Best Practices

# Table of Contents

[Table of Contents](#h.wucj6secit4g)

[Starting on a new customer project](#h.x570f34y9h3f)

[Start with the starter project](#h.xj7bd5bfgnjv)

[Customize the project](#h.d51m6egffkp)

[Start modeling](#h.a3i7zf9jduv3)

[Data Spaces](#h.cb1u4vvu49vw)

[Permissions](#h.po3pond7jxrx)

[Purging](#h.nh0jbm4d5d3l)

[Naming](#h.v2w99od3zmxh)

[Computed fields](#h.rwwp082ygfm9)

[Modules](#h.ucvknmq7e8tx)

[Embedded v. In Module](#h.4ypmue3yscv2)

[Custom Code Guidelines](#h.w17cksxoczih)

[Data Models](#h.fdtpr3yzdxc5)

[Tables](#h.mhb4si9quzsl)

[Foreign Keys](#h.1dewbp5vkekt)

[Naming](#h.zi5bmk9ydyti)

[Required fields](#h.56cdipvqdg4f)

[Inherited fields](#h.s4uj9o4lyipl)

[Organization of data models](#h.blry44vzdq6a)

[Users/Roles](#h.4c60oyjx1s3h)

[Collaboration on projects](#h.ymd486fseh8b)

[Checking in updates to ps library](#h.eckebiwggczk)

[Troubleshooting](#h.kuvumccfuls2)

[Cannot open a dataSet [SqlChecker] error](#h.kwg11kcm1otm)

# Starting on a new customer project

## Start with the starter project

Download the starter workspace and the starter project from Git/Bitbucket.

* <https://bitbucket.org/orchestranetworks/psworkspace>
* <https://bitbucket.org/orchestranetworks/ps-mdm>

Follow the instructions, modify the web.xml, etc to make this project one for the new customer. See [PS Getting Started > Setting up...](https://docs.google.com/document/d/1BI_mmvWprSr9I3WVKy9cd525g2b8wXU8sMx-HDaY7QQ/edit#heading=h.yiu7lbqsbiwj) for detailed instructions.

## Customize the project

Admin icon and color palette -- lots of wow factor in such an easy configuration step.

## Start modeling

Identify the logical domains of the customer’s EBX data. Be sure to keep an open mind about future domains and choose critically what might be common reference data that would be reused across multiple domains. Then create data models for each domain:

* Type model
* Reference model
* Domain model

Including:

* Common types (PS)
* Common reference

By convention, the ‘types’ models contain reusuable complex and simple types. PS framework ships with Basic Types Data Model which includes Base (id, name, description), Base String (maxLength=80), etc.

For example, an entertainment customer may have one domain for its content catalog and another for its talent hierarchy. Data models might include:

* Content Data Model
* Content Reference Data Model
* Talent Data Data Model
* Talent Reference Data Model
* Common Reference Data Model
* Common Types Data Model

For each domain, there will be a master data space and data sets, logically named:

* Content Master Data Space
  + Content Data Set
  + Content Reference Data Set
* Talent Master Data Space
  + Talent Data Set
  + Talent Reference Data Set
* Common Master Data Space
  + Common Reference Data Set

More generally, for each domain, there could be up to three data models:

1. Types
2. Reference Data
3. Master Data

And these should be named:

1. <Domain>Types Data Model
2. <Domain> Reference Data Model (includes type model above and other required types models)
3. <Domain> Data Model (includes above type model as well as Common Types Data Model (if applicable) and Basic Types Data Model)

# Data Spaces

## Permissions

When creating child data spaces for workflows, etc, be sure to copy permissions <add detail>

## Purging

It is best to set up automatic purging of closed data spaces regularly to get the best memory usage/performance from EBX5.

## Naming

When a workflow creates a child data space, it is nice to give it a logical name based on that workflow instead of one with the usual timestamp-based naming.

# Computed fields

As computed fields represent a performance concern when used in tables expected to have many records, or when used in rules, etc, we have some general guidelines on how to best make use of computed fields.

1. disable history and replication
2. avoid using computed fields in constraints if the computation is expensive

# Modules

## Embedded v. In Module

How do you choose whether to have a data model be embedded versus in-module?

The difference is that embedded is stored in a data space much like master data and must be migrated between environments like master data. The plus is that it is version controlled inside EBX because it's using data spaces.

In module means the data model is exported as an xsd and stored in the model (war). The disadvantage is that version control is up to customer. The advantage is easy migration between environments.

