Model Data

A complete Guide to Model Data in Charis

Table of Contents

Introduction

In a Charis solution, there are two distinct types of databases:

- **model database** and
- **custom database**.

Model database is created and maintained by modeling users—the developers of the solution. It is defined at design time and stores the **model** of the solution.

Custom database, on the other hand, is maintained by the customer's users—the end users of the solution. Each customer typically has their own custom database. Even within a single organization, multiple custom databases may exist, for example for development, testing, and production environments.

Charis developers provide many aspects of the model. The foremost aspect is **entity**. Developers define entities (the structure of data) and end users maintain **data** for those entities.

However, some aspects of the solution may require that the developer not only defines the structure for data but even defines **data** for it.

Because this data exists already during modeling, it can be:

- referenced by other model elements, and
- directly be used in the solution's code.

Unlike custom data, this data is part of the model itself. In Charis, it is referred to as **model data**.

Model data is **replicated into the custom database**, making it accessible at runtime. This enables also custom data to reference objects of the model.

model data is found in both databases

Even though those model data objects appear in both databases, they are still considered "model data" because:

- They originate from the model.
- They are defined and controlled by developers—at design time.
- They cannot be fundamentally altered by end users.

Example: Privileges

- A Charis model defines an entity (a structure for data) named **privilege**.
- Developers also provide data for it as **model data**.
- User entry points (also part of the model) may require specific privileges, so privileges must be selectable during modeling.
- At runtime, customer users create application users (custom data).
- These users must be assigned privileges—the same privileges as defined by the developers.

Because privileges, in this example, are defined in the model but used by model objects (user entry points) **and** custom objects (users), they must be available in both databases.

Why use Model Data

There are two main reasons to use model data.

Predefined Data

If certain objects are expected to exist in the system, developers can prepare them in advance by defining them as model data and creating them at design time.

As a result, these objects are already available at runtime, even if the custom database is newly created. End users can immediately work with them without needing to create them first.

Code depending on Data

If a reference to an object may vary at runtime, but the application logic depends on what is being referenced, developers can define those objects as model data.

By doing so, the referenced objects are already known at design time, allowing developers to safely reference them in code and build logic that depends on them.

How to define the Model for Model Data

```
1. Mark entity as “model data”
2. Invoke behavior “assure usable”
```

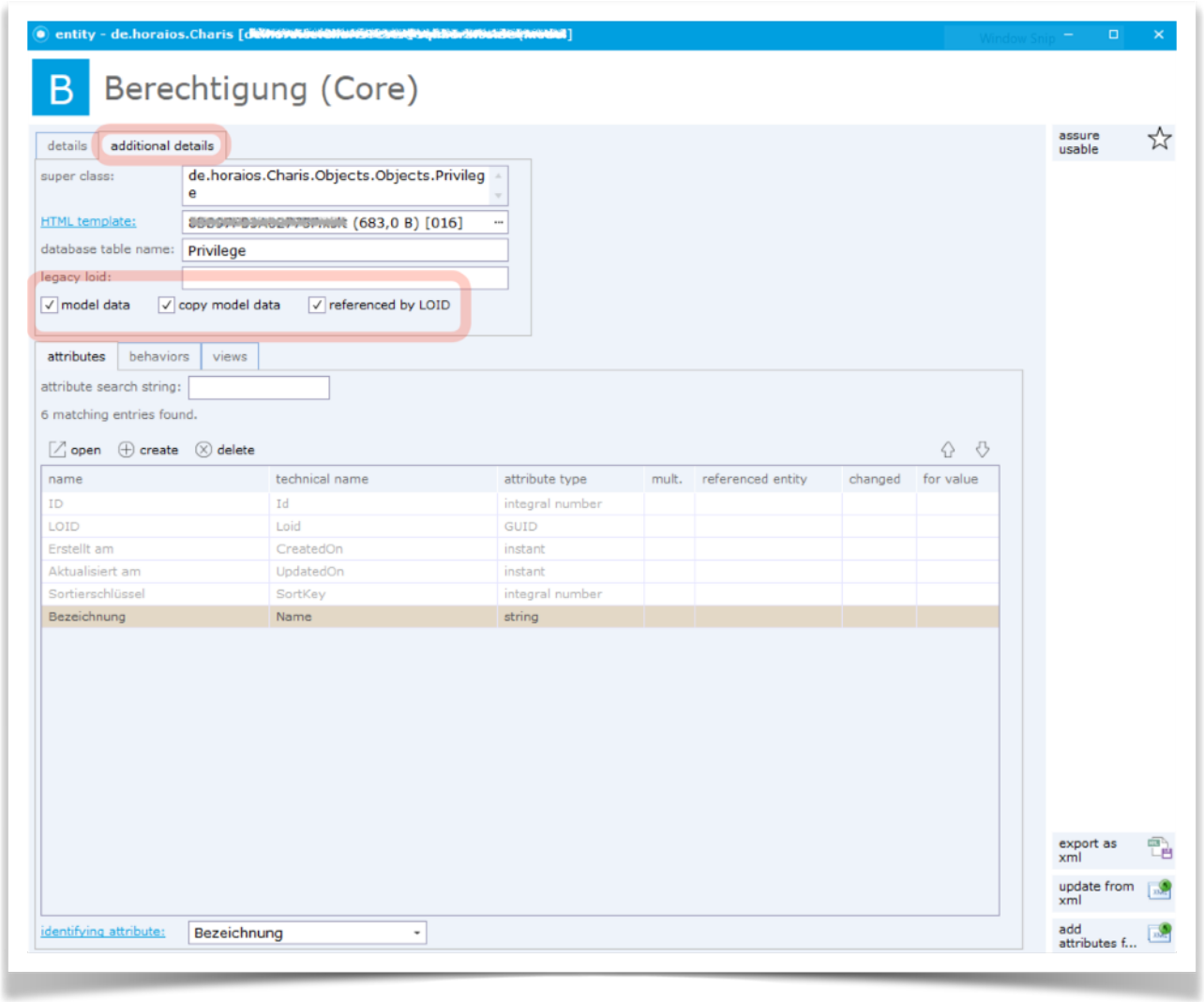
When defining entities, developers describe the structure of the data (what data exists and how it is organized), and users fill this structure with actual information at runtime.

