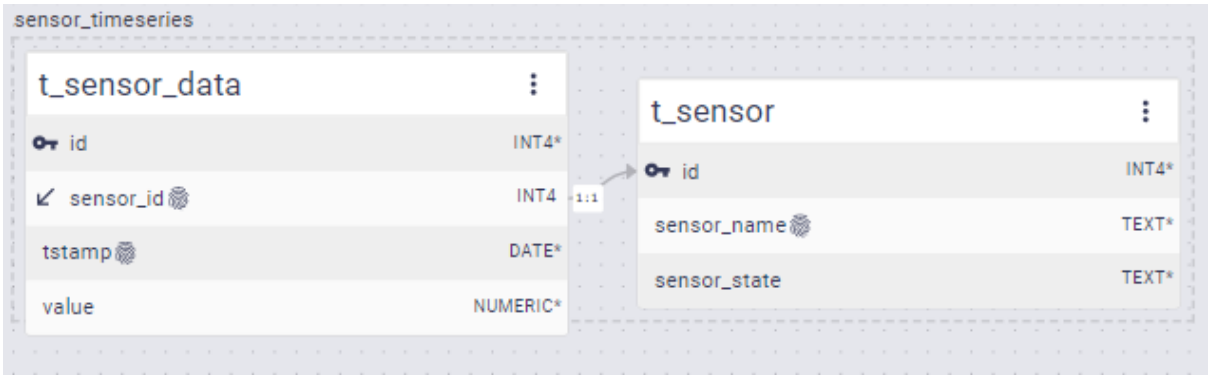
This extension helps to manage rental cars. It stores information about categories, models, locations as well as drivers and bookings. It's a blueprint for helping people to get started quickly.

Note that exclusion operators are used to avoid overlapping bookings. Check out our blog posts dealing with exclusion operators to learn more.

Extension: sensor_timeseries

Purpose:
Handle sensors and time series.

ER model:



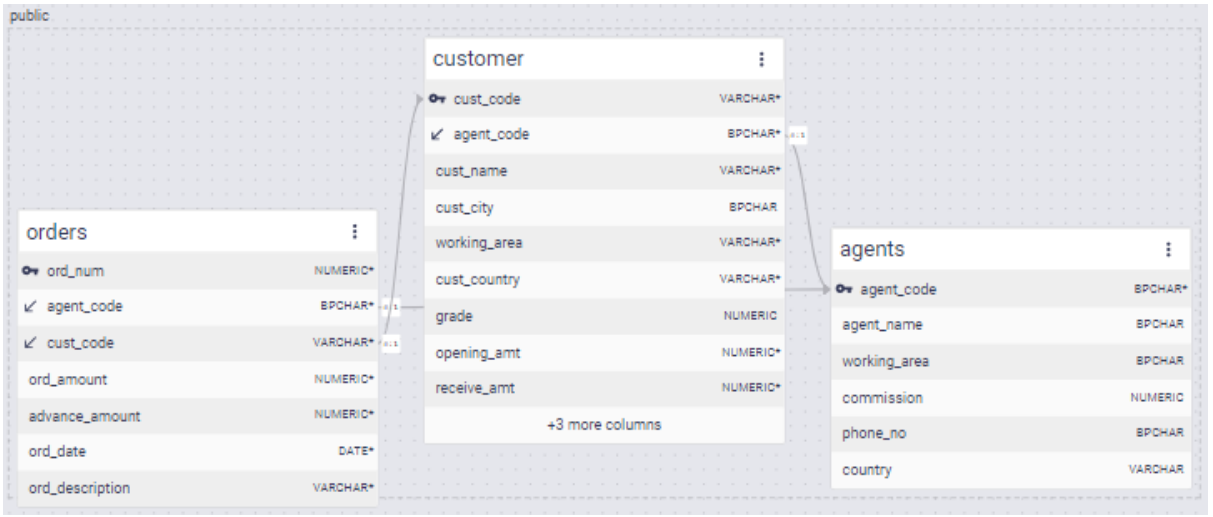
Description:
The model consists of two tables. Sensors and sensor data. It's a blueprint to getting started and to store more comprehensive information in a simple manner. Note that in case you want to store billions of rows, partitioning the sensor_data table is an option for scalability reasons. CYPEX is perfectly capable of handling partitioning.

Extension: agents_customers_orders

Purpose:

A basic model to handle agents and sales orders.

ER model:



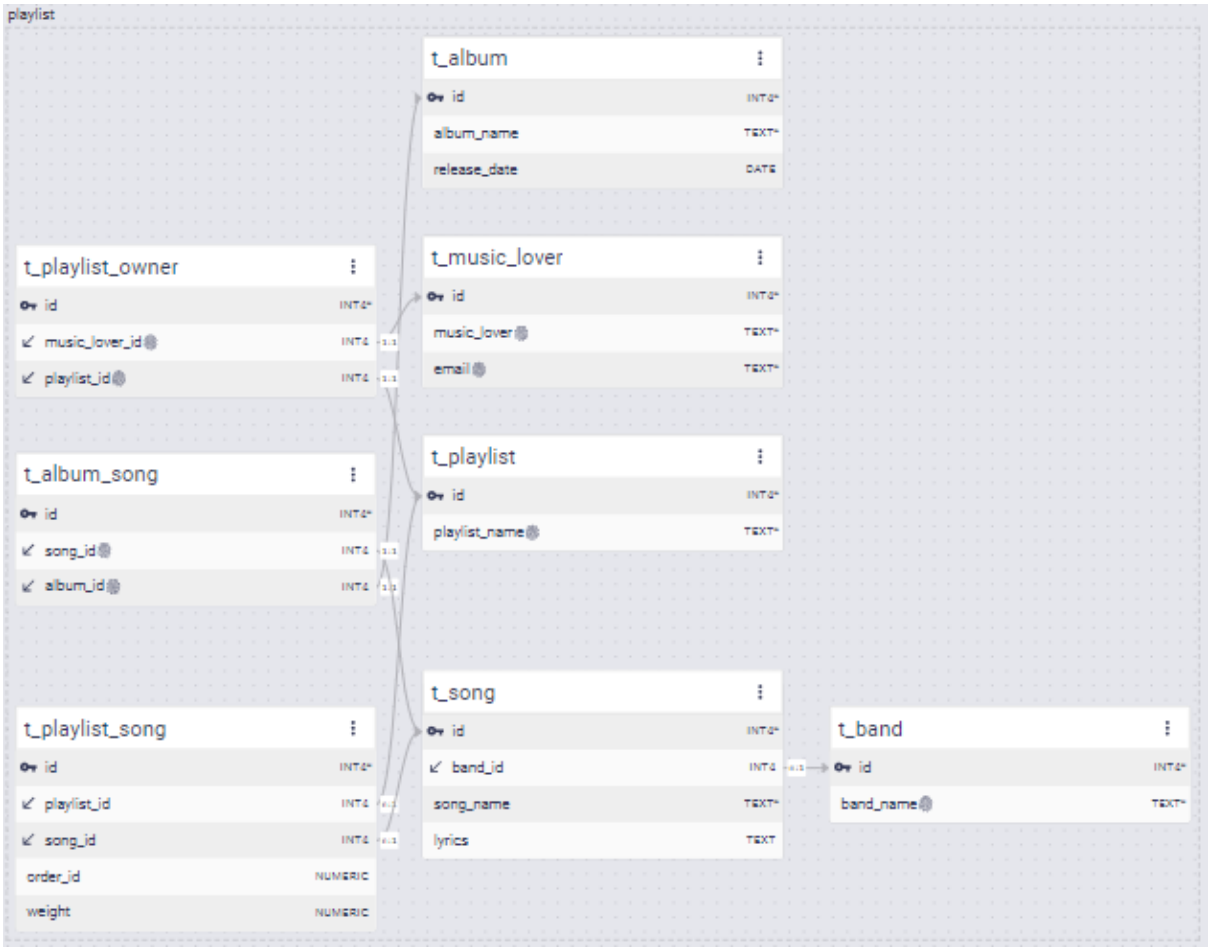
Description:

This module is mainly used for educational purposes. It stores information about agents, customers and customer orders. It's a basic 3-table model which can be expanded upon.

Extension: playlist

Purpose:
A model to handle music

ER model:



Description:
This module is mostly a way to demonstrate the capabilities of CYPEX in an educational setting. It stores information about bands, songs, playlists and a lot more. It's one of the bigger extensions used for educational purposes.

Extension: persons_and_friends

Purpose:

A model to handle friendship.

ER model:

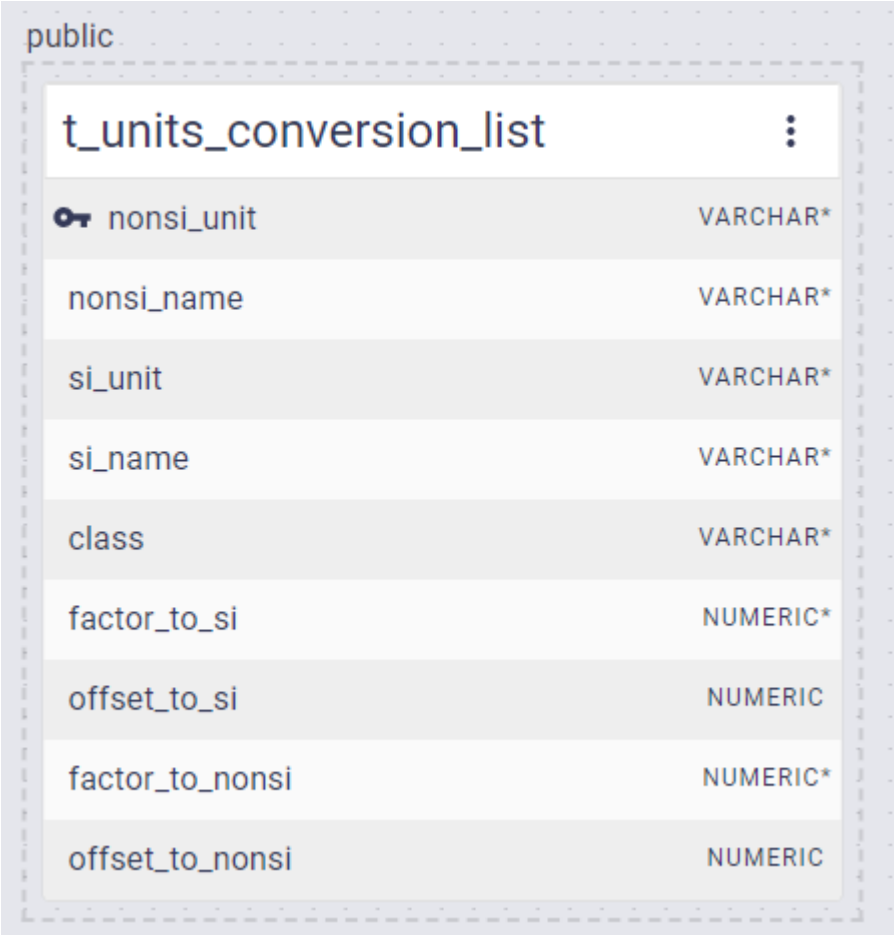
Description:

This module describes friendship relations. A list of people is used to connect friendship relations (m : n). It's an ideal start to model all kinds of relationships.

Extension: unit_conversions_list

Purpose:
A powerful module to handle unit conversions

ER model:



Description:
This module contains functions which can be used in CYPEX to perform all kinds of unit conversions (e.g. km -> meters and alike). It contains a handful of stored procedures as well as a config table holding information about conversion rules.

Converting a numeric value from one unit to some other unit:

```
CREATE OR REPLACE FUNCTION convert_units(
      value           numeric,
      input_units     varchar(50),
      output_units    varchar(50)
```

```
)
RETURNS numeric …
```

The following listing shows, how meters can be converted to kilometers:

```
test=# SELECT convert_units(100, 'm', 'km');
 convert_units
---------------
         0.100
(1 row)
```

Check if a unit can be converted or not (km -> meter is ok, km -> gallons isn't). The function will error out in case a conversion is impossible:

```
CREATE OR REPLACE FUNCTION check_units(
      input_units     varchar(50),
      output_units    varchar(50)
)
RETURNS void …
```

Convert a unit to its standard unit:

```
CREATE OR REPLACE FUNCTION convert_units_from_si(
      value                 numeric,
      output_units          varchar(50)
)
RETURNS numeric …
```

The following example shows how 100 meters can be converted to the standard unit (1 km units):

```
test=# SELECT convert_units_from_si(100, 'km');
 convert_units_from_si
-----------------------
                 0.100
(1 row)
```

The entire process is driven by configuration tables:

```
INSERT INTO t_units_conversion_list  VALUES
-- temperature
('F','farenheit','K','kelvin','temperature', .55555555, 255.37222222, 1.8,
-459.67),
('C','celsius','K','kelvin','temperature', 1.0, 273.15, 1.0 , -273.15),
('R','rankine','K','kelvin','temperature', .55555555, 0.0, 1.8, 0.0);
```

```
-- insert values without offset
INSERT INTO t_units_conversion_list(nonsi_unit, nonsi_name, si_unit,
               si_name, class, factor_to_si, factor_to_nonsi)
VALUES
       ('km', 'kilometre', 'm', 'metre', 'length'  ,1000.,       0.001),
       ('hm', 'hectometre', 'm', 'metre', 'length' ,100.,        0.01 ),
       ('dam', 'decametre', 'm', 'metre', 'length' ,10.,         0.1  ),
…
```

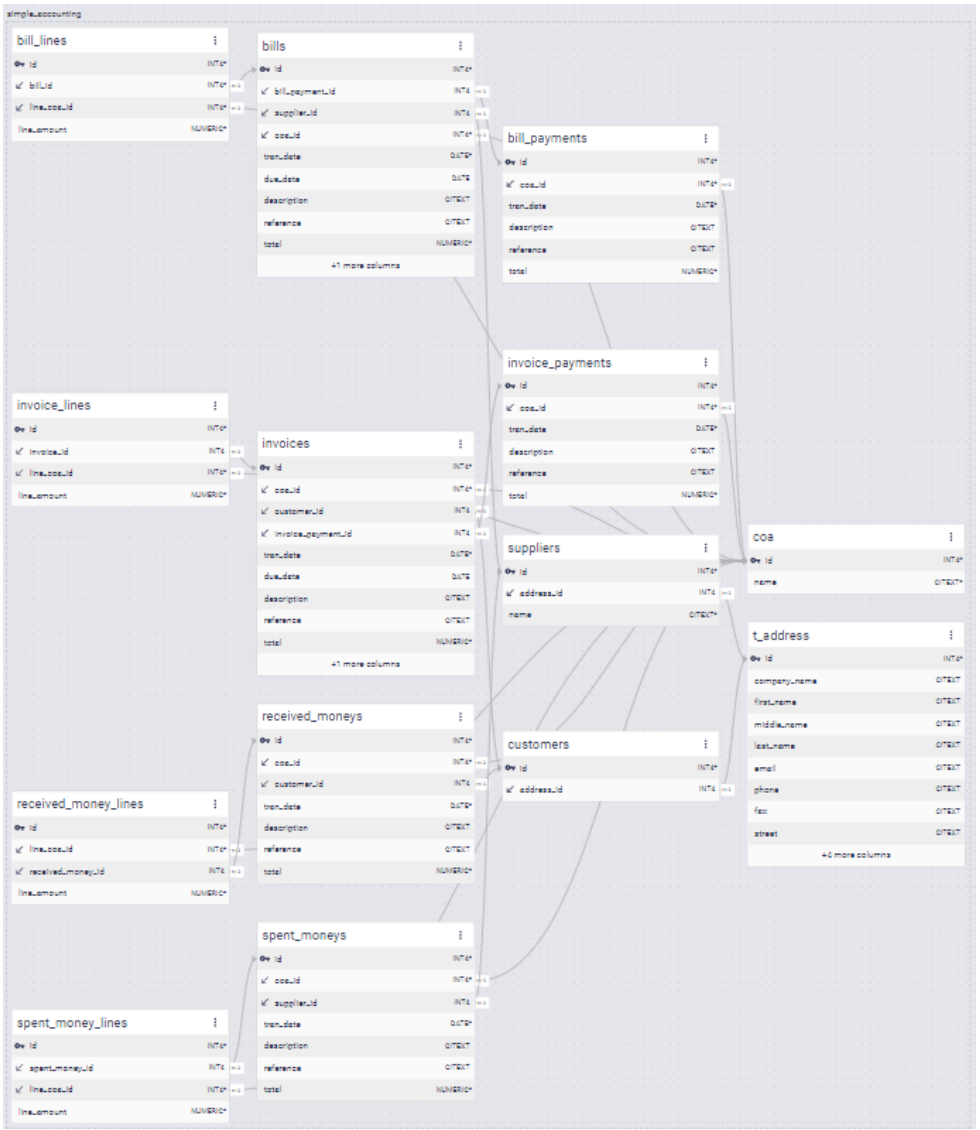
If further conversions are needed, add entries to the config tables.

Extension: simple_addresses

Purpose:

Storing addresses given ISO countries

ER model:



Description:

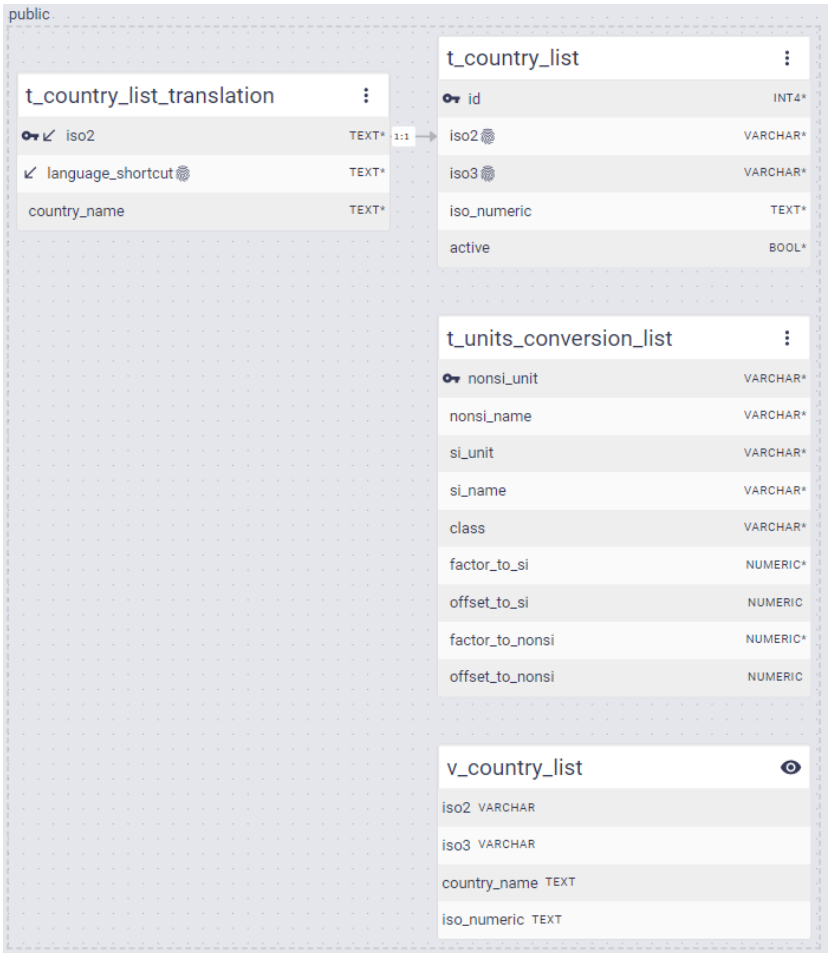
A module to handle address data. To make this module work, the CYPEX country_list extension must be installed.

