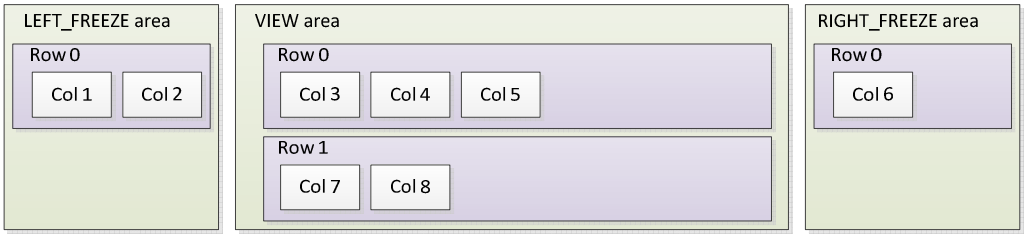
Data model creation for representation of a table column

Create a *TableColumnMetadata* TypeScript class that represents the definition of a table column. The data model must comprise the following information:

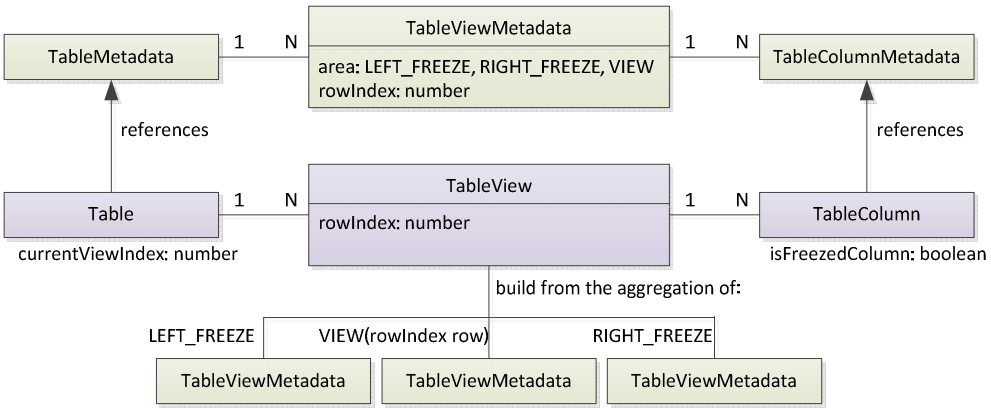
- Column unique identifier
- Preferred width. The width is an abstract and relative value that will be compared with the preferred width of the other columns to compute its real width on the screen
- Type of data that column will display (string, number, date, boolean, other). That information can be used among other things to determine if the table cell value must be left or right aligned
- The area the column belongs to (LEFT_FREEZE, RIGHT_FREEZE, VIEW)
- The index of the row to which the column belongs. LEFT_FREEZE and RIGHT_FREEZE areas have only 1 row.
- Index position of the column within that row



Data model to easy access and manipulates the columns that compose a table

Create a *TableViewMetadata* TypeScript class to reference all the columns (*TableColumnMetadata*) that belongs to a same area and row inside that area.
Create a *TableMetadata* TypeScript class to reference the *TableViewMetadata* dedicated to the LEFT_FREEZE area, the *TableViewMetadata* dedicated to the RIGHT_FREEZE area and a list of *TableViewMetadata*, one for each row of the VIEW area:

The *TableViewMetadata* and *TableMetadata* classes must declare methods to add *TableColumnMetadata* instances. Additional methods will be declared later on when implementing the drag and drop and column resize features.



Instead building the web page directly from the *TableMetadata*, *TableViewMetadata* and *TableColumnMetadata* data model an intermediary layer is used. That intermediary layer can be

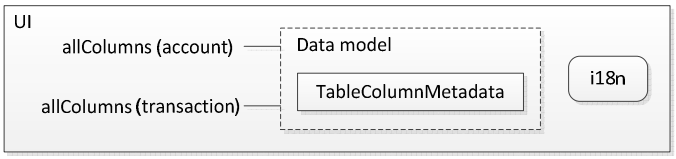
thought as the data model of the widgets that will be used to render the table. The usage of that intermediary layer will become obvious during the implementation of the drag and drop and column resize features.

Create a *TableColumn* TypeScript class that encapsulates a *TableColumnMetadata*. Create a *TableView* TypeScript class that represents a table view as displayed onto the web page. That class references a list of *TableColumn*. That list is built from the aggregation of the *TableColumnMetadata* that compose the LEFT_FREEZE area, that compose a specific row of the VIEW area and that compose the RIGHT_FREEZE.