For model data, developers go one step further: they not only define the entity but also create objects of that entity within the modeling environment. These objects become part of the model itself.

Defining model data therefore starts with defining an entity, just like any other entity. Then, on the "additional details" tab, the option "model data" is activated.

When this option is enabled, the system automatically activates:

- "copy model data", ensuring the data is available in the custom database, and
- "referenced by LOID", allowing the objects to be reliably referenced.



defining an entity as model data

As for any other entity, an administration layer can be created for model data entities by invoking the behavior "assure usable". This will make sure of the following structure for the model data entity:

- a form view,
- a table view,
- an administration entity, and

- a user entry point

In this case, both the administration entity and the user entry point are also marked as “model data”.

How to provide Data for Model Data

1. Generate code
2. Close Charis
3. Ensure the custom project is compilable
4. Restart Charis
5. Invoke the model data entry point
6. Create, update, or delete objects

As with any other entity, after defining an entity for model data, code must be generated before it can be used. The application must then be restarted for the changes to take effect.

In this case, however, the application that needs to be restarted is Charis itself. 🤯

After restarting Charis, all user entry points associated with model data are grouped and presented in a separate window titled “model data”.

If this window is not visible, it can be opened by focusing the main window of Charis and pressing **F8**.

Sometimes, generating code and getting the custom project to compile can be challenging during development. The following recommendations can help:

- Make small changes in the model and generate code frequently.
  - This makes it easier to identify which modeling step caused a problem.
- Make small changes in code and commit often.
  - This helps you track changes and revert if necessary.
- Commit code changes before generating code.
  - This allows you to easily discard generated code and retry if something goes wrong.
- Make use of the available code generation options.
  - Option “validate model”: Validation of the model before code generation can be skipped.
  - Option “assure objects project is published”: Building and publishing custom project before code generation can be skipped.
  - Option “re-generate all entities”: Only code for changed entities, attributes, etc. is generated. This speeds up code generation considerably.
  - Option “generate application project” and “generate service project”: Generation can be focused on the custom library project to speed up code generation.
  - Option “generate database backup”: Generating the backup file of the model database can be skipped to speed up code generation.

How to access Model Data Objects in Code

1. Create, update, or delete model data objects
2. Generate code
3. Reference model data objects using the automatically generated static variables of their base class

For every model data object, a **static variable** is generated in the corresponding base class. The name of the variable is derived from the value of the identifying attribute. The value is automatically converted into a format that is valid for identifiers in the programming language.

For example a privilege named "Administrator" can be accessed using the static variable `Administrator` in class `PrivilegeBase`:

```
var privilege = PrivilegeBase.Administrator;
```

Using static members to access model data objects provides several benefits:

- Safe and direct access
- Compile-time safety through strongly typed access
- Clear and readable code when working with predefined data
- Easy to find references to model data objects

How to reference Model Data within the Model

As with any other entity, model data entities can be used as the referenced entity of an attribute.

If the recommended option "referenced by LOID" is enabled for the model data entity, the corresponding attribute must use the type "**reference by LOID**".

This ensures that references to model data objects are consistent, stable, and safe, even across deployments and updates.

Although not recommended, model data can technically also be referenced by ID.

Replicating into Customer Data

When a new version of the solution is rolled out, the model data is replicated into every custom database to ensure that all required predefined objects are available at runtime.

Model Data referenced by LOID

If model data is referenced by LOID, existing model data in the custom database is not simply overwritten. Instead, it is **merged** with the model data from the updated model:

- **New model data objects** are added to the custom database.
- **Existing objects** are updated if they have changed in the model.

Because model data is identified via stable identifiers (LOIDs), the system can reliably match and update the corresponding objects across versions.

As a result, a single table can contain objects originating from more than one model. This is a significant advantage, since it is common for multiple Charis solutions to share the same custom database.

However, deleting model data objects must be handled explicitly using a data migration task.

Model Data referenced by ID

If model data is referenced by ID, maintaining consistent identifiers requires a different approach. In this case, the corresponding table in the custom database is **truncated**—all data is deleted and sequences are reset. After that, objects are recreated based on the model definition.

This approach makes it impossible to combine model data from multiple models, as each replication **fully overwrites** the previous one.