Extension: country_list

Purpose:

Country lists and ISO codes

ER model:



Description:

This module provides country lists and ISO codes. All officially recognized countries are listed including various incarnations of ISO codes. It allows users to quickly fill up “drop-downs” containing country codes without having to load those lists manually. Country names are represented in English and German. However, other languages can be added easily.

Extension: basic_types types

Purpose:
Provide basic and commonly used data type abstractions

ER model:
No tables needed.

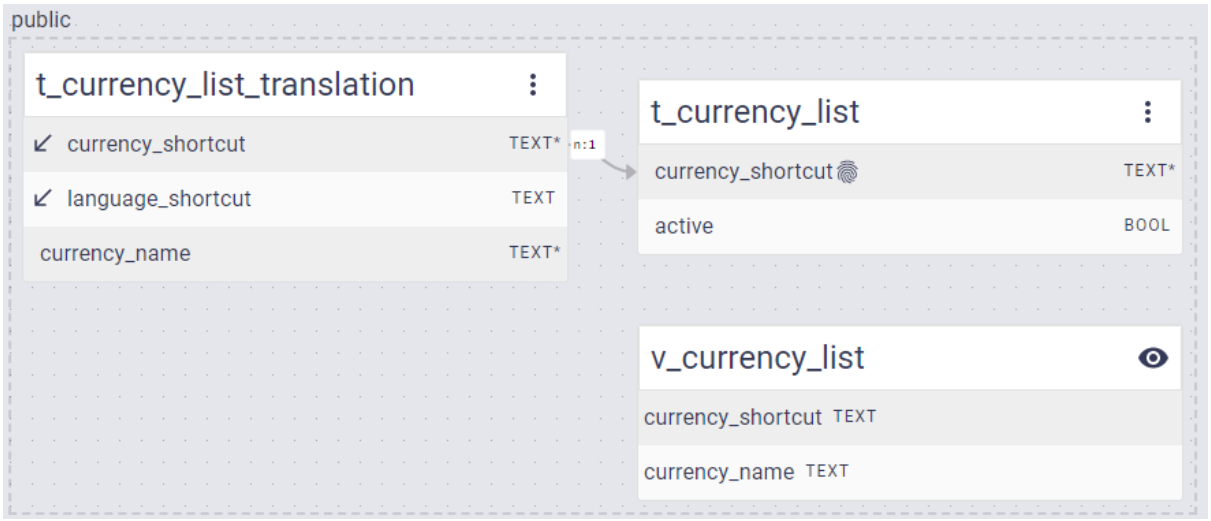
Description:
The following types are provided by the extension:

- color_code: Hex codes to store colors
  - Format examples: `#00ccff, #039, ffffcc`
- alphanumeric_string: A string which only supports ASCII characters and numbers (no blanks, etc.)
- password_text: At least 1 lowercase, 1 uppercase, 1 number, 1 special character and at least 8 characters long
- url: Matches http and https URLs.
- domain: Match domain names
- credit_card: Match card numbers
  - Amex Card
  - BCGlobal
  - Carte Blance
  - Diners Club
  - Discover Card
  - Insta Payment Card
  - JCB Card
  - Korean Local Card
  - Laser Card
  - Maestro Card
  - Mastercard
  - Solo Card
  - Switch Card
  - Union Pay Card
  - Visa Card
  - Visa Master Card
- hex_value: Hex values such as `#a3c113`
- number_positive: Positive numbers (NULL allowed, 0 allowed)
- number_negative: Negative numbers (NULL allowed, 0 allowed)
- int8_positive: Positive 8 byte integer (NULL allowed, 0 allowed)
- int8_negative: Negative 8 byte integer (NULL allowed, 0 allowed)

Extension: currency_list

Purpose:
Ready-to-use currency lists

ER model:



Description:
This extension provides a ready-to-use list of commonly used currencies (EUR, USD, GBP, CHF) which can easily be extended. v_currency_list provides a list of those currencies given your default CYPEX language determined by `cypex.current_language()`. The default language of CYPEX can be changed in the config table of CYPEX.

Extension: interest_rates

Purpose:
Basic functions to calculate loan-related information

ER model:
No tables needed

Description:
The following function is provided to calculate monthly payments:

```
CREATE OR REPLACE FUNCTION loan_calculate_rate(
      v_sum           numeric,
      v_interest_rate       numeric,
      v_months        int
)
RETURNS numeric …
```

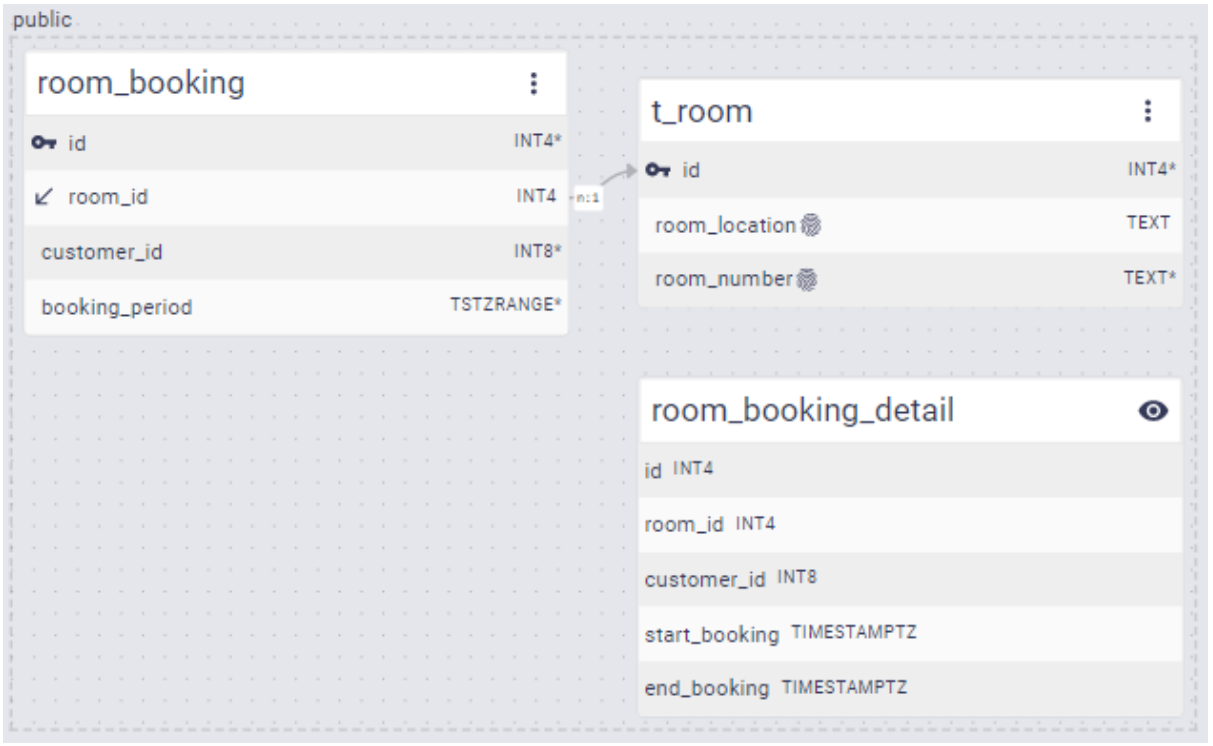
Here is a sample:

```
SELECT interest_rates.loan_calculate_rate(1000.0, 5.0, 12);
 loan_calculate_rate
---------------------
             85.6075
(1 row)
```

Extension: room_booking

Purpose:
Basic functions to manage hotel room bookings

ER model:



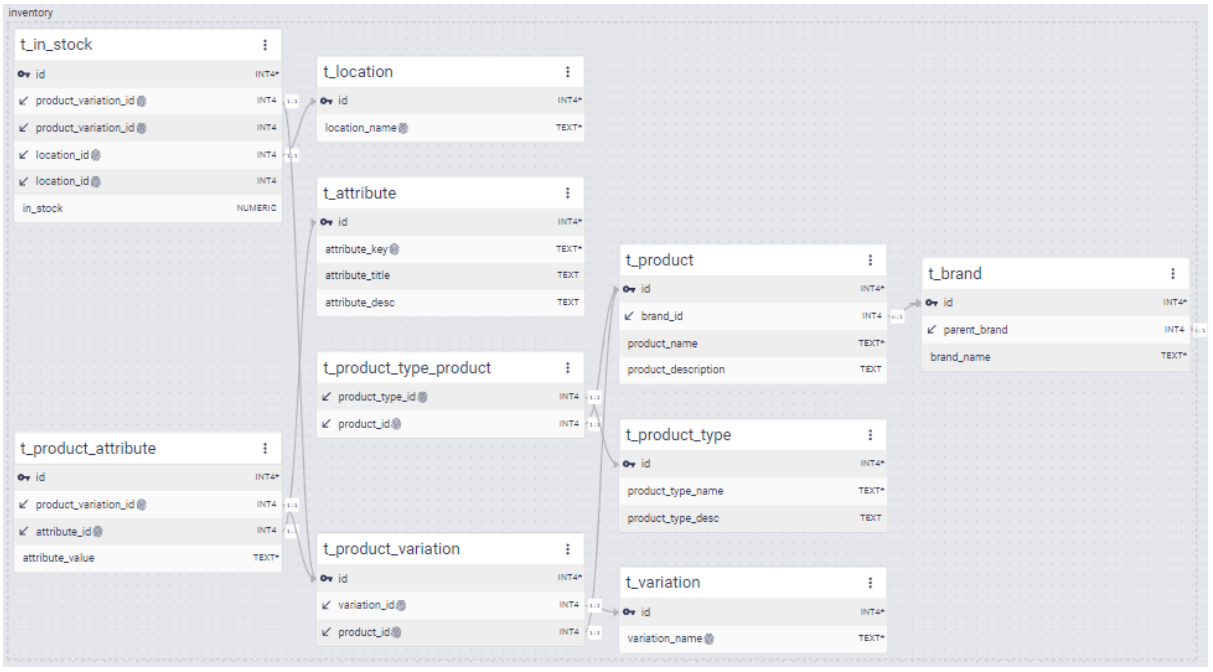
Description:
The model describes a basic hotel including bookings. It's designed as a starting point for more comprehensive models.

Extension: inventory

Purpose:

Basic functions to manage inventory

ER model:



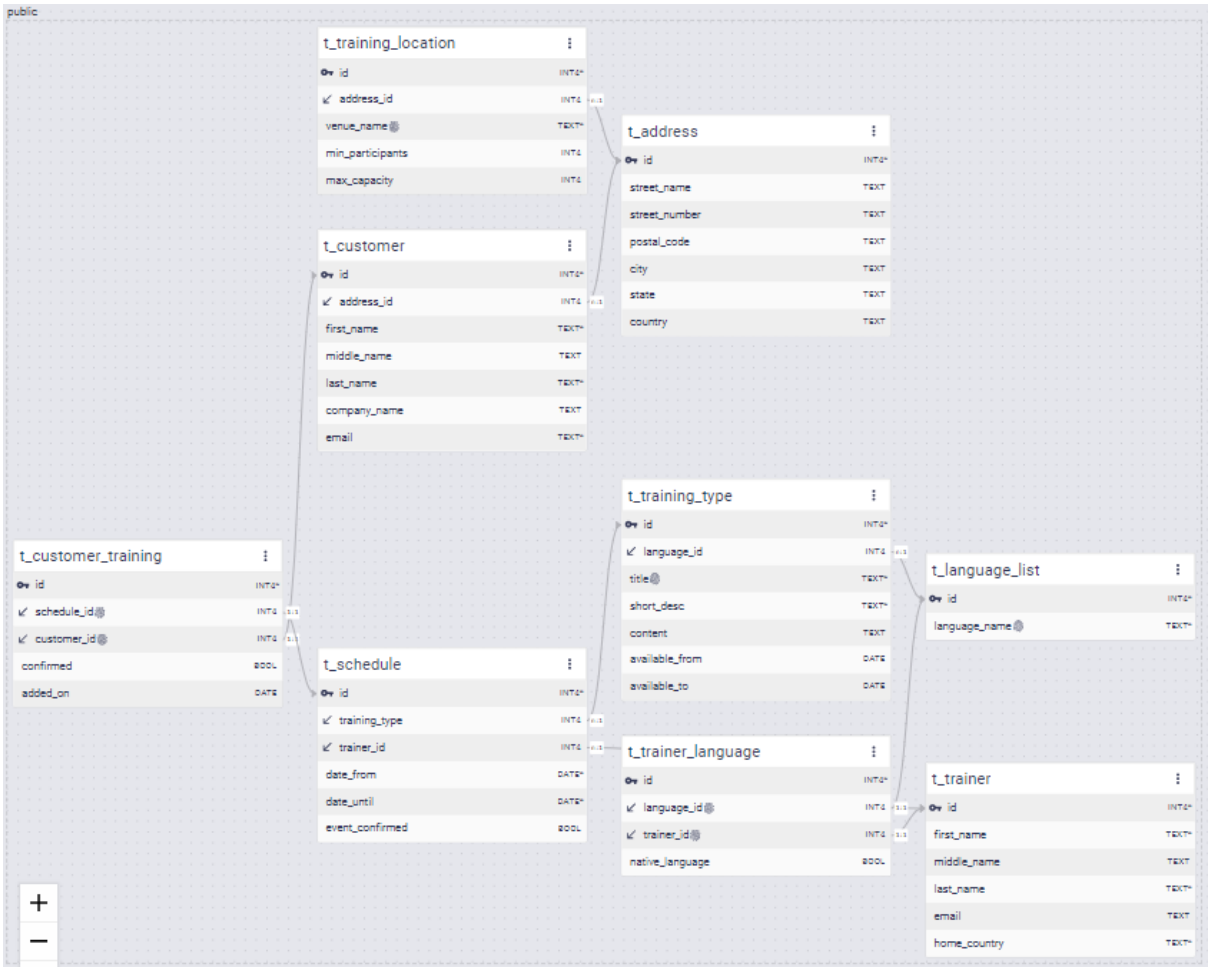
Description:

The inventory model describes brands, products, product types as well as inventory in an easy-to-use way. It allows users to various products and provides generic ways to handle product attributes.

Extension: training_courses

Purpose:
Manage trainers, training courses as well as customers.

ER model:



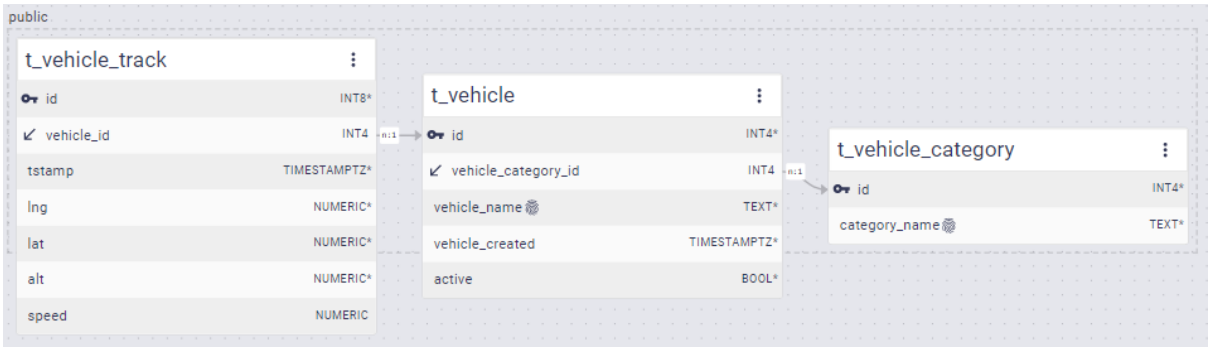
Description:
This model handles training related issues. Trainers can speak various languages and are assigned to different types of training. Training can take place in different locations, which are assigned to customers. Again this is a blueprint to develop things into more complex models.

Extension: gps_tracking

Purpose:

Manage GPS tracks

ER model:



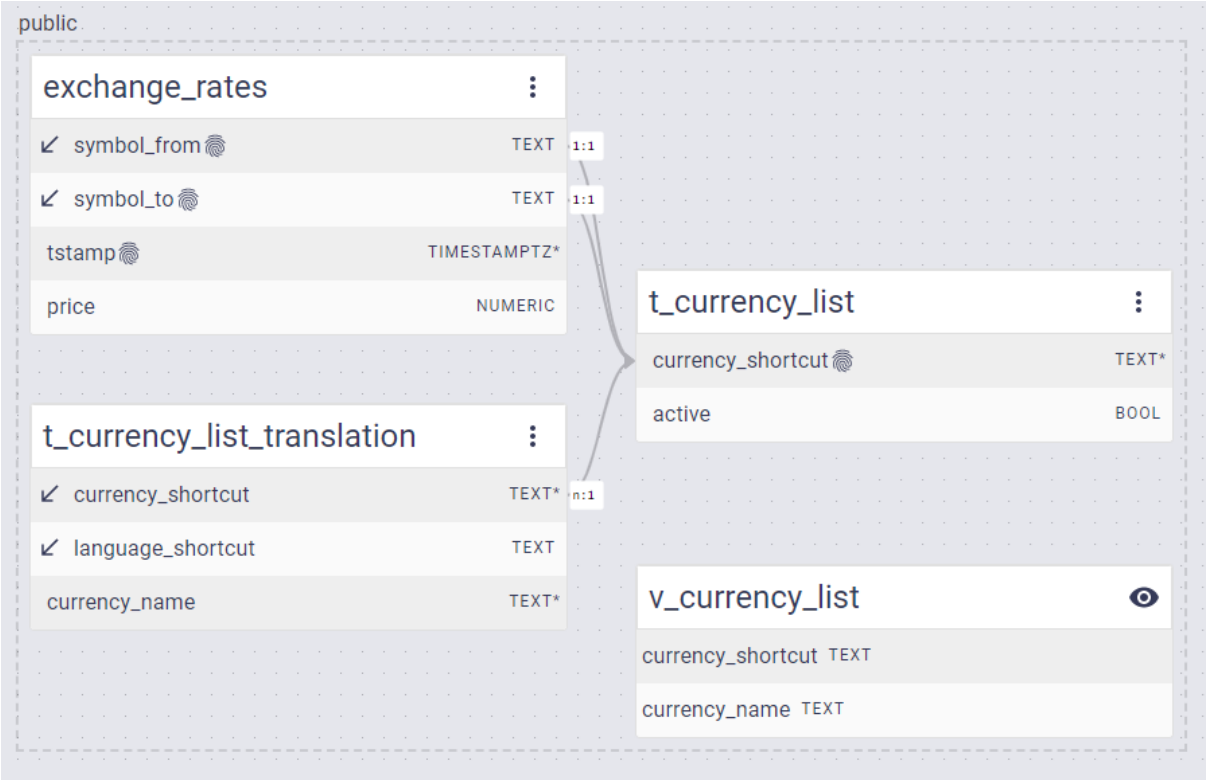
Description:

Used to store vehicles (which can be a special category of vehicles). Those vehicles are then assigned to GPS tracks. CYPEX can then visualize those tracks using GeoJSON documents.