## Custom Code Guidelines

1. Never use magic strings if you can avoid it. Add to a constants class or create an enum class so that these values are shared and can change in a reduced number of places.
2. Use AdaptationUtil to follow model references from a value context or an adaptation.
3. When possible, use a Constraint rather than a TableConstraintWithRecordLevelCheck because the Constraint message will show up on the field in question in the UI rather than only summarized at the record level.

# Data Models

## Tables

base/id primary key

base/name uniqueness

historization: all data spaces

reference data models: tables alphabetized

## Foreign Keys

Not recommended to override the default record label

When part of a primary key, make minOccurs= 1/check null input

## Naming

field names lower camel case

acronyms, camel (first upper, rest lower)

table/group names upper camel case

labels capitalized all capitalized

leading underscores for inherited or internal use only fields, usually ‘hidden’ (advanced properties)

FK lists: name is singular (for XML), label is plural

## Required fields

By default, advanced properties: check null input yes

Although not necessary for key fields, it doesn’t hurt and avoids a model validation warning…

As a best practice, we generally make Boolean fields required with a default of ‘No’.

Note: booleans with default false -- when adding a field, existing data will not have a value for it -- bulk change required to update to default

## Inherited fields

Many times it will make data entry easier to display a hierarchy in the UI. For example, a UI for entering an address will often have the user choose the country first in order to get a filtered list of states. But since state references country, it is redundant to make country a persistent value of address. Really, it should be an inherited field, inherited from state’s country. The best practice for dealing with such an inherited field that is really used as ‘syntactic sugar’ in the UI, is to use the (new in ps library) ResetInheritedFields trigger on the table. ResetInheritedFields can be configured with a ‘;’ separated list of path strings representing the inherited fields you always want to reset before applying a modification (never want to persist the overwrites). If you already have a custom trigger for the table, you can simply add a handleBeforeModify to reset the field to inherited with a line like:

context.setValue(AdaptationValue.INHERIT\_VALUE, fieldPath);

## Organization of data models

By convention, within a domain, it is a good idea to try and delineate reference data from master data, although the lines can be blurred sometimes. In general, the master data set for a domain will have the primary tables associated with the business (for example, Cafe for Panera’s Cafe Master, and ContentContainer and ContentItem for Viacom’s Global Content Catalog). When one of the ‘root’ tables has a lot of associated data stored in association tables and that sort of hierarchy, as a best practice, we add a group for the associated, subtending, tables. For example, at Panera there is Cafe and then a group, CafeDetails with a label of ‘-- Details’ so that it looks like a hierarchy under Cafe. In this group are all the association tables and data that lives conceptually under Cafe.

As a best practice, when creating an association field, if creating one with tableRefInverse (the most common, I think?), the association table will be in the same data model. Here is why this is a best practice. It is bad style to create circular dependencies between data models. And since the association table owns the foreign key reference, if that table is not in this data model, it is in a data model that by necessity depends on this data model. By that token, this association must not be owned by this table and should not be added. Since it is a best practice, we assert that all association fields will be to tables in the same data model. Note well, some ps-library functions will depend on this assertion.

# Users/Roles

Tech admin -- like DBA

# Collaboration on projects

Each customer project will have its own eclipse workspace and its own copies of EBX Home and EBX Server (and whatever app server that customer is using). Then there will generally be one additional project for the customer’s modules. This will include the ps libraries.

This customer project should be checked into Git and managed there. Model xsds should be imported by team members whenever a commit has changes to them. At least one model should include the services ‘Import dev artifacts’ and ‘Export dev artifacts’ and this service should be used whenever a commit has changes in those files. Whenever a change is made to the directory or custom views, etc, export should be used.

## Checking in updates to ps library

# Troubleshooting

## Cannot open a dataSet [SqlChecker] error

Sometimes a dataSet becomes inaccessible with this error:

[SqlChecker] Adding a new NOT NULL column into a non-empty table is not supported; at least a default value should be specified.

In the warnings section, you may get a clue which table added such a field (or use the source control history of the xsd, if that is an option), such as:

Modification of primary key for table [/root/cafeDetails/CafeEquipment] can cause validation errors, and may not be accepted by RDBMS.

If you purge/delete history on the data space, turn off history on the offending table, then re-publish the data model, you should be able to get to the data set. Then you can restore historization to the table and republish and all should be well.

1. Turn off history on the table
2. Administration > Actions > Create deletion requests  
   Delete all tables history for changed tables to remove clutter
3. Administration > Actions > Execute purge
4. Turn on history on the table

Best practice: if you need to change/add a pk on a table, follow these steps:

1. Delete the table data
2. Remove table from replication
3. Disable history on table
4. Publish
5. Purge table history
6. Add/change key, re-enable replication/historization
7. Publish