Tracker Tabular

Tracker Tabular is built as an API to import and export tracker information as lists. While the API is used within Tiki, it can also be used to create custom import/export interfaces when appropriate.

Defining a Schema

While building the schema, the process will make sure the field and mode selected exist within the specified tracker definition. The final validation call will make sure that there are no conflicting fields.

Sample schema assembly

```php
$tracker = \Tracker_Definition::get(4);

$schema = new \Tracker\Tabular\Schema($tracker);
$schema->addColumn('itemId', 'id');
$schema->addColumn('sku', 'default');
$schema->addColumn('division', 'text');
$schema->addColumn('supplier', 'lookup-simple');
$schema->addColumn('price', 'formatted');
$schema->addColumn('name', 'default');
$schema->addColumn('introductionDate', 'yyyy-mm-dd');
$schema->addColumn('introductionDate', 'unix')->setReadOnly(true);
$schema->setPrimaryKey('itemId');

$schema->validate();
```

Importing

The process of importing data implies reading the data from a source, and writing it into tracker storage. The following example provides a generic example importing anything into a specific tracker.

You could also use a CsvSource instead of an AutoDetectCsvSource if you want to allow the operation over only a select collection of fields. In this case, you would need to provide the schema to the source. For example, you could have employees only able to update the price of items and a specific category while importing, creating limited privilege imports.

Simplest import

```php
$tracker = \Tracker_Definition::get(4);

$source = new \Tracker\Tabular\Source\AutoDetectCsvSource($tracker, '/tmp/import');

$writer = new \Tracker\Tabular\Writer\TrackerWriter;
$writer->write($source);
```

Exporting

Data Export can be performed on a full tracker directly from the tracker data, or based on a query result. Both sources work using PHP's generators to avoid loading too much data into memory at one time and
reduce required memory usage.

Performance of the query-based export will depend on the used engine. Lucene would be the slowest as all data needs to be re-fetched. MySQL will only re-fetch multi-value fields when they are needed. ElasticSearch will rarely need to re-fetch anything.

Full Tracker Export

```php
$schema = $this->getSchema();

$source = new \Tracker\Tabular\Source\TrackerSource($schema);

$writer = new \Tracker\Tabular\Writer\CsvWriter('php://output');
$writer->sendHeaders();
$writer->write($source);
exit;
```

Query-based export

```php
$schema = $this->getSchema();

$query = \TikiLib::lib('unifiedsearch')->buildQuery([  'type' => "trackeritem",  'tracker_id' => "4",]);
$query->filterCategories("1 or 2 or 3"); // Anything from the Search_Query class

$source = new \Tracker\Tabular\Source\QuerySource($schema, $query);

$writer = new \Tracker\Tabular\Writer\CsvWriter('php://output');
$writer->sendHeaders();
$writer->write($source);
exit;
```

Adding Modes

To be exportable, a field must support the Tracker_Field_Exportable interface. The only requirement is to support the getTabularSchema() method. Here is a sample method from the Drop Down field type.

```php
function getTabularSchema()
{
    $schema = new Tracker\Tabular\Schema($this->getTrackerDefinition());
    $permName = $this->getConfiguration('permName');
    $name = $this->getConfiguration('name');
    $possibilities = $this->getPossibilities();
    $invert = array_flip($possibilities);
    $schema->addNew($permName, 'code')
        ->setLabel($name)
        ->setRenderTransform(function ($value) {
            return $value;
        })
        ->setParseIntoTransform(function (& $info, $value) use ($permName) {
            $info['fields'][$permName] = $value;
        });
    $schema->addNew($permName, 'text')
}```
The defined schema above defines two modes for the same field: code and text. Note that the second one is marked as incompatible with the first one, as when importing they would fight each other.

Each field defines `setRenderTransform` and "parseIntoTransform". The first one takes the value from the source and serializes it. The second one takes the serialized version and converts it to the actual value. The functions are defined as closures, giving them access to the parent scope and the entire field data.

The render transform can receive data from the tracker source or the query source.

- The tracker source will always only provide the $value for the field. $extra will contain the itemId and status keys.
- The query source will provide the exact match key from the result array as $value. $extra will be populated with what is specified by `addQuerySource` calls, allowing to make some values to simpler names. In this case, it is not really required as the text is obtained from a static list. However, in other cases, such as item links, it can avoid making an extra query to fetch it.

Because these functions will be called multiple times, it is worth it to do some of the work up-front, like obtaining the possibilities, and generating the invert map, which will be made available through the **use** statement.

Quite often, the same values will need to be fetched over and over again. Item links often point to a small set of items. However, it is impossible assume that this will always be the case. The `Tracker\Tabular\Schema\CachedLookupHelper` is available to help with the caching process.

- A function can be specified to load a base set of items on demand
- A function can be provided to load each item individually
- Static methods are available to quickly generate lookups and invert lookups from tracker data

The component allows to initialize the behavior from within the `getTabularSchema()` function, supporting both the small set and large set scenarios, without paying any initial costs. The data will only begin loading when needed the first time, hence the lookups needed for export will not be loaded on imports.