Extension: exchange_rates

Purpose:
Handle exchange rates

ER model:

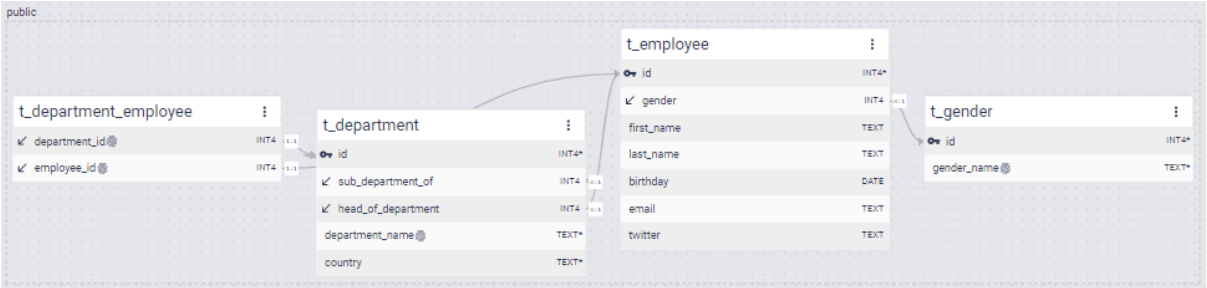


Description:
This module offers an easy way to store currencies as well as exchange rates. Currency names can be translated to ensure multi-language support. The price is stored for any point in time.
The currency_list extension is required for this module.

Extension: team_list

Purpose:
Manage team lists

ER model:



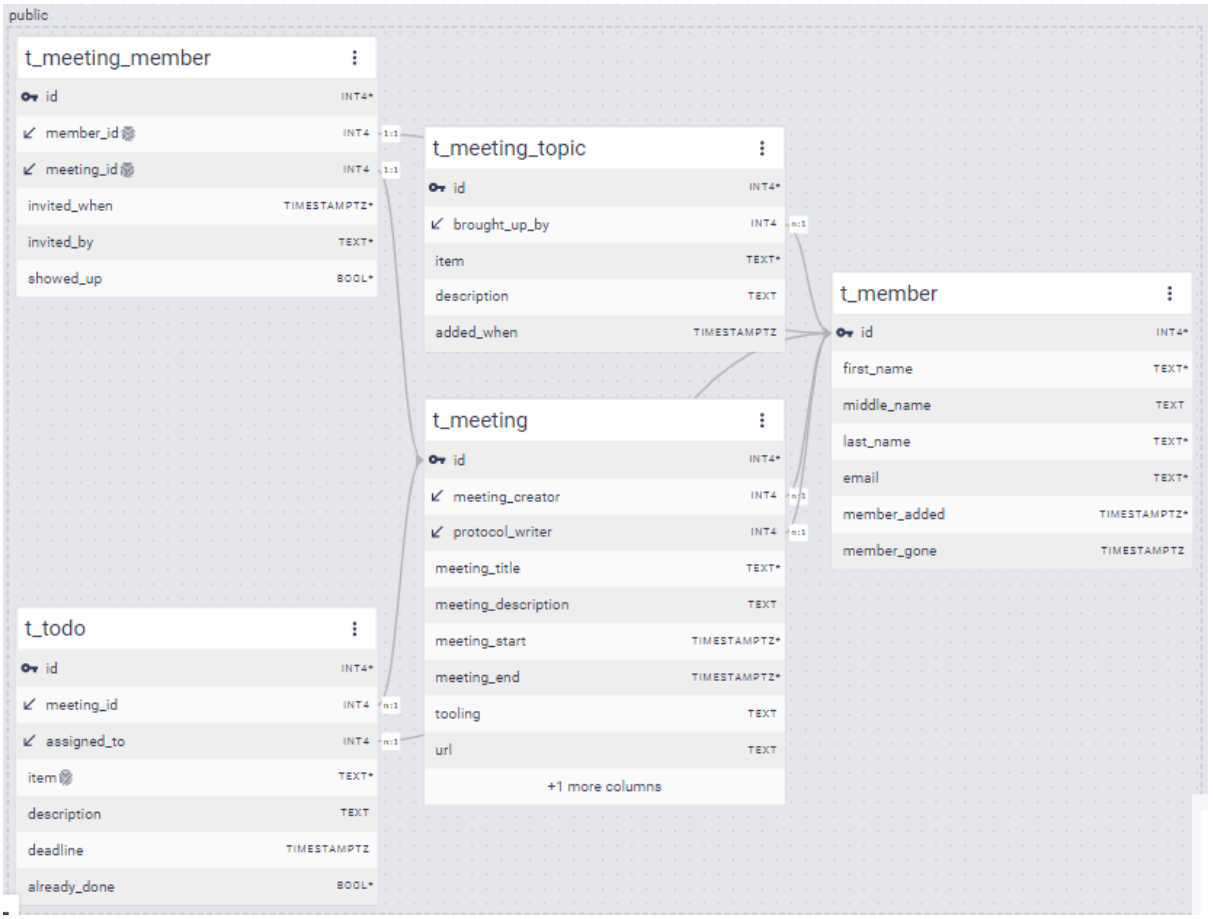
Description:
The idea of this module is to give users the ability to store team lists. Employees are assigned to a list of departments. Column lists can easily be extended.

Extension: jour_fix

Purpose:

Handle TODO items and jour fix meetings

ER model:



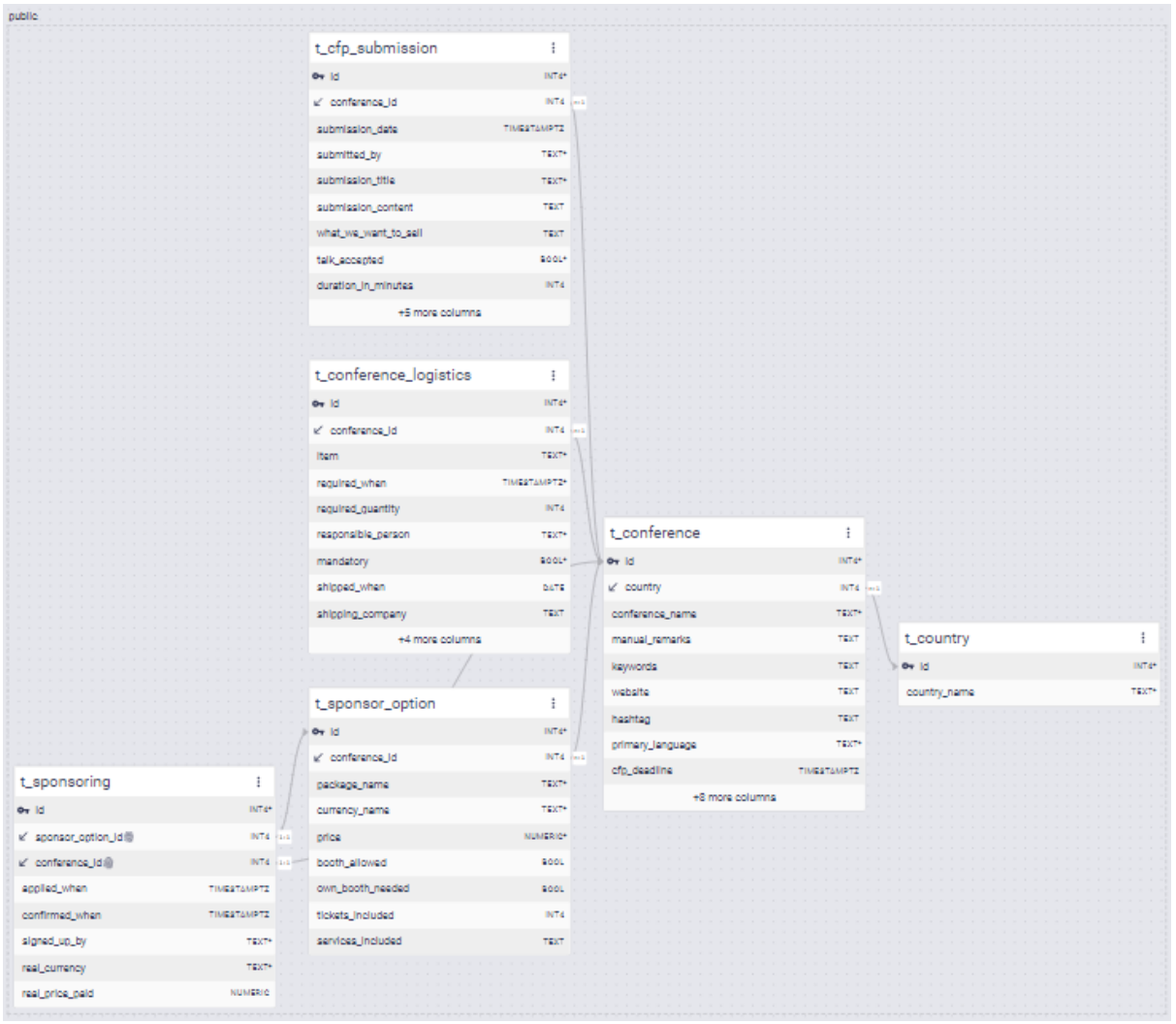
Description:

The core idea is to give users the ability to handle TODO items coming out of team meetings related to many different topics.

Extension: conference_sponsoring

Purpose:
Manage sponsors and logistics for a conference

ER model:

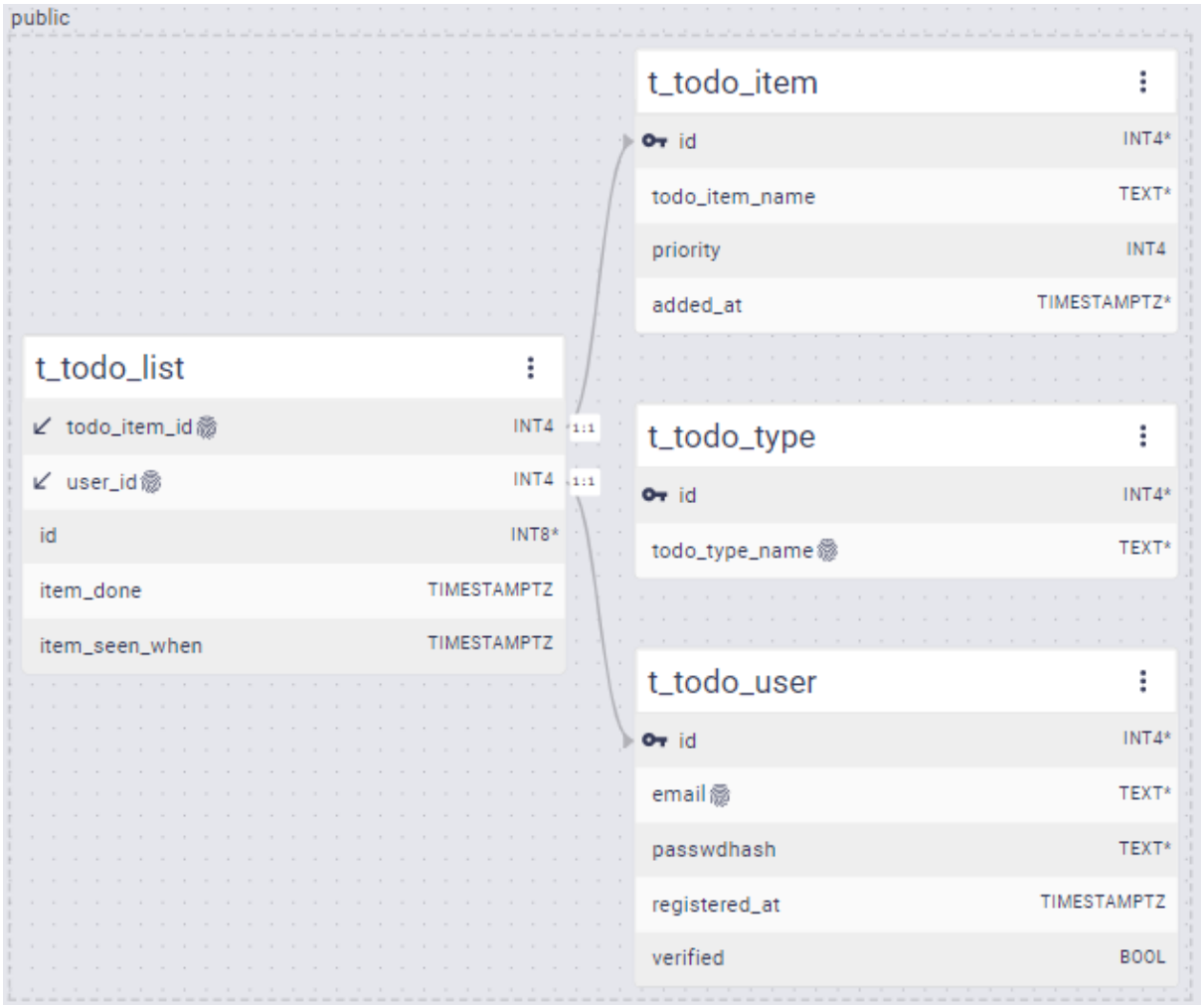


Description:
Conferences can be challenging. This is especially relevant in case it's necessary to coordinate sponsoring as well as conference logistics. This model handles conference sponsorship-related tasks and helps to store information about conference logistics. Which items have been sent to which conference? What is the tracking data? etc.

Extension: todo_simple

Purpose:
Manage simple TODO items.

ER model:

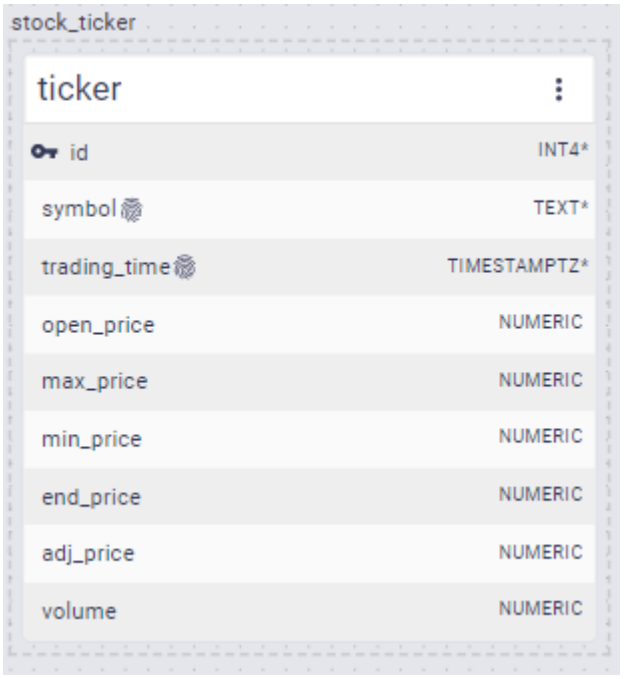


Description:
TODO items are assigned to TODO types as well as to users who are supposed to handle those items. It's a simple yet efficient model to store tasks.

Extension: stock_ticker

Purpose:
Manage stock prices

ER model:



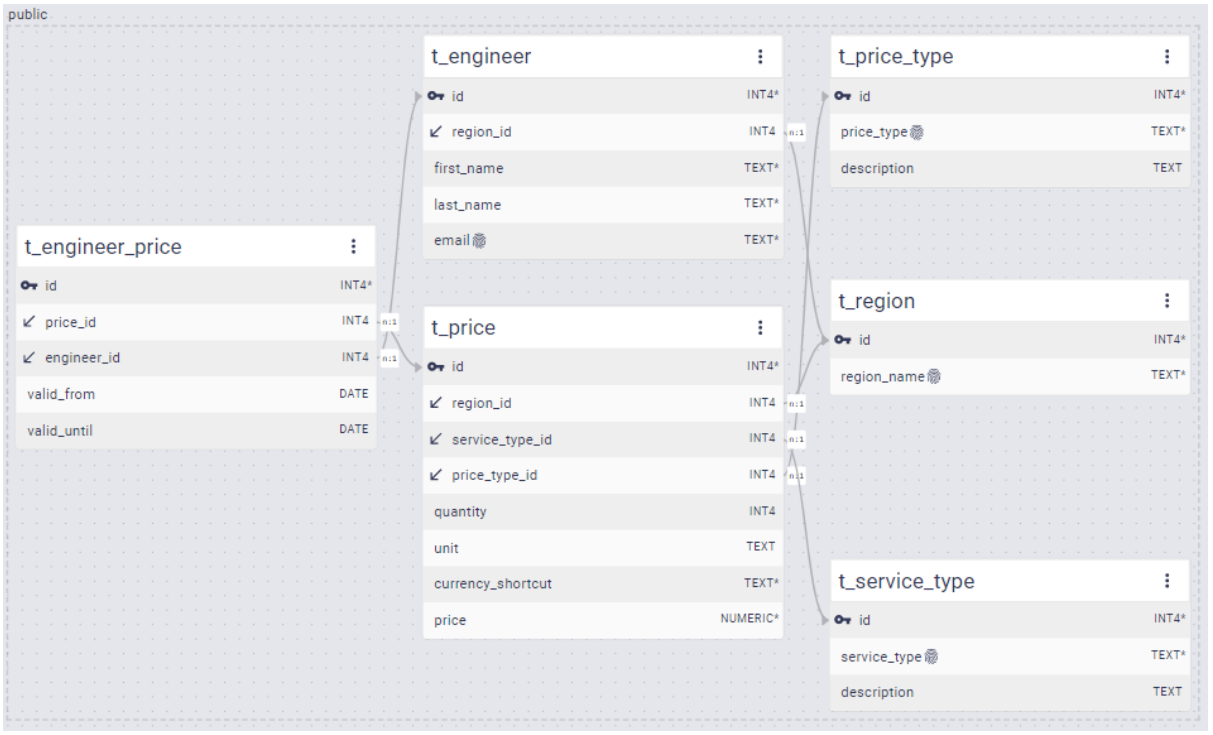
Description:
The stock ticker module has been modeled after the Yahoo Finance API. It can be used directly to store data coming from this API.

Extension: consulting_prices

Purpose:

Manage prices for engineers, depending on the region

ER model:



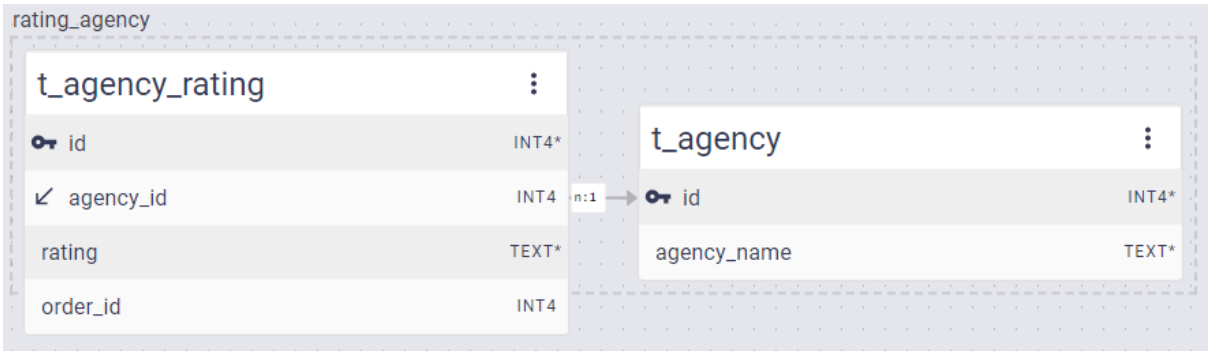
Description:

Often prices depend on regions, type of service and so on. The consulting_prices extension contains an ER-model which reflects those aspects of pricing and allows you to store prices depending on service types and region.