Finally create a *Table* TypeScript class that encapsulates a *TableMetadata* and that references all the created *TableView* instances. Add methods to instantiate a *Table* instance and all its children from a given *TableMetadata* and all the columns it references. Add a field to indicate the index of the current view to display. Initialize that field with 0 so that the columns of the first *TableView* will be rendered by default.

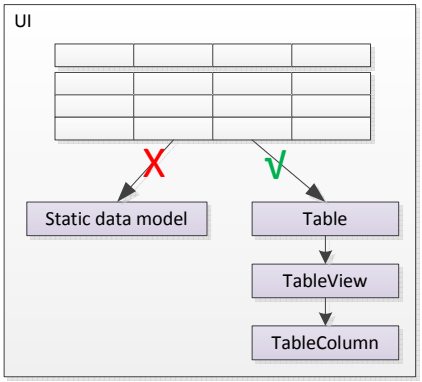
Declaration of all the account and transaction table columns using the new data model

Create instances of *TableColumnMetadata* for each column that currently composes the account summary table and the transaction summary table. For each instance, insert the column label into the i18n file (key=column unique identifier, value=column label)
Group these instances into a set for the account table and into a set for the transaction table.



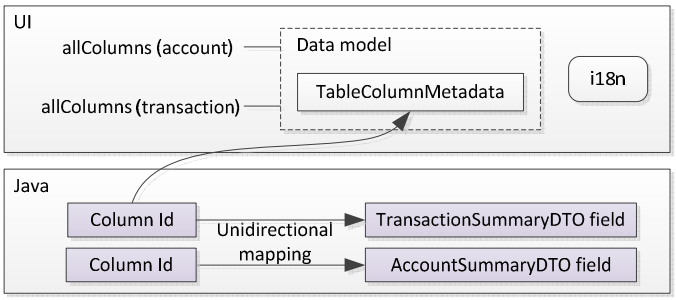
Modify current transaction and account table to use the new data model

Adapt the currently used table widget to use the new data model composed of the *Table*, *TableView* and *TableColumn*. Each panel of the carousel must reference a different *TableView* instance and display the contained columns. The *TableColumn*.isFreezedColumn field indicates if the column is a freezed column and must be displayed with a grey background. The weight of each column can be obtained from the *TableColumnMetadata*.preferredWidth.



Map table columns with the Java data model that hold a transaction or account

For each *TableColumnMetadata* instance declared in the UI, map the column id to a field of the TransactionSummaryDTO and AccountSummaryDTO Java class. These classes are data structures that hold respectively a transaction or an account



Communication of the columns to be included in a CSV export

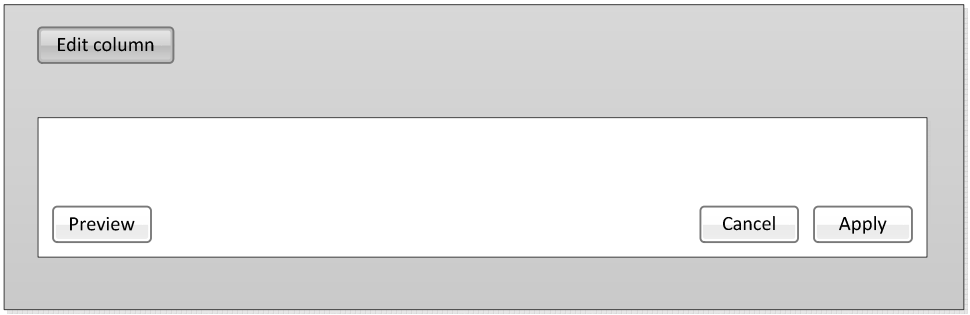
Modification of the /api/transactionexport and /api/accountexport REST services to accept a list of column ids. The column ids identify the columns to be included in the generated CSV file. The list indicates in which order these columns must appear in the generated file. The Java layer must generate an empty file if that list if missing but generate an explicit WARN logging message.

Modification of the UI code to provide that column list. The list must start with the columns contained in the LEFT_FREEZE area, followed by the columns contained in the different rows of the VIEW area, followed by the columns contained in the RIGHT_FREEZE area.