Extension: rating_agency

Purpose:
A basic model to handle rating agencies

ER model:



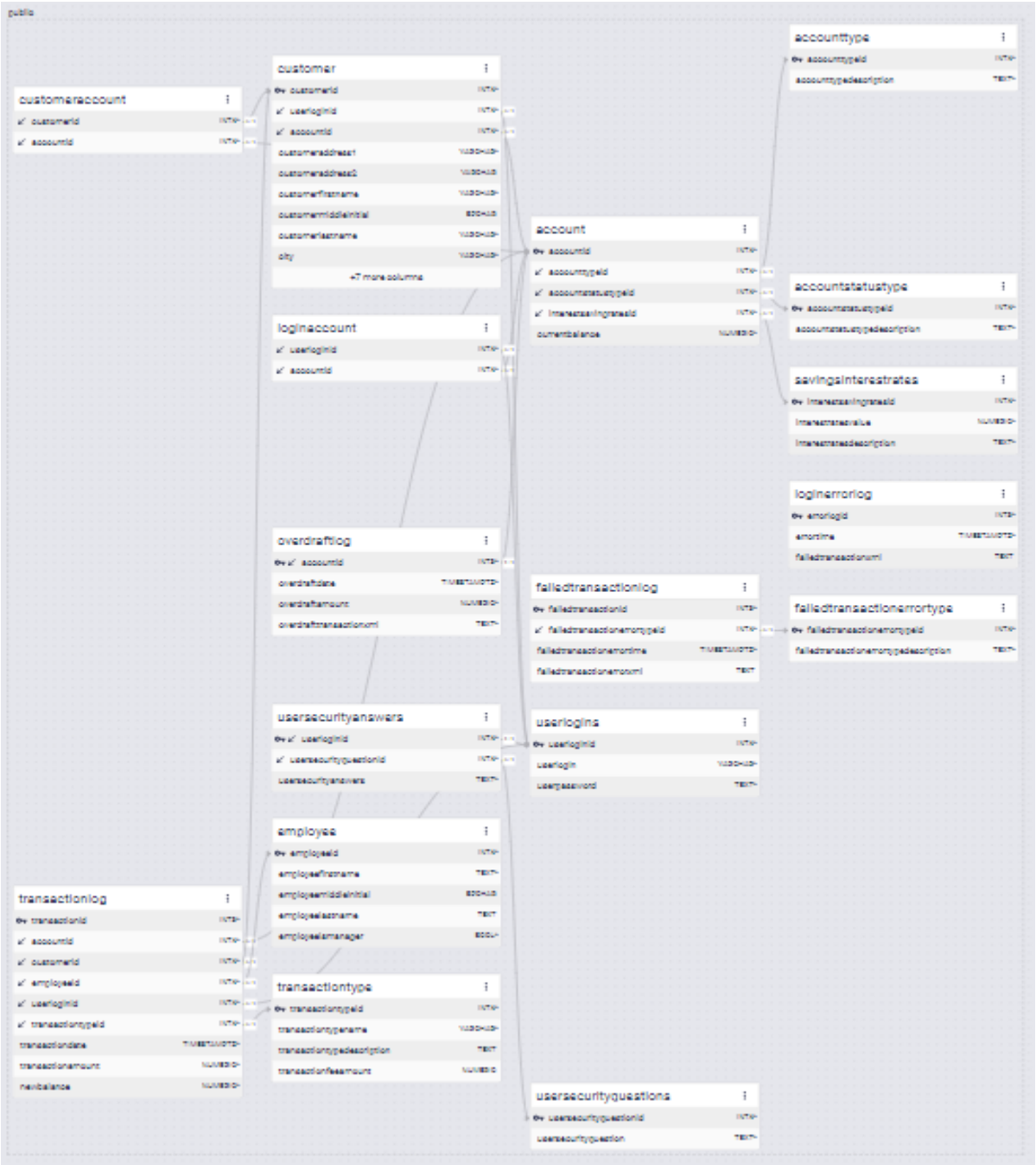
Description:
A basic data model capable of storing information about rating agencies.

Extension: bank_account

Purpose:

Store bank accounts

ER model:



Description:
This model is a comprehensive module which is capable of managing bank accounts as well as many aspects of infrastructure. It can handle:

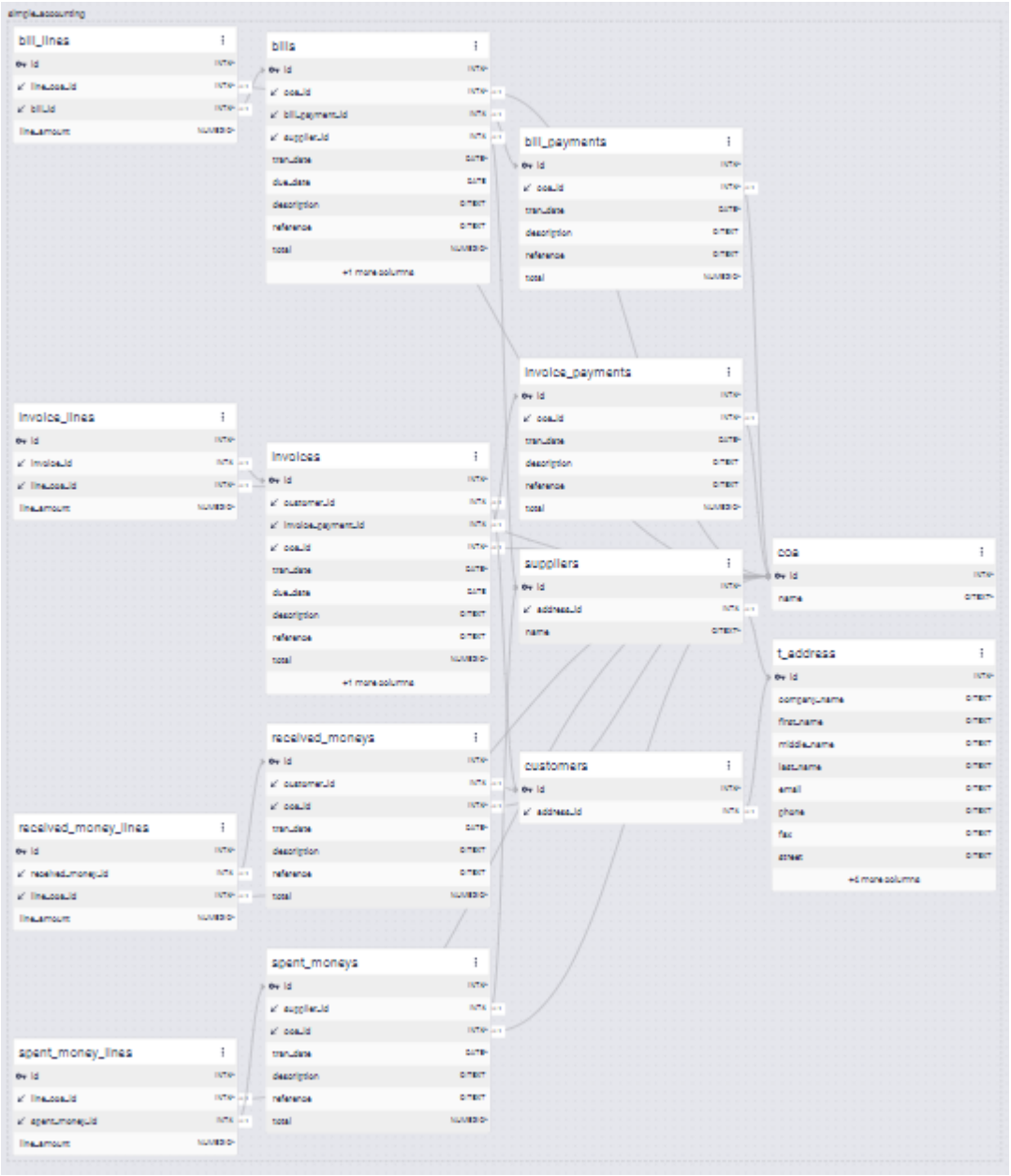
- User logins
- Security questions
- Account types
- Interest rates
- Account status
- Failed transactions
- Error logs
- Employees
- Transaction types
- Transactions
- Accounts
- Customers

Sample data is available.

Extension: simple_accounting

Purpose:
Accounting and invoicing

ER model:



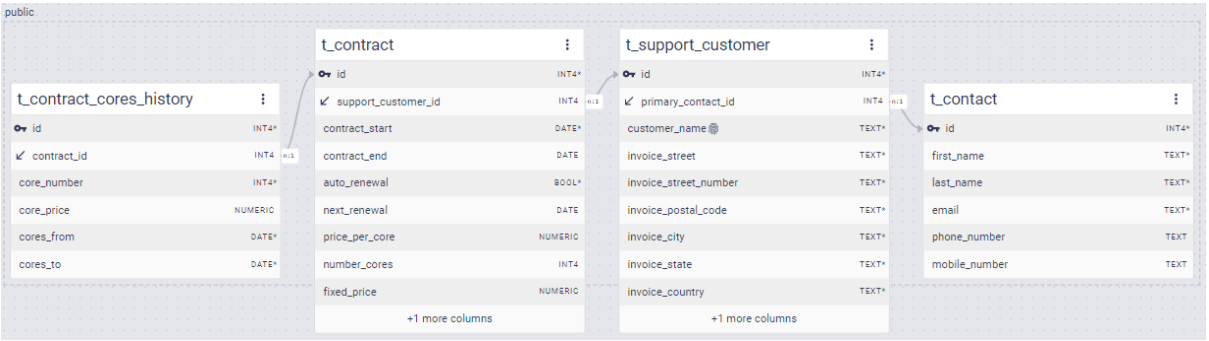
Description:
This model contains a simple bookkeeping infrastructure which consists of addresses, customers, invoice payments, invoices as well as invoice components ("lines"). The presence of the PostgreSQL contrib package is needed to satisfy the dependency on the citext extension (= "case insensitive text").

Extension: support_customer

Purpose:

Managing support customers

ER model:



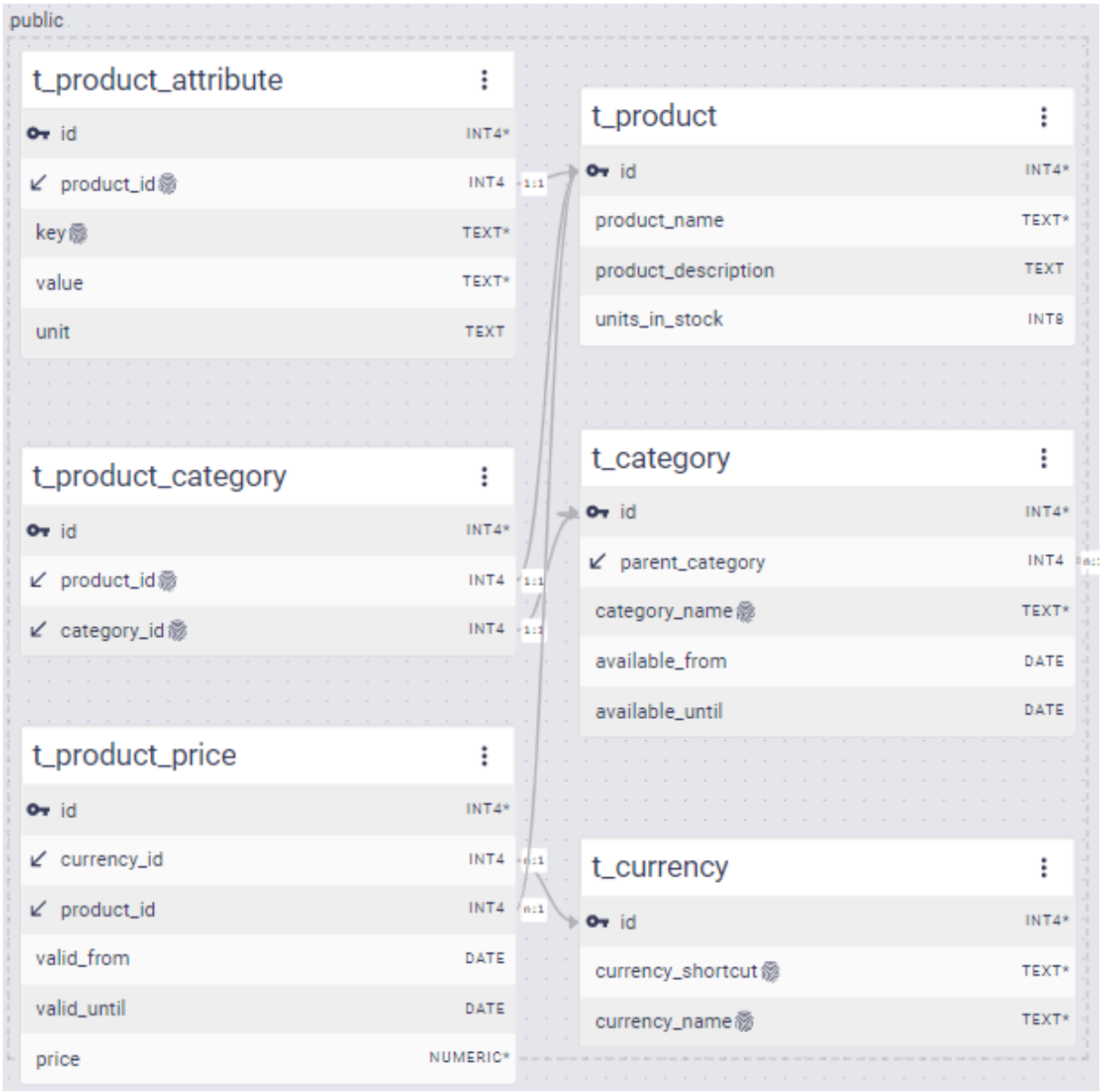
Description:

This module offers users the capability to handle support customers and contracts.

Extension: products_simple

Purpose:
Storing products and product categories

ER model:



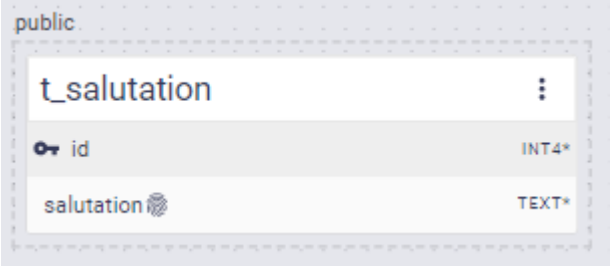
Description:

Products can have various categories and can be assigned to attributes. Prices can be in varying currencies and can be valid for different periods of time.

Extension: salutations

Purpose:
Ready-made salutations

ER model:

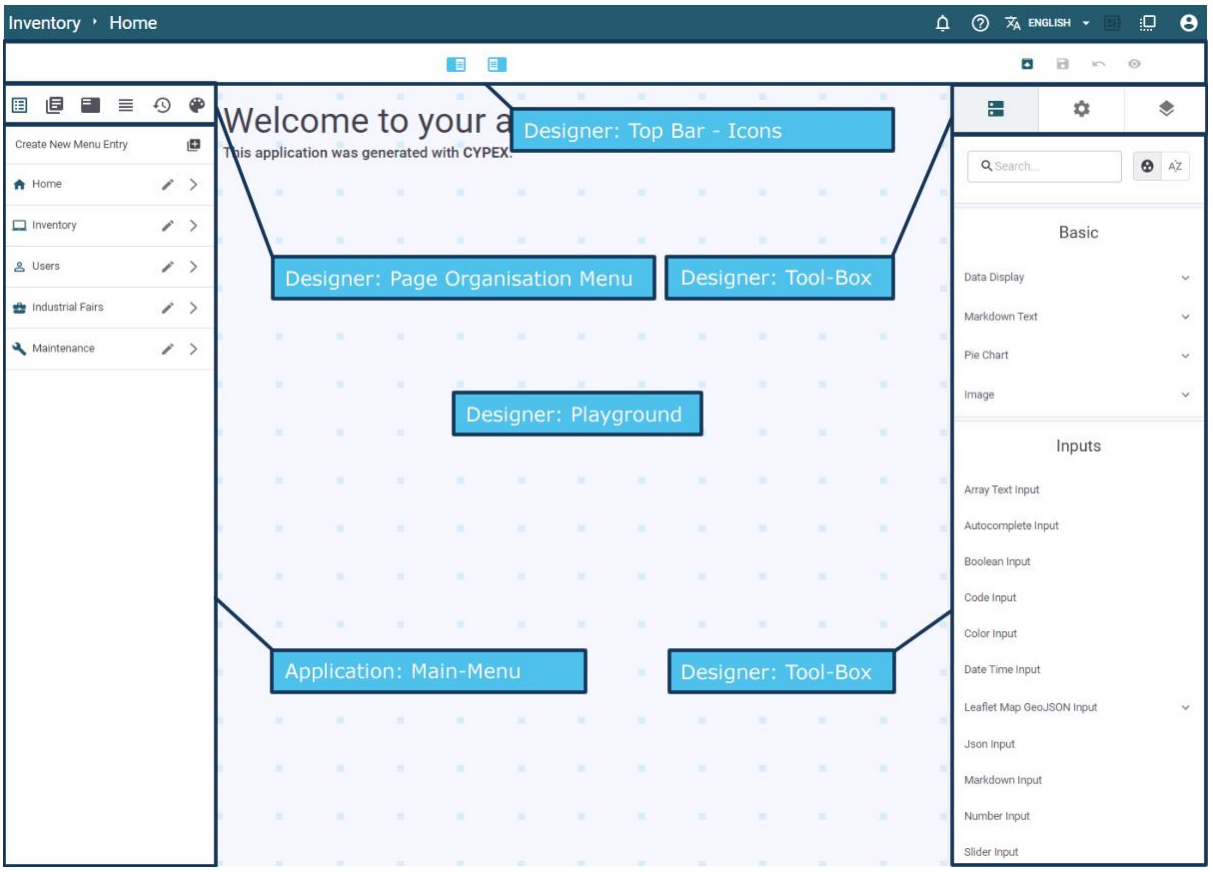


Description:

The “salutations” module will provide a list of ready-to-use salutations (e.g. “Mr”, “Mrs”, etc.). It helps to reduce the effort to store addresses and other person-related data.

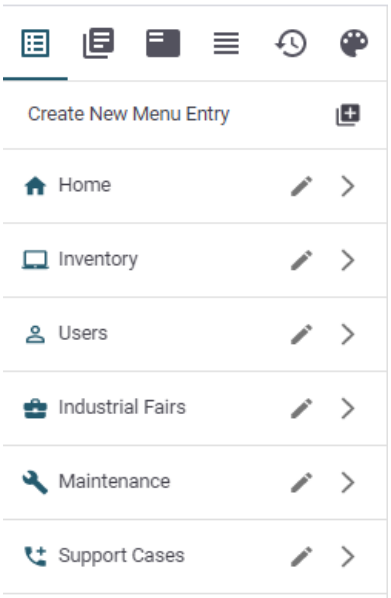
Application Designer

The CYPEX Application Designer is a low-code development platform.
The designer provides a wealth of functionality which can be used efficiently to quickly and easily build your application .



Let's discuss these elements in more detail and see how they can be used.

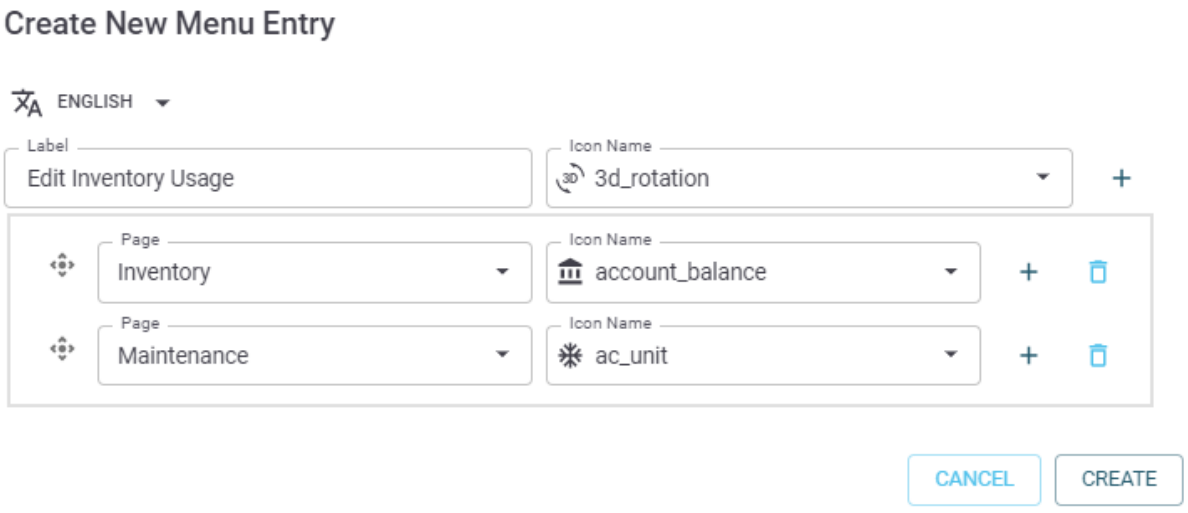
Section: Main Menu - Menu Entries



The first element is the “menu editor”. It allows you to modify various aspects of the application. It's of vital importance. It allows you to adjust your menu items. You can also handle software revisions. In this section, we will guide you through these features and explain step-by-step what can be done and which purpose these features serve to create even better apps.

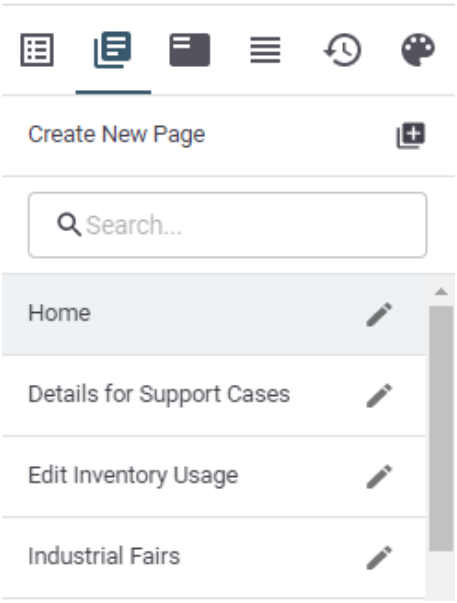
Create New Menu Entry

The first thing to understand is how to create new menu entries. Note that the menu is highly dynamic. The default renderer will create one page per query. However, this might not be your desired layout. You can modify the layout by adding entries and assigning icons to those entries as shown below:



Note that there is drag-&-drop functionality in place, which allows you to flexibly adjust the order of pages in the menu. You can move them around and change them according to your needs.

Section: Main Menu - Pages



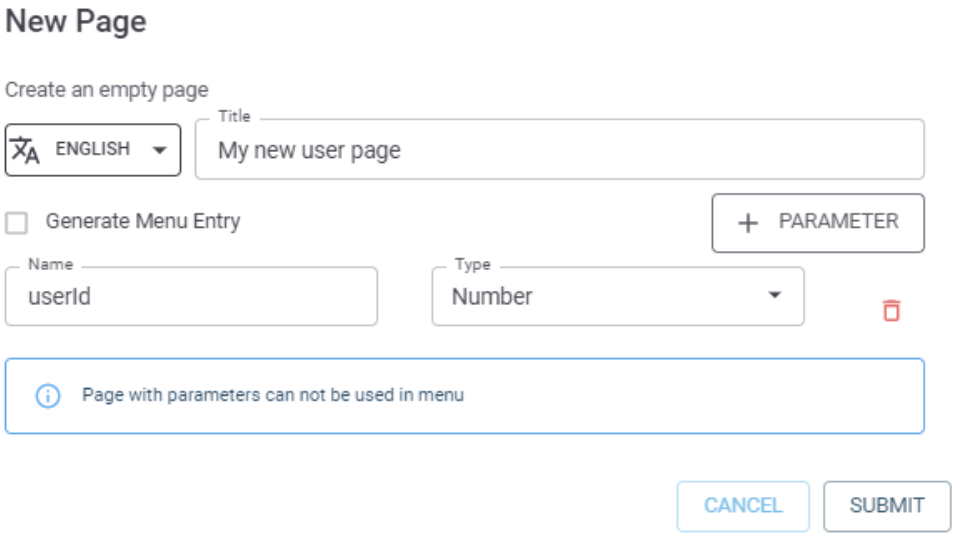
It's also possible to create completely new, empty pages. The second icon will help you to achieve exactly that. Again, this is fully customizable.

Note that an alternative to completely new pages is to use “incremental rendering”. The idea is that you don’t have to start from scratch.

While new pages are ideal for dashboards, completely empty pages can be more work in case you want to build forms. In those cases, incremental rendering presents a time-saving alternative to a fully manual process.

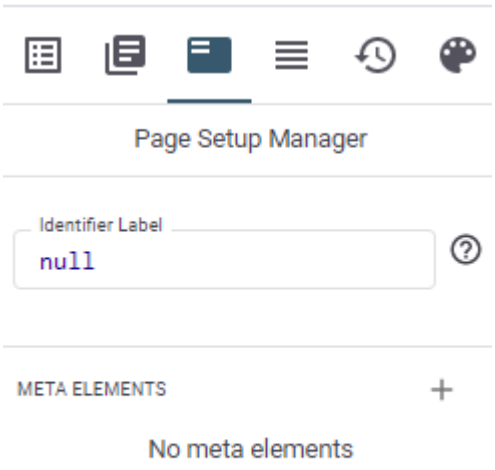
Create New Page

Creating a new page is easy, as shown in the next listing:



Note the idea of passing parameters to the page. Why does it matter? Suppose you want to have a page that shows all there is to know about a certain product. CYPEX needs to know which product you’re talking about; a parameter is needed to provide this info to the page. Passing parameters is a common process which is highly relevant to most applications.

Section: Main Menu - Current page



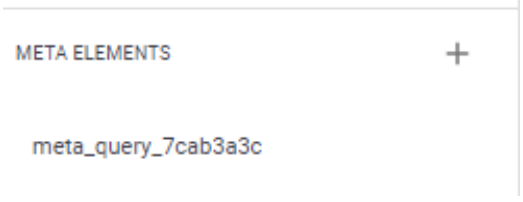
Managing the title of a page is done through the “Identifier Label”. Often the title of a page has to be dynamic. Perhaps you would like to add the name of a product to the title or perhaps you want to calculate some other value, and use it as a page title. The “Page Setup Manager” allows you to do exactly that.

If you want to have a specific title in the header of the application (App Bar), you can use the input “Custom Expression". This is often used for pages which have an “Edit form”.
If you set the "Identifier Label" to `location.queries.identifier`, the app bar will contain a title such as: "Inventory > Edit Inventory > 1" where "1" is a return value of `location.queries.identifier`.

In general, you can use any custom JavaScript expression here. However, it's also possible to use static text containing the desired value.

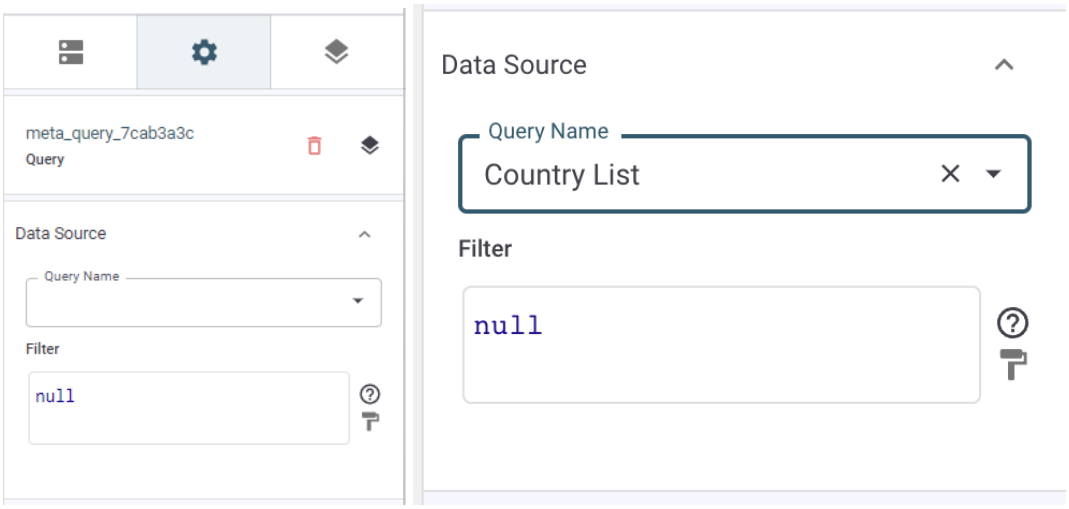
MetaElements: Hidden fields

A page might consist of more than meets the eye. Often a data source is needed to calculate the behavior of a page. Maybe you only want to display a table containing notifications in case some threshold has been reached. Or maybe you want to control the color of a text field depending on some numbers coming from various data sources.



In those cases you'll need MetaElements which are basically hidden data sources (“hidden fields”). MetaElements are elements that don’t have a position on the page but can be used to fetch data. This is especially important if you are using custom expressions.

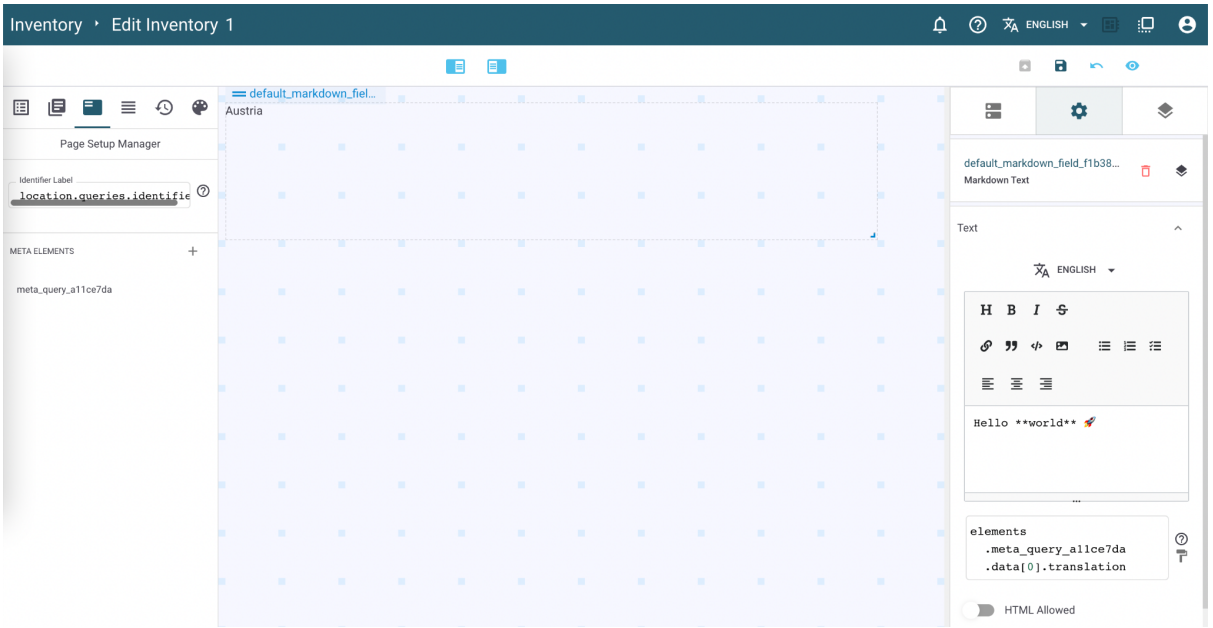
Select a data source and optionally set a filter:



The filter might be of key importance because it allows you to expose only a subset of data which can increase performance and make things more secure.

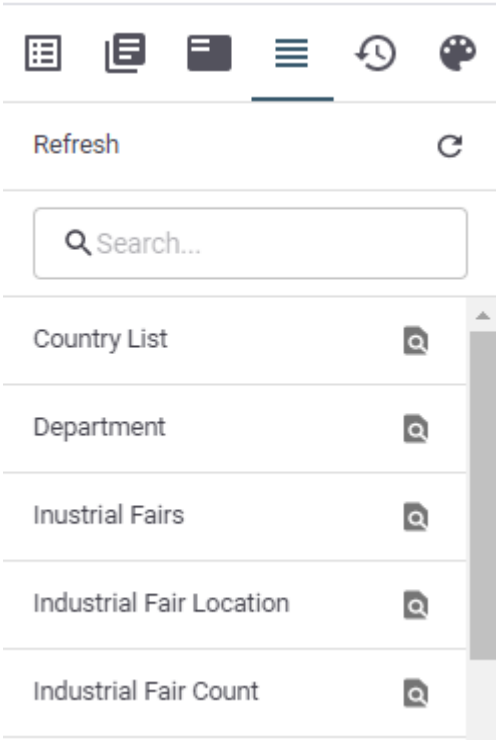
Once the customer data source has been created, you can use this MetaElement inside a custom expression. Note that the data source is otherwise not visible on the page - all we have done is to make the page aware of this data source.

The following screenshot shows how:



Section: Main Menu - Queries

Queries represent the data sources that actually field your page. In CYPEX every defined in the model builder is usable in the GUI to provide data for some graphical element. Of course this is only true in case you have enough permissions.

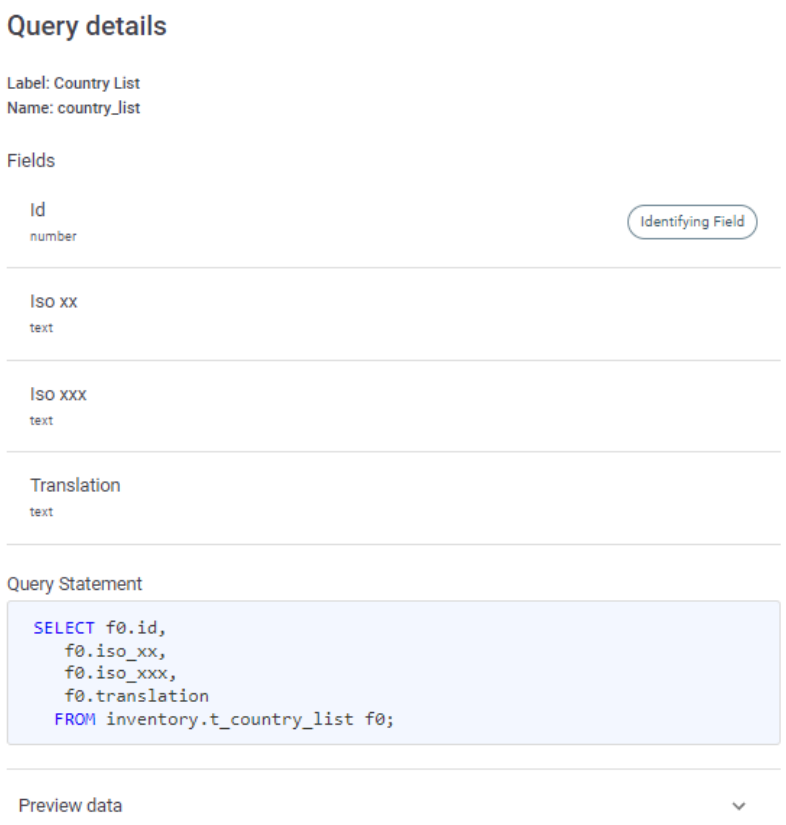


The point is: Without queries it's impossible to display anything on your screen. To avoid constant switching between the admin panel and the graphical editor you can get an overview of your queries and their definition in a dedicated menu.

If you click on the symbol in the right hand side of the query the tool will open a visual representation of your query, so that you can see how it has been defined and what data it contains.

Query Details

The query details might look as follows:



Note that you'll also see which (if any) field is used as an identification field. Those fields are a kind of primary key for your query and help you to identify the rows you need, quickly and easily. Many graphical elements such as form need those identification fields to ensure that the correct row is updated in case a change is made to the data.

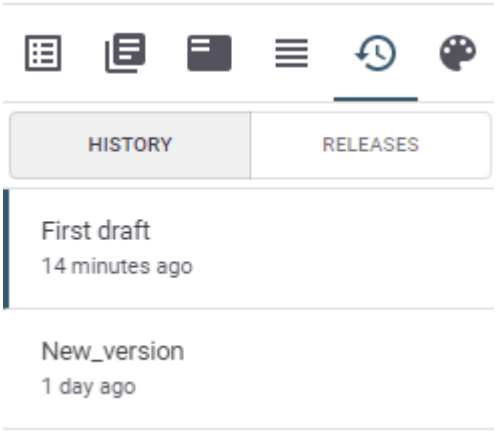
Section: Main Menu - History

In this section:

- History
- Releases

Why do they matter? Every change made to a CYPEX app isn't lost - all changes will be tracked and you can always return to a prior state. However, not all prior states are created equal:

History



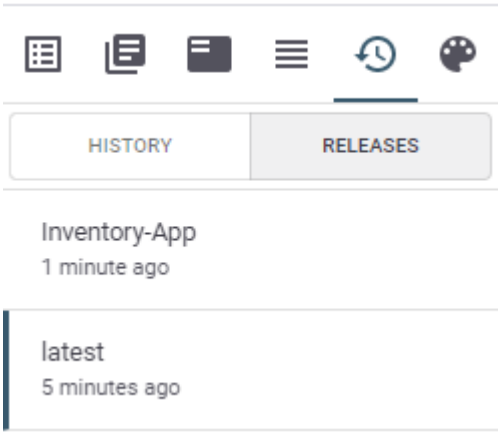
First, let's talk about the history tab: It allows you to see all prior versions. You can go back to any previous version by clicking on it and by following the instructions.

Note that if you go back in time, changes that happened later will be lost forever. You need to keep that in mind to avoid destroying valuable work.