Edit column model window creation

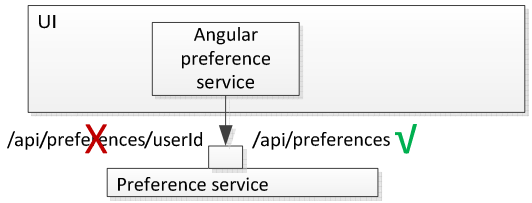
Insert a button labelled *Edit column* or insert an icon button into the transaction and account summary web pages.
Associate a click onto that button with the opening of a modal window.
The modal window has the following characteristics:

- Its width is fixed and equal to 85% of the current screen width
- Its height must guarantee the full display of the window content (thus without displaying a vertical scrollbar)
- It must be horizontally centered
- It must display a Preview, Cancel and Apply buttons at the window bottom
- The modal window must be closed on click on any of these 3 buttons
- The area outside the modal window must be greyed with an opacity of 0.35
- The modal window must be closed on click on the area outside of the window



Integration and adaptation of the Angular preference service

Integrate the current Angular preference service and adapt it to not include the identifier of the logged user in the URL requests. Configure the UI with the URL of the preference service endpoint



Definition of the preference key for the storage of the table column

Preference key proposal: `preference.registry.transaction.custom.myviewname={…}`

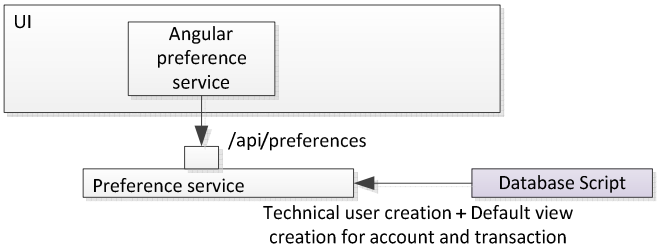
The preference key must identity the application (registry)
The preference key must identify the page that displays the table (transaction)
The preference key must distinguish between pre-defined views and user custom views (custom)
The preference key must include the name given to the view (myviewname)

Default view definition for the transaction and account summary tables

Creation of a script to update the preference database that:

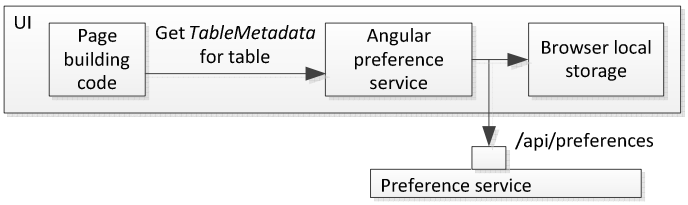
- Creates a technical user under which preference settings to be made available to all the users can be stored
- Creates preference entries for a default view for the transaction and for the account table. The preference value must be set to the JSON serialization of the *TableMetadata* referenced indirectly by the transaction and account tables
  The preference keys are:
  `preference.registry.transaction.predefined.default`
  `preference.registry.account.predefined.default`

Modification of the preference service to return the preference settings of the technical user when these settings are not present for the currently logged user



Retrieval of the view definition for the transaction and account summary table from the Angular preference service

Modification of the transaction and account summary web page controllers to retrieve from the Angular preference service the *TableMetadata* that contains the columns to be displayed in the table. The preference keys to be used for the data retrieval are:
`preference.registry.transaction.predefined.default`
`preference.registry.account.predefined.default`



Available columns widget implementation

- Display a matrix wherein each row is composed of exactly 4 columns. There is no constraint on the number of rows. All the matrix cells have the same width and height.
- The matrix displays the table columns that are not currently in use in the transaction or account summary table.
- These non-used columns are all the columns referenced by the *allColumns* field and which are not referenced in the *TableMetadata* instance used by the transaction or account summary table. These non-used columns are *TableColumnMetadata* instances
- A different matrix cell is used to display the label of a single non-used column
- The matrix must foreseen enough rows to display all the non-used columns
- The non-used columns must be displayed sorted in alphabetical order
- Each non-empty matrix cell is drag-able. During a drag operation the cell style must be modified so that the user can easily identify the dragged cell.
- Each matrix cell is the drop area for a column that the user moved from the *used columns* widget (see later). The drop of a column triggers a re-ordering operation to guarantee that the non-used labels are still alphabetically sorted.
- The matrix must provision an empty cell – a cell that does not display a non-used column. That empty cell will be used as extra dropping area for a column that the user moved from the *used columns* widget. That empty cell helps to improve the understanding of the user that he can drop columns onto this *available columns* widget.