Here is what it will look like if you attempt to go back in time to get rid of undesired changes that happened later (e.g. new bugs, wrong approaches, etc.):

Releases

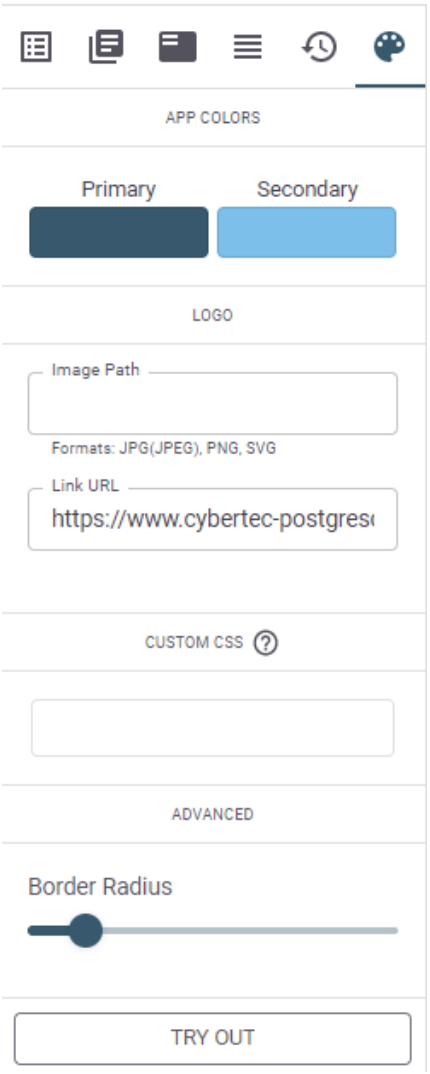
Releases are an important part of CYPEX. You can turn the current version into a release, which tells the system that it is now dealing with a stable, production-ready release of the software:



Releases can be tested and then published for end users. Releases are therefore stable versions of your software:

Section: Main Menu - Style

The next important feature in the life of an application is its styling which will provide end users with the desired look and feel, and ensure that CI (Corporate identity) is maintained.



The last tab will guide you through the styling process. Select your desired colors and use the logos of your choice to ensure a consistent look across all your applications.

The logo is usually required by the customer. However, there are many more options, including custom CSS.

Custom CSS

The default layout is suitable for many applications. However, often it's necessary to support custom CSS. The way this works is as follows: Elements can be selected via special HTML attributes, namely `data-cypex-element-id` and `data-cypex-element-type`.

Here is an example:

```
[data-cypex-element-type="default_table"] tr td:first-child {
   background: lightblue;
}

[data-cypex-element-type="default_markdown_field"] {
   color: "#777",
}
```

```
[data-cypex-element-type="default_internal_link_button"]
button:hover {
   filter: contrast(2.5),
}
```

Of course there are limits to what is feasible. However, the most common changes are perfectly feasible and supported by the tooling.

Section: Top bar

Now let's focus on the top bar and see what can be done there:

Section: Top bar - Icons

Let's tackle the icons bar at the top first. There are a couple of icons there which are relevant to the end user:

In the first line, the “bell” symbol will inform you about pending notifications. CYPEX notifications are stored in a table. In case a notification is marked as unread, the bell symbol will light up and notify end users.

The question mark symbol will display general information about the application you’re using. For example, you can see which release you’re using.

Changing the desired language can be done with the next icon in the list. By default, the language of choice is English. However, additional languages can easily be added as needed.

Finally, switch to the admin panel and exit the edit mode of the WYSIWYG editor.

Let's now focus attention on the second list of icons: The blue icons in the middle of the screen allow you to hide the panel on the left as well as the panel on the right side. This is necessary to figure out what the application looks like.

The next button (currently in gray) allows you to quickly create a release. Usually you save changes and then turn them into releases. However, you can also go the direct route and create releases more quickly. This is especially important in case hotfixes have to be deployed.

The “disk button” saves the current changes but doesn’t create a new release. It should be used to save incremental changes which aren't supposed to make it to the end user directly.

Finally, you can revert your latest changes, and preview changes without leaving the edit-mode.

Section: Tool box

The Tool-Box section is the most important section in the application designer. All available design elements are listed, and can be configured easily using our graphical interface. All items offer drag & drop support and can be flexibly placed in the grid.

Basically, the toolbox section consists of three parts which allow us to compile a user interface efficiently:

Design elements

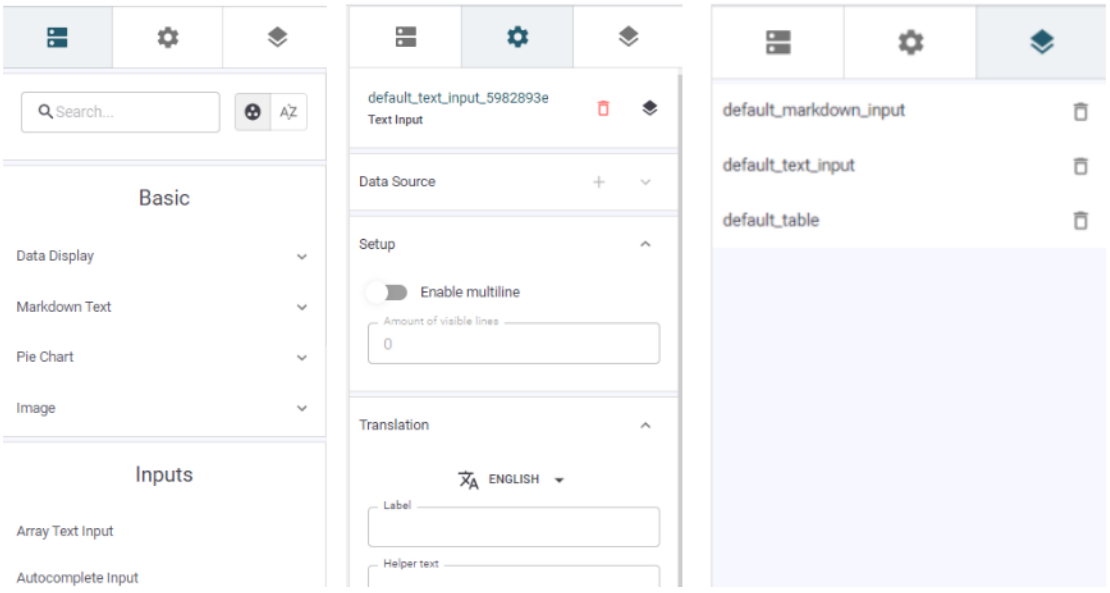
A list of all GUI elements currently available in CYPEX. There is everything from input elements to graphs and GIS elements.

Design element attributes

In case a design element is chosen, users are able to configure it. Depending on the type of GUI element you'll see different configuration parameters. What unifies those items is that data usually comes from a data source (“query”).

Copied design elements

The editor allows you to copy as simple as well as complex, nested elements. As objects can be nested it can be quite hard from time to time to understand what is actually copied and what isn't. The “copied design elements” overview allows you to more quickly gain an overview of what is copied and what can be inserted at some other place in your application.

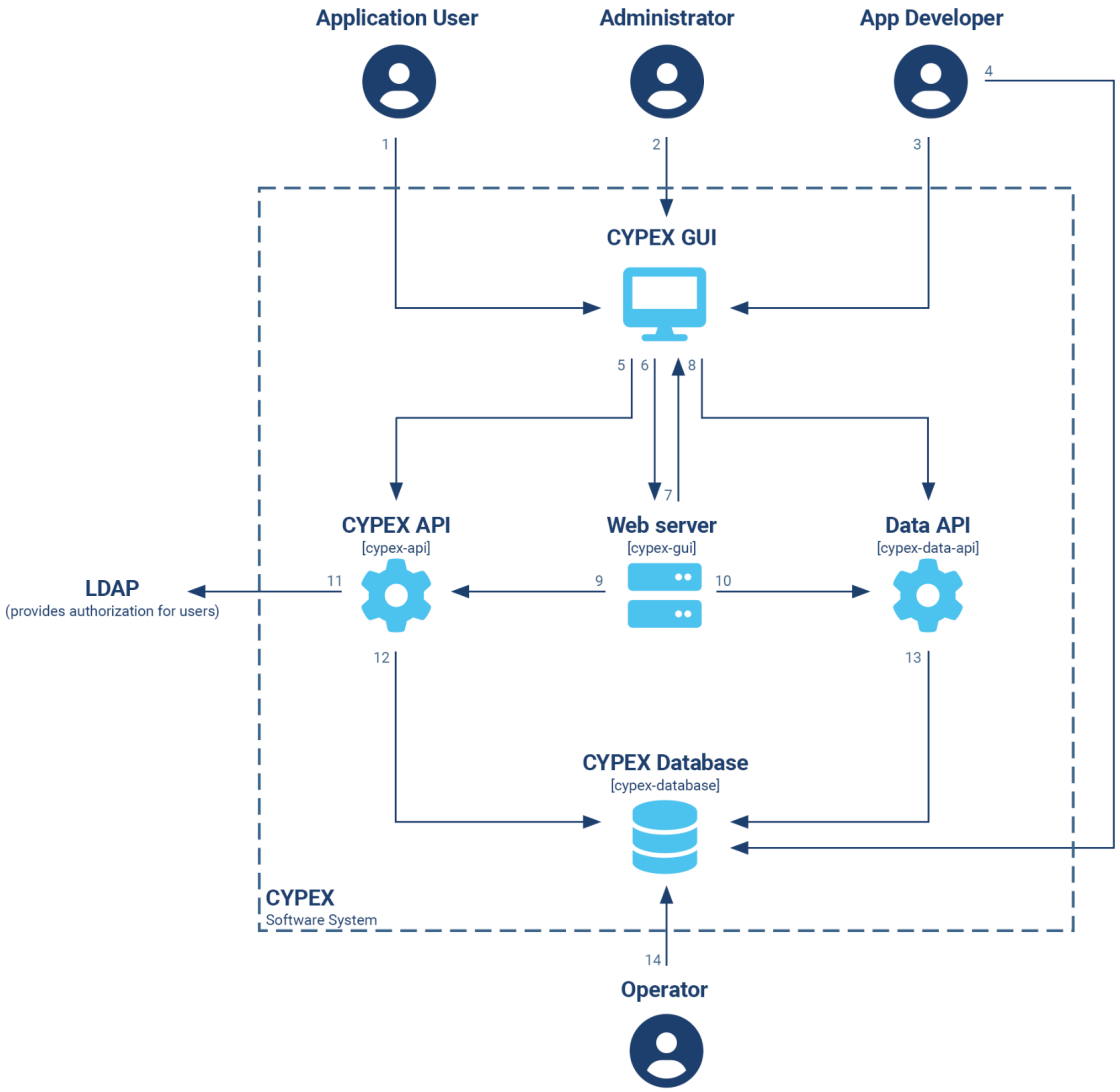


CYPEX internals

In this section, you'll be guided through the internals of CYPEX. You'll get to know the basic architecture of the solution and gain some insights into how things work. It helps to understand some basic concepts, in order to use CYPEX even more efficiently.

CYPEX software architecture

Before we look at the architecture of a CYPEX app from an end user perspective, we first want to understand the overall software layout:

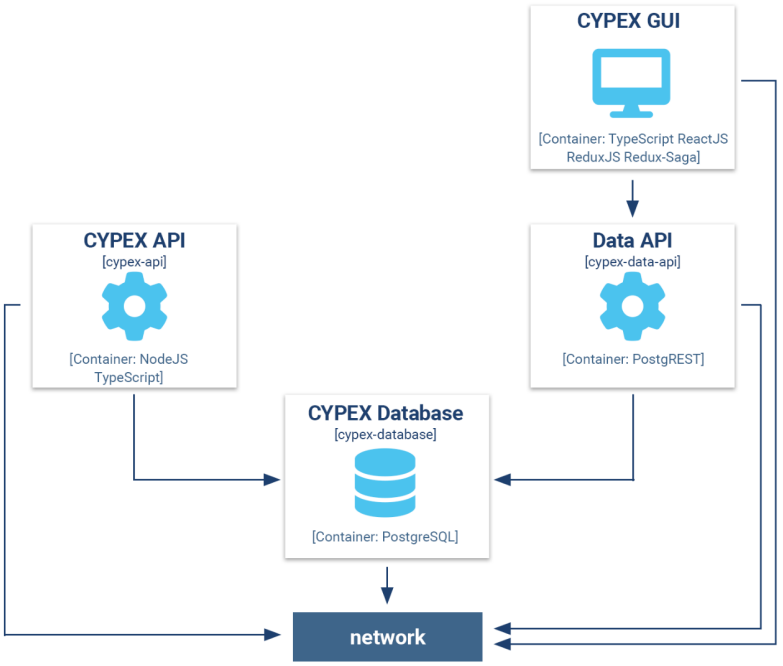


Delivering CYPEX

CYPEX is delivered as a set of Docker containers, which makes deployment easy and efficient. In general, CYPEX can run on top of an existing, standard PostgreSQL database. There are no dependencies on external extensions.

> **NOTE:**
> CYPEX does support GIS data types provided by PostGIS, but that's the only extension which is (optionally) needed. (No hard requirements).

CYPEX consists of the following containers:



- CYPEX GUI
- CYPEX API
- CYPEX data API
- CYPEX database

Let's take a look at each of these containers in a bit more detail.

CYPEX GUI ("renderer")

The CYPEX GUI container contains the end-user side of the tool chain. It contains a single web application and is the main entry point for all end-users. This is what is generally known as "the renderer".

The way it works is that it fetches a JSON document describing the application from the backend and turns it into a usable application in the browser. As previously stated, a CYPEX app is basically a giant JSON document describing the page and its interaction with the world.

As part of the container, we ship nginx, which acts as a reverse proxy for APIs. We use OpenResty to serve static data.

The following technologies are used.

Technology:

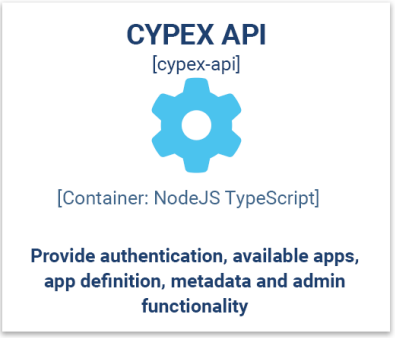
- TypeScript
- ReactJS
- ReduxJS
- Redux-Sata
- nginx
- OpenResty

Let's now focus on the way CYPEX handles data.

CYPEX API

There are two basic APIs: The CYPEX API and the CYPEX data API. The CYPEX API provides the the following functionality:

- Authentication services
- List of available apps
- Application definitions
- Meta data
- Administration functionality



The CYPEX API provides basic infrastructure and handles non-app related data using a standard REST interface (JSON).

Technology:

- TypeScript
- nodeJS

CYPEX data API

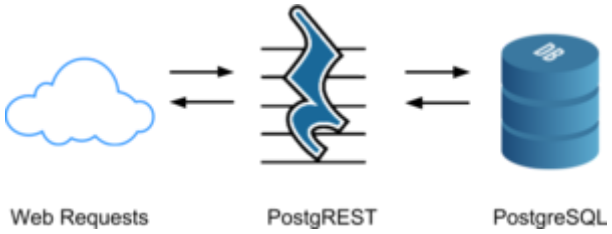
The CYPEX data API is used to serve application data. Every piece of end-user data will come from this side and not from the internal APIs.

Why is that necessary? PostgREST generates the API automatically from the database. This is due to various reasons:

- CYPEX is standard-compatible
- It relies on standard tooling
- Automatic documentation of app side
- Reliable and battle-tested

PostREST exposes exactly one schema as an auto-generated API. That's one (but not the only) reason why CYPEX uses views to abstract access to data. Using views exposed as a single schema by PostgREST, you can ...

- Handle security better (no need to modify permissions on base tables)
- Support apps working on multiple schemas
- Be more robust when it comes to changing column names, etc.



PostgREST is standard software widely used in the community.

CYPEX database

Finally, there's the database container. Strictly speaking, any PostgreSQL database is fine. However, to improve the user experience we will also ship PostgreSQL as part of the entire package. This makes it a lot easier for people who aren't yet running PostgreSQL at scale.

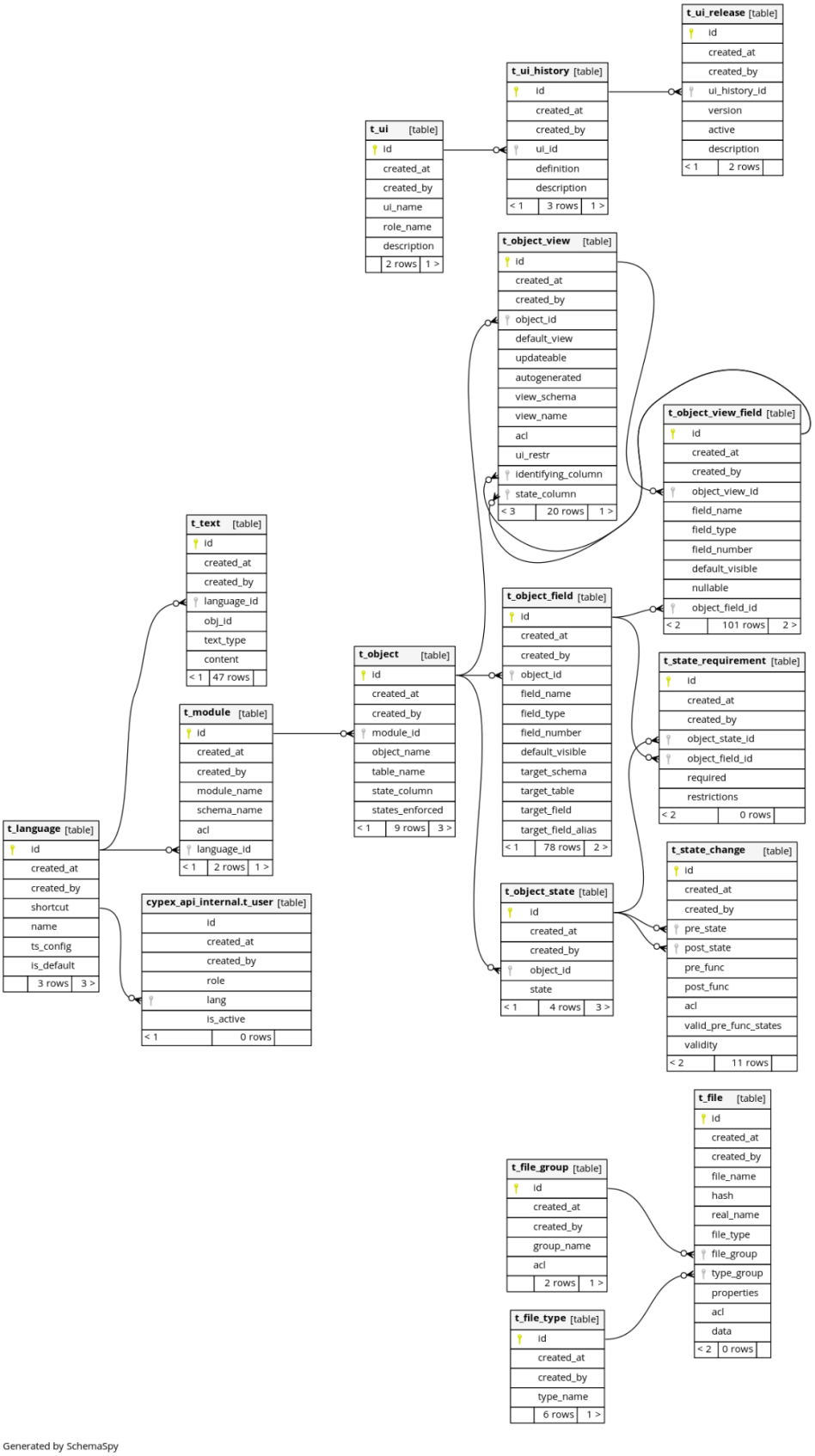
Upgrading CYPEX

If you want to upgrade, all you have to do is to run new containers. Usually no further action is needed. However, we will provide change scripts in case they're necessary in order to upgrade.

Please contact our support team for further information.

CYPEX internal data structure

In this section, we'll dive into the SQL structure of CYPEX itself and learn how data is stored inside the tooling. Here is the main data structure:



The purpose of the tables above is as follows:

Table cypex_api_internal.t_user

In CYPEX, there are three different types of users:

- Standard PostgreSQL Users
- Integrated users
- LDAP users

It can also be the case that a PostgreSQL user in the background is mapped to various email addresses in the frontend for authentication purposes.

Table t_file, t_filegroup, t_file_type:

CYPEX allows users to upload files. Since it's vital to maintain transactional integrity and expose those files via a REST interface, you can't just store them in a directory. Files in a filesystem can hardly be protected and in that case, you couldn't properly handle permissions. In addition to that, it's important to maintain the ability to back up an entire CYPEX deployment using a single database backup.

Therefore all files are stored in a table (t_file). In CYPEX, files have types and belong to groups. This is allows it to handle groups and permissions more easily and in a more organized way.

Table t_language

CYPEX supports various languages. The language table contains the supported languages. The table is mainly used to ensure referential integrity across the system. Note that not all texts are stored on the database side. Some texts are also part of the JSON document sent to the rendering engine.

Table t_module

CYPEX is structured in 3 levels. Note that the levels aren't immediately visible to the end user. Behind the scenes, we have a hierarchy of "Modules -> Objects ("tables CYPEX is keeping track of") -> Objects views (= "queries").

The t_module table is the fundamental building block to represent this hierarchy at the database level.

Table t_object

Objects are basically “tables CYPEX is tracking”. Tables are a fundamental building block of any relational database. It can very well be the case that a single relational model is the foundation for more than one CYPEX application. Therefore an application has to know which tables to track in order to store metadata (column names, etc).

At the object level, CYPEX also tracks whether workflows and constraints are enforced inside the metadata. CYPEX enforces workflows by deploying triggers and constraints on the underlying tables.

Table t_object_field

CYPEX needs a lot of metadata to fuel the default rendering process. Therefore a lot of information about fields is stored in the t_object_field table. This includes, but isn't limited to: field names, field orders, visibility, etc.

Table t_object_state

In case workflows are enabled for an object, we need to store the states an object can have (“Status” in the GUI example - see the section “Creating Workflows” above). As an example: A contract can be “offered”, “signed”, “rejected” and so on. The states associated with an object are in t_object_state. States can be added on the fly using the CYPEX GUI.

Table t_object_view

The CYPEX core engine knows the concept of “object views”. To the end user “object views” are presented as “queries” in the model builder. The idea is to have an abstraction layer between tables and the way data is presented. This is especially important in case of aggregations, default filters and alike. Metadata is associated with every object view (names, translations, etc.).

Table t_object_view_field

Similar to the way object columns are treated, we also keep track of object view columns. Object views (= queries) can have completely different columns than the underlying object does. (As an example, think of aggregations)..

Table t_state_change

States are the foundation of every workflow. State changes are a way to move from one state to another. Somebody might move a contract from “offered -> signed” (= ”sign”) - but not from “signed -> offered”. Control this using database side constraints.

However, often the next state has to be calculated using functions. The way to do that is to use “pre-funcs” and “post-funcs”. The “pre-func” is called before a state is left (to determine where to go in the state machine). The post-func is called before entering the target state. We use standard PostgreSQL stored procedures to handle this behavior.

Note that the GUI does not fully support this concept yet.

Table t_state_requirement

It can happen that states need certain preconditions. As an example: A contract can only be in state “signed” if there is pricing information entered and so on. The t_state_requirements table defines which of those requirements have to be met.

Table t_text

Texts can be assigned to pretty much everything. This includes objects, columns, states, state changes and a lot more. In CYPEX, all configuration tables share a common sequence, providing us with a system-wide unique ID. The advantage is that every piece of information can be identified clearly in a unique way. Therefore it's easy to attach texts in various translations to everything stored in the database.

The t_text table is the place to store all those translations for all objects in the CYPEX metadata.

Table t_ui

A single database might serve more than just one UI. Let's imagine a webshop: The end user part (“customers”) will run application A while backoffice people will operate using application B. Both applications will access the same underlying data.

The way we represent that in CYPEX is by allowing multiple GUIs for the very same data to exist at the same time. In general, GUIs can be assigned to roles which means that a group of people can share the same graphical user interfaces.

Table t_ui_history

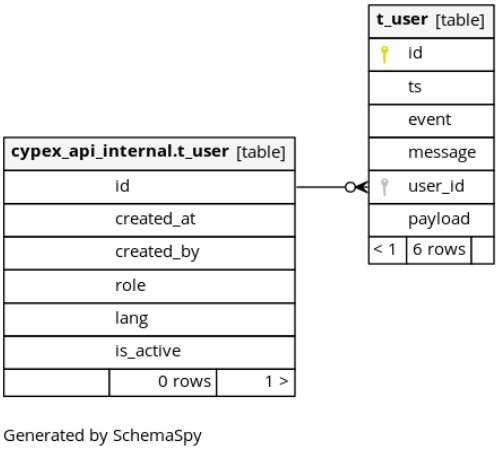
To allow for proper versioning all histories of graphical user interfaces are kept. This allows CYPEX to support releases, which allow superusers/admins to change applications while they are actually in use.

Security-related data structures

So far, we've discussed application-related metadata. In this section we'll use

Table t_user

We've already discussed internal users. However, there is more: You can map internal users to database users, as shown in this ER diagram:



But the story isn't as simple as it might seem:

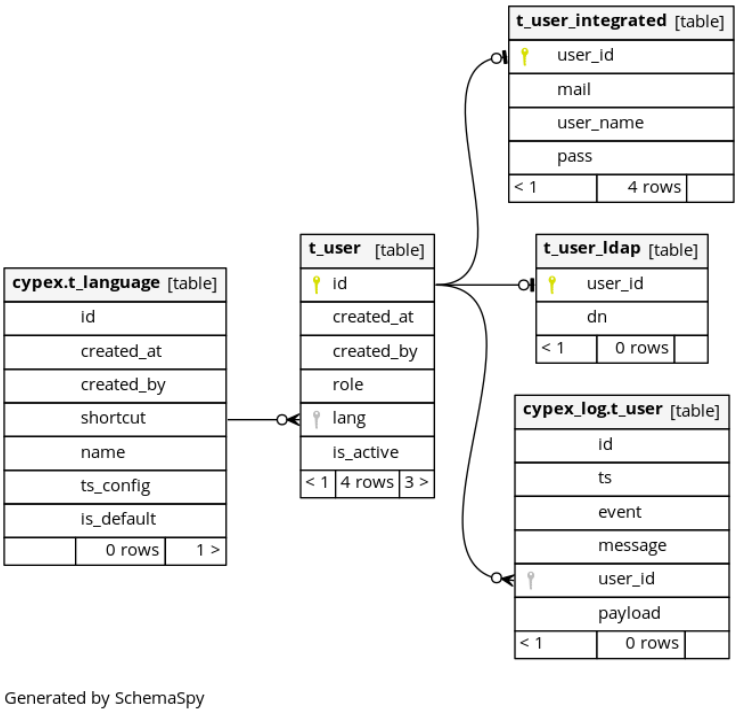


Table t_user_ldap

In case LDAP authentication is enabled, you have to map LDAP users to internal users (= database side). You need LDAP support to handle single-sign-on.

Table t_user_integrated

Integrated users support the idea of allowing multiple logins mapping to the same PostgreSQL user. Keep in mind that permissions on the “CYPEX Data API” side are controlled by the PostgreSQL user side. By defining an integrated user, it’s possible to map various logins to the same backend user. The same is true for LDAP as well.

Table cypex_log.t_user

In CYPEX, security is of the utmost importance. Therefore all access to the application is tracked and audited. The cypex_log schema facilitates tracking and auditing.

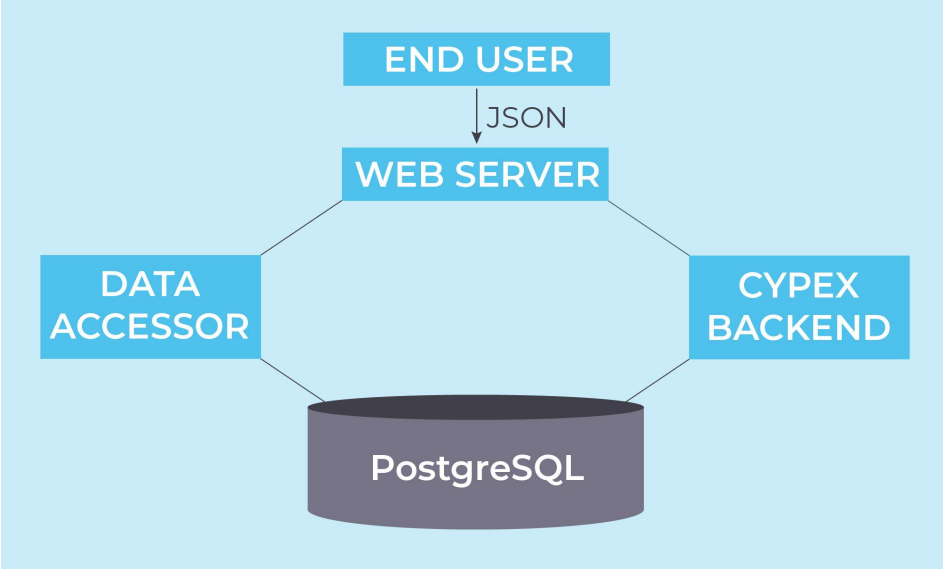
Application structure

This section includes a brief description of the underlying architecture used by CYPEX. It's presented from an end-user point of view. It's based on PostgreSQL, and stores a lot of metadata inside the database. This includes:

- Workflow definitions
- Object descriptions
- Pre-rendered definitions
- User mappings

All other components are controlled based on this information. The end product is a JSON document, which is sent to the client. The information is then rendered on the client. To make the process efficient, the JSON document is pre-computed and stored in PostgreSQL as well.

Let's take a look at the basic architecture:



As mentioned above, the "end product" is a JSON document rendered by the browser. To produce this document, we use middleware which creates the desired data. The core idea is to have everything ready for immediate use, to maintain good performance.

Fetching data is done using an API interface which is generated by inspecting the data model as well as the server side code. The API can also be accessed directly in case you want to write custom code.

State machine internals

Let's spend some time on the internals of the state machine. Basically all this metadata is stored in tables which can be found in the “cypex” schema. The state machine will create triggers on the data tables to ensure that data has to be correct on all levels.

Keep in mind: Most people will access data directly using their web browser. However, it’s also possible to just skip the GUI and talk to the API generated by CYPEX directly. Therefore it’s possible to enforce constraints, permissions and alike at the lowest level possible. It’s necessary to make absolutely certain that nobody can evade the business rules enforced by the model.

The integrity of data is one of the most important assets of a professional relational database. Therefore we do everything to protect your data. Let's have a look at an example: If an invoice is either “paid” or “unpaid”, we do not allow “maybe” or “who knows”. At the end of the day, you want to be “paid” and CYPEX enforces data integrity by all possible means. Fortunately, PostgreSQL provides us with the transactional foundation we need to actually do that. All layers built on top of PostgreSQL (= GUI, API, etc.) will automatically inherit PostgreSQL’s restrictions and business rules.

To show you what this means in real life, we've included a code snippet:

```
cypex=# \d todo.t_todo
                                Table "todo.t_todo"
  Column   |  Type   | Collation | Nullable |               Default
-----------+---------+-----------+----------+--------------------------------
 id        | integer |           | not null | nextval('todo.t_todo_…)
 tstamp    | date    |           |          | now()
 todo_item | text    |           | not null |
 status    | text    |           |          | 'created'::text
Indexes:
       "t_todo_pkey" PRIMARY KEY, btree (id)
Check constraints:
       "cypex_761c0b39d568e31024e53b9c3eadb8c5" CHECK (status = ANY
('{created,accepted,success,failed,rejected}'::text[]))
Triggers:
       zzz_e92d74ccacdc984afa0c517ad0d557a6 BEFORE INSERT OR DELETE OR UPDATE ON
todo.t_todo FOR EACH ROW EXECUTE FUNCTION cypex.trig_enforce_state_change('status')
```

As you can see, CYPEX generates a trigger with a unique name to ensure consistency and enforces those states' changes. It comes with some performance penalty, but is necessary to maintain integrity. Also keep in mind, if workflows are changed AFTER loading a lot of data changes to the workflow, it might be time consuming - because PostgreSQL has to revalidate those constraints.

We strongly advise CYPEX users against changing those constraints manually. Instead, use the CYPEX-internal functions to make sure that the metadata catalog stays consistent.

User management

A high level of protection must be assured of your data. We put great emphasis on security, and ensure that data is protected at all times. As part of that, our user management is based on a solid, well-tested user concept.

Understanding the CYPEX user concept

The first question we have to answer when talking about security is: "What is a user?". Having a clear picture in mind is important to understand the big picture.

There are three basic types of user authentication:

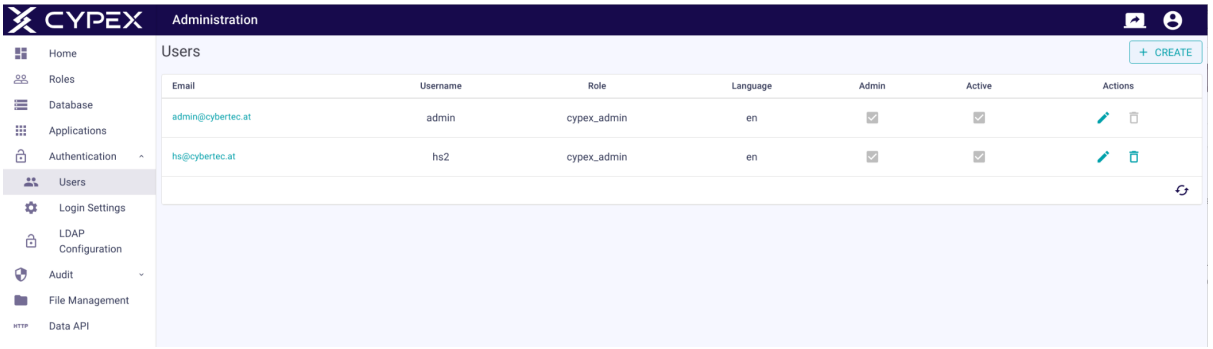
- Case A: "Database user" equals "application user"
- Case B: "Database user" is mapped to a "login user"
- Case C: "Database user" is mapped to single-sign on users

In the first case, life is fairly simple: You can log into an application using the same name and password as your database user. For many basic applications, this is perfectly fine.

However, sometimes (Case B) you are facing the situation that various "login users" should point to the same database role.

Here is an example: jane@example.com and jack@example.com are both fulfilling the role of "bookkeeper". We definitely want to separate the logins, but behind the scenes, they have the same permissions. In small companies, this is usually the default way of handling things.

To map login names to database users, you use the CYPEX admin panel to achieve the proper configuration.



Case C is the most “enterprise-ready” way of handling user authentication. CYPEX is able to handle generic modules to map CYPEX internals to external systems, which allows us to connect to systems usually used for single-sign on (LDAP, ActiveDirectory, etc). There are a variety of ways to connect to single-sign on systems: First of all, you can use PostgreSQL onboard, which means using authentication and the “Case A”-style.

Depending on your infrastructure, various levels of complexity and customization of the authentication module might be required.

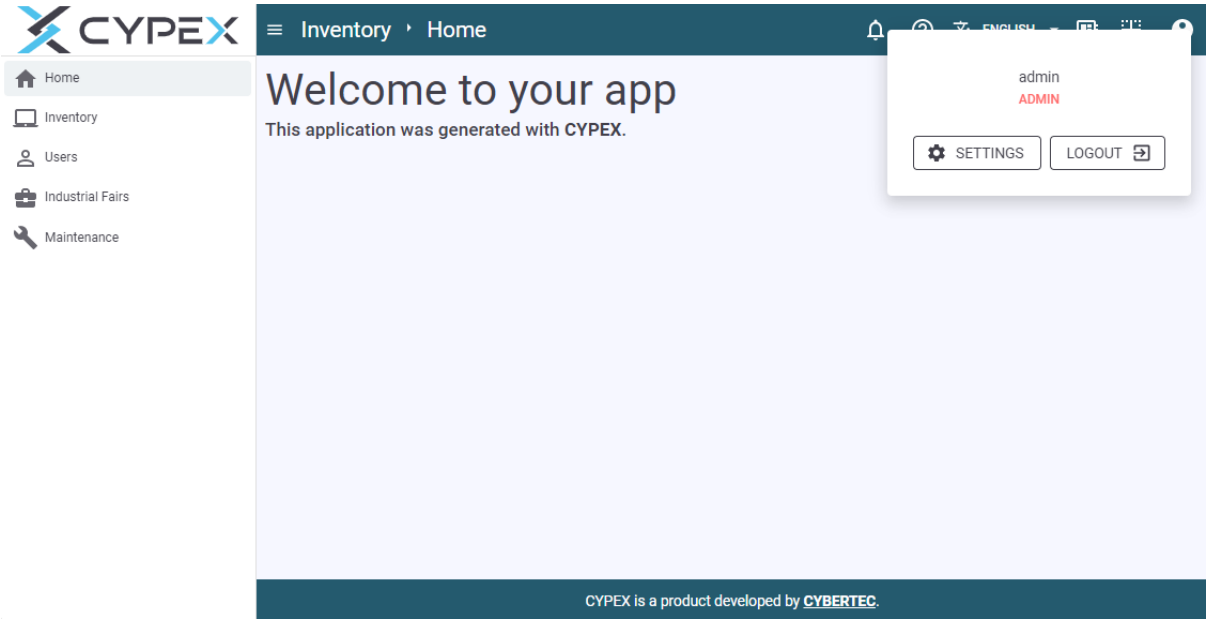
In general, it's always advisable to strongly focus on database-side permissions. In particular, PostgreSQL Row-Level-Security has proven to be a valuable asset in real-world applications.

Changing Password

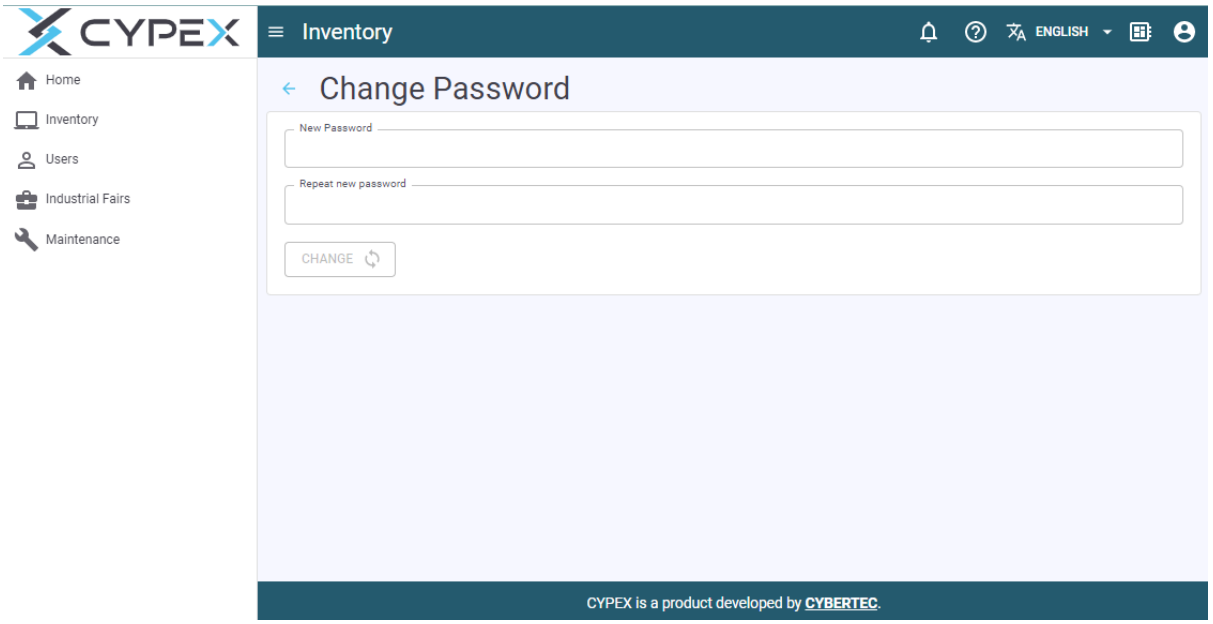
It makes sense to change passwords on a regular basis. In this section you'll learn how to perform such a task and which features are supported by CYPEX:

Changing our own password

The first thing to look at is how to change your own password. To do that, click on the user profile icon on the right side of the panel. A small overlay will appear and a click on the “SETTINGS” button opens the “change password” form.



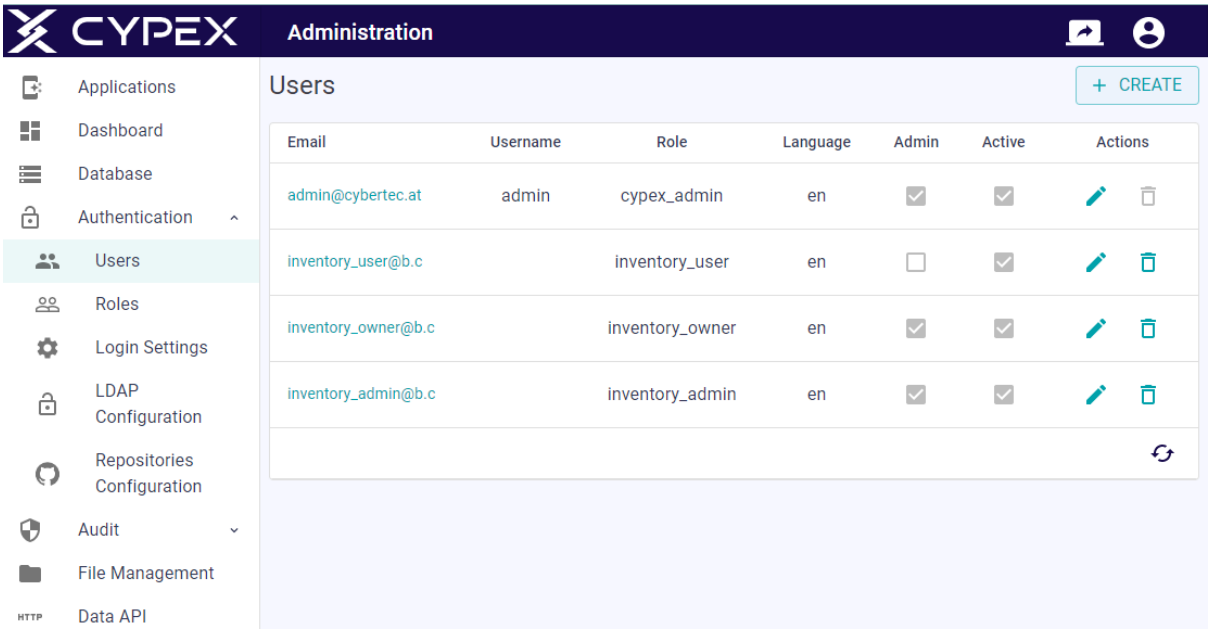
To change the password, type in the “New Password” field.



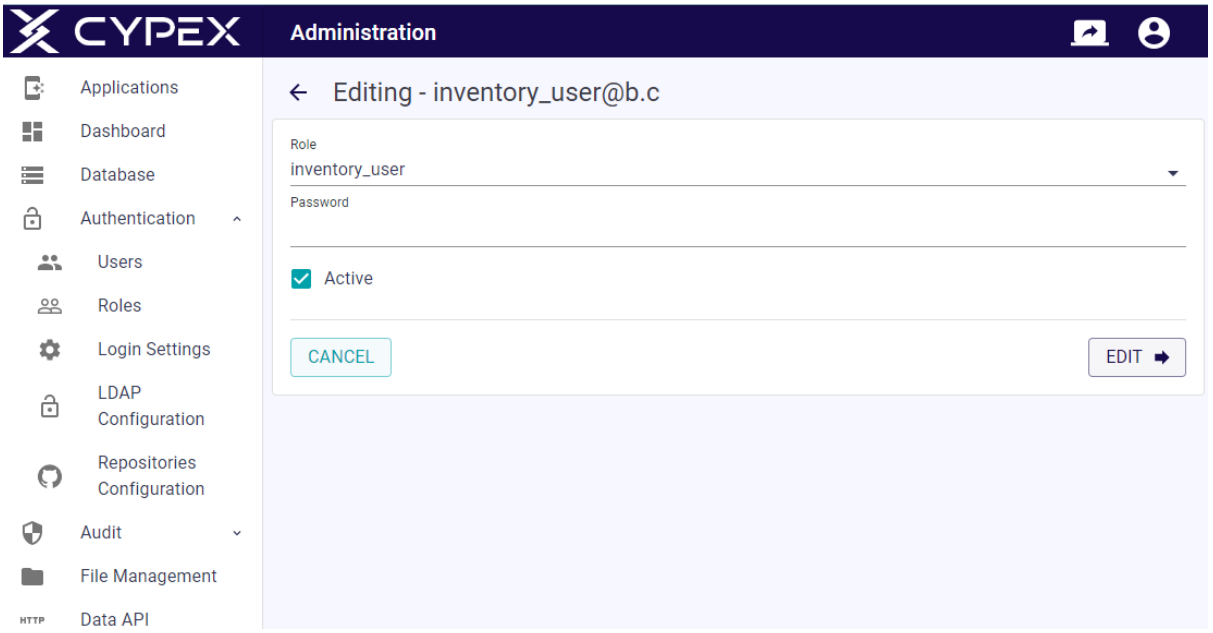
The new password will be active instantly. However, active sessions will not be terminated unless a user proactively logs out. As long as the JWT (= JavaScript Web Token) is valid, users can continue working normally.

Changing passwords as admin for other users

In addition to changing your own password, superusers can also change other users’ passwords quickly and efficiently. In the “authentication” section of the CYPEX admin panel you can click on “users”. There you'll find a list of users:



By clicking at the “pen” you'll find your way to the desired form which allows you to change the password easily. Again, changing the password isn't going to terminate existing sessions:



Please ensure that passwords are sufficiently strong. At the moment, CYPEX does not enforce password rules. The reason is that the PostgreSQL protocol is implemented in a way that the server never sees the plain-text password. Therefore, we cannot guarantee the strength of the password on the server side.

Known bugs and pending improvements

In this section we'll discuss known bugs as well as features which are still missing and which might be implemented in the foreseeable future.

Security features

This section will discuss missing security related features which will be added to CYPEX in the future to make the product more comprehensive.

Ability to create nested roles

Currently it's possible to create database roles in the admin panel. However, it's not possible to assign those roles to other roles yet. We'll fix this in the future and make the feature more complete.

Provide an overview of permissions

In the future we'll provide an easy-to-use overview to give developers a better way to keep track of permissions. In addition to a complete list, we're planning to create a diagram.

View handling

The following view-related issues are known and should be kept in mind to ensure smooth operation of CYPEX. Note that "queries" in CYPEX are stored as views on the database level to ensure dependency tracking as well as security abstractions.

CREATE VIEW … WITH CHECK OPTION

Currently CYPEX doesn't use "WITH CHECK OPTION". Why does it matter? Suppose you get a query that only shows "data in your country". In case the query is simple, PostgreSQL will make it "auto-updatable" which means that you can INSERT, UPDATE, and DELETE. However, without the "WITH CHECK"-option you can theoretically insert data which cannot be seen later anymore (by adding data not in your country). This will be fixed in the future.

Views and dependencies

Since we're using views to abstract the underlying data model from the GUI side of the app, you need to be aware that PostgreSQL will drop cascading objects. This is relevant because it can remove the data source needed by your app.

We are currently working on code to make dependency tracking easier and more transparent.

Security barrier views

At the moment we don't use "security barriers" views for efficiency reasons. We therefore recommend not using stored procedures which make extensive use of RAISE NOTICE. Also make sure that functions which have side effects or use RAISE NOTICE are checked and marked as NOT LEAKPROOF.

Data type handling

Currently the "interval" data type isn't fully supported under every circumstance. Therefore "interval" is mostly seen as text and does not offer any additional functionality which might be desired.

We're working to remove this limitation.

GIS data handling

CYPEX supports GIS data. However, to render GIS data in the GUI, the query you are using has to provide the frontend with a GeoJSON.

There are currently two options to handle this:

- Create a GeoJSON as part of a query
- Use a ready-made GeoJSON column inside the underlying tables

Since GeoJSONs aren't automatically generated by the default rendering process, you need to generate them as part of the query. The following queries contain examples which show how this can be done:

```
select
      inspection.id,
      tv.license_plate,
        tv.vehicle_model_id,
        tv.registered_country,
        ST_AsGeoJSON(gps_pos)::jsonb gps_pos,
        x.server_tstamp tracked_at,
        inspection.tstamp inspected_at,
        inspection.base_station_id,
        inspection.inspection_type_id,
        inspection.aggregate_status
      from backoffice.t_vehicle tv
          left join backoffice.t_vehicle_gps_device tvd
          on (tv.id=tvd.vehicle_id)
      left join lateral
        select
          g.id,
          g.gps_pos,
          g.server_tstamp
        from gps_track.t_track AS g
        where
          g.device_code = tvd.gps_device_id and
          g.server_tstamp > g.server_tstamp - interval ''1 week''
        order by g.server_tstamp desc LIMIT 1) x on true,
      lateral (
        select
          ti.id,
          ti.base_station_id,
          ti.tstamp,
          ti.inspection_type_id,
          ti.aggregate_status
        from backoffice.t_inspection ti
        where
          ti.vehicle_id = tv.id and
          ti.inspection_state = ''clean''
        order by ti.tstamp desc limit 1
      ) as inspection,
      backoffice.t_inspection_type tit
```

```
      where
        inspection.inspection_type_id=tit.id and
        tit.inspection_type = ''rent_out'' and
        (tvd.active is null or tvd.active)
      order by inspection.tstamp desc
SELECT
      json_build_object(
        'type', 'FeatureCollection',
        'features', data_danger_zone.tracks || ST_AsGeoJSON(data_danger_zone.*)::jsonb
      ) danger_zone_tracks
    from (
      select py.area_name as name,
        py.polygon as geom,
        jsonb_agg(ST_AsGeoJSON(pt.*)::jsonb) as tracks
      from (
        select vlt.license_plate, vlt.server_tstamp, vlt.gps_pos
               from  cypex_generated.v_vehicle_latest_gps_track vlt
        ) as pt
      JOIN (
        select a.*
        from
          backoffice.t_danger_area a,
          backoffice.t_area_type b
        where a.area_type_id=b.id and b.area_type = 'danger_area'
        and now() between a.active_from and a.active_until
      ) py ON ST_Intersects(py.polygon, pt.gps_pos)
      group by py.area_name, py.polygon
    ) as data_danger_zone
```

In this case, we used the ST_AsGeoJSON function to do the magic. But here's one more example:

```
with stations_fence as (
        select
          ST_Union(ST_Buffer(ta.gps, 50000)::geometry) as fence
          FROM backoffice.t_base_station tbs inner join backoffice.t_address ta
               on (tbs.address_id=ta.id)
      ), violated_tracks as (
        select
          jsonb_agg(json_build_object(
              ''type'', ''Feature'',
              ''geometry'', ST_AsGeoJSON(tt.gps_pos)::jsonb,
              ''properties'', json_build_object(
              ''license_plate'', tv.license_plate,
              ''registered_country'', tv.registered_country,
              ''tracked_at'', tt.server_tstamp
            )))::jsonb as tracks
        from
          gps_track.t_track tt inner join
          backoffice.t_vehicle_gps_device vd on (tt.device_code = vd.gps_device_id)
          inner join backoffice.t_vehicle tv on(vd.vehicle_id=tv.id),
          stations_fence
        where
          not ST_Intersects(stations_fence.fence, tt.gps_pos)
          and  tt.server_tstamp > tt.server_tstamp - interval ''1 month''
      )
      select
        violated_tracks.tracks || ST_AsGeoJSON(stations_fence.fence)::jsonb
```

```
            as violated_tracks
    from stations_fence, violated_tracks
```

Take note that in case of a default query (= “SELECT * FROM tab”), you don’t have to worry about INSERT, UPDATE, and DELETE. PostgreSQL knows that the view is auto-updatable and you don’t have to add additional code. However, this isn’t true in case of a query that generates a GeoJSON.

A trigger has to be added on top of the query (= view) to teach PostgreSQL how to modify data. Note this view isn't auto-updatable anymore, and therefore the way back to the table has to be defined by developers.

The following listing shows how such triggers can be made:

```
CREATE OR REPLACE FUNCTION danger_area_fn() RETURNS TRIGGER
AS $$
  DECLARE
    _polygon backoffice.t_danger_area.polygon%type;
    new_return record;
  BEGIN
    if new.polygon isn't null then
      select ST_GeomFromGeoJSON(coalesce(new.polygon::json->'features'->0->'geometry',
new.polygon::json)) into _polygon;
    end if;
    IF (TG_OP = 'INSERT') THEN
      INSERT INTO backoffice.t_danger_area(
        area_name,
        polygon,
        reason,
        active_from,
        active_until,
        area_type_id
      ) VALUES (
        new.area_name,
        _polygon,
        new.reason,
        new.active_from,
        new.active_until,
        new.area_type_id
      );
      select
        * into new_return
      from cypex_generated.v_backoffice_t_danger_area
      where id = currval('backoffice.t_danger_area_id_seq'::regclass);
      return new_return;
    END IF;
    IF (TG_OP = 'UPDATE') THEN
      UPDATE backoffice.t_danger_area SET
        area_name = new.area_name,
        polygon = _polygon,
        reason = new.reason,
        active_from = new.active_from,
        active_until = new.active_until,
        area_type_id = new.area_type_id
      WHERE id=new.id;
```

```
      return new;
    END IF;
    IF (TG_OP = 'DELETE') then
      delete from backoffice.t_danger_area where id = old.id;
    END IF;
    return null;
  END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER t_danger_area_trigger
  INSTEAD OF INSERT OR UPDATE OR DELETE ON cypex_generated.v_backoffice_t_danger_area
  FOR EACH ROW
  EXECUTE PROCEDURE danger_area_fn();
```

If you want to know more about triggers in PostgreSQL, check out the PostgreSQL documentation..

ER-model related issues

At this point it's not yet possible to create data models from scratch inside the admin panel. We're working hard to fix this issue in the next release.

Missing model creation

You'll soon be able to create models from scratch and you'll be able to define triggers on “queries” to make it easier to insert and update more complicated operations. You will also make better use of the information available inside the data model during default rendering.

Workflows and foreign keys

At the moment, foreign keys are defined on columns. In case states are derived from a column, take all existing values from this column. However, you often may be in a 1:n relationship, with the workflow playing out on the “n” side of the foreign key relation.

In future releases we will allow the workflow to take all possible keys from the “1” side of the join as the “n” side might not contain all values known to the “1” side of the relation.

Alternatively you should be able to specify some kind of data source to fetch all possible states from the existing model (especially important in case you’re dealing with more transitive or more complex models in general).

Pre-func and post-func enabled workflows

At this point you can transition from one state to the other. However, what if state changes should only happen under certain conditions? Let’s take a look at an example: A person applies for a bank loan. The loan is only granted in case some more complicated calculations give the OK. Currently it's possible to do this using triggers. However, in the future our team will integrate this with workflows more tightly. The idea is to use “pre-funcs” and “post-funcs”. A pre-func will be called shortly after leaving a state to determine the target state (= loan will be granted or rejected based on some other data).

Graphical user interface

In this sectionwe'll discuss bugs and missing features related to the graphical user interface designer.

Multiple file uploads

At this point files have to be uploaded one by one. Multi-file uploads are currently not working. However,we'll support this feature in the future.

Handling of password fields

At the moment CYPEX does not support HTML password fields. HTML Password fields do not show “letters” but use “dots” to hide the password. This will most likely be changed in the future.

Glossary

abstraction layer	software that translates higher-level requests into lower-level commands the computer can use (for example, an API, or Application Programming Interface, is an abstraction layer which communicates between an application and the operating system.)
FDW	**F**oreign **D**ata **W**rapper: "a library that can communicate with an external data source, hiding the details of connecting to the data source and obtaining data from it." source: PostgreSQL Documentation
GUI	**G**raphical **U**ser **I**nterface: the software which allows users to visually control an underlying data structure, as opposed to a command-line interface, which requires the user to memorize text-based commands.
JSON	**J**ava**S**cript **O**bject **N**otation is a common data format used to exchange data, i.e. between web applications and servers. JSON is a language-independent data format. JSON file names use the extension .json. source: Wikipedia
relational model	"The relational model (RM) for database management is an approach to managing data using a structure... where all data is represented in terms of tuples, grouped into relations... Users directly state what information the database contains and what information they want from it, and let the database management system software take care of describing data structures for storing the data and retrieval procedures for answering queries." source: Wikipedia

CYBERTEC PostgreSQL International (HQ)

Gröhrmühlgasse 26
2700 Wiener Neustadt
Austria
Phone: +43 (0)2622 93022-0
E-Mail: sales@cybertec.at

CYBERTEC PostgreSQL Nordic

Fahle Office
Tartu mnt 84a-M302
10112 Tallinn
Estonia
Phone: +372 53070910
E-Mail: sales@cybertec.at

CYBERTEC PostgreSQL South America

Misiones 1486
oficina 301
11000 Montevideo
Uruguay
E-Mail: sales@cybertec.at

CYBERTEC PostgreSQL Switzerland

Bahnhofstraße 10
8001 Zürich
Switzerland
Phone: +41 43 456 2684
E-Mail: sales@cybertec.at

CYBERTEC PostgreSQL Poland

Aleje Jerozolimskie 93
HubHub Nowogrodzka
Square, 2nd floor
02-001 Warsaw
Poland
E-Mail: sales@cybertec.at

CYBERTEC PostgreSQL South Africa

No. 26, Cambridge Office
Park
5 Bauhinia Street, Highveld
Techno Park
0046 Centurion
South Africa
Phone: +27(0)012 881 1911
E-Mail: africa@cybertec.at

www.cybertec-postgresql.com
Facebook | Twitter | LinkedIn | Xing | GitHub