ajja Documentation

Release

gocept gmbh
co. kg

March 17, 2016

Contents

CHAPTER 1

Advanced forms in JavaScript

The latest stable version is ; you will find it at github.

ajja is a MIT licensed library, written in JavaScript, to build forms and collections from JSON data directly in the browser.

The name *ajja* is an Hebrew firstname and also abbreviates the basics of this library: **aj**ax and **ja**vascript.

For most developers, building forms is a bothersome, boring and mostly repeating task. *ajja* makes building forms flexible and almost fun. It renders forms in the browser based on a simple JSON datastructure, using advanced technology like HandlebarsJS and KnockoutJS. Field types may be inferred from the data you provide and form fields are saved automatically to the server.

As a developer, all you have to do is include the sources (see *Installation*) and write some JavaScript code:

```
var form = new ajja.Form('form', {save_url: 'save'});
form.load(
    {'firstname': 'Robert', 'is_child': true},
    {'firstname': {'label': 'Name'}, 'is_child': {'label': 'Child?'}}
);
```

CHAPTER 2

Contents

2.1 Introduction

2.1.1 Installation

ajja comes with files ready to be delivered to the browser. There is no need to browserify our code. It is, however, up to you to minify or bundle the library in the course of your deployment workflow.

Installation via Bower (recommended)

The recommended way to install *ajja* is via bower.

```
bower install ajja
```

Manual installation

If you prefer to include *ajja* by hand, please make sure to include the files in the correct order. For the Javascript code, you can copy & paste the following snipped.

```
<script type="text/javascript" src="src/helpers.js"></script>
<script type="text/javascript" src="src/templates.js"></script>
<script type="text/javascript" src="src/collection.js"></script>
<script type="text/javascript" src="src/form.js"></script>
```

You will also need to include the libaries needed by *ajja*. They are listed in the depencencies section in *bower.json*.

Installation via Fanstatic

There is a fanstatic integration package available for your Python backend. Install it via pip.

```
pip install ajja
```

Then, include the resources in your View.

```
from ajja import form
form.need()
```

2.1.2 Migration

From 2.x to 3.0.0

`ajja` now only accepts *precompiled* templates generated via `Handlebars.compile()`. So if you have custom templates that you used with `ajja`, you now must wrap them into a `Handlbars.compile()` call.

Furthermore, to overwrite the standard templates, just add your compiled templates to `ajja.templates['<name_of_the_template>']` or register them with `ajja.register_template('<name_of_the_template>', '<your_html')` (available from version 3.0.1).

The built-in templates were renamed and the `gocept_jsform_templates` namespace was removed. Have a look inside the *templates* folder for the new names.

From 1.x to 2.0.0

We switched the template engine from `jsontemplate` to `Handlebars`. So if you have custom templates, make sure to rewrite them as `Handlebars` templates.

2.2 Usage

2.2.1 Quick Start

The following examples assume that there is a `Message` model on the server with `title` and `description`. We further assume that the model is accessible through a REST API where an HTTP `GET` request for a message's URL, `message/<id>`, returns JSON data representing the current values, while an HTTP `POST` request for `message/<id>` accepts JSON data to update the model.

Note: Field values will be sent to the server one by one as they are edited. This means that if your model applies validation rules that consider multiple fields at the same time, you need to be careful to provide useful default values to make sure that any edits to form fields can be stored even if a model object has just been created.

Rendering a Form

First install *ajja* via *bower* (or *fanstatic* or *manually*)

```
bower install ajja
```

Add a placeholder inside your DOM

```
<div id="form"></div>
```

Initialize the form via *ajja* and load current state from server

```
var form = new ajja.Form("form");
form.load("message/1");
```

The response from the server should look like `{"title": "", description: ""}` and is analysed to create an input field for each attribute. The type of the input field is based on the data type of each attribute and defaults to a simple text input for empty / `null` values.

On `load` the placeholder will be replaced by the following HTML

```
<form method="POST" action id="form" class="ajja form-horizontal">
    <div class="statusarea"></div>
    <div class="field form-group" id="field-title">
        <label for="title" class="col-sm-3 control-label"></label>
        <div class="col-sm-9">
            <input type="text" data-bind="value: title" name="title" class="form-control" value />
        </div>
        <div class="col-sm-offset-3 col-sm-9">
            <div class="help-block error"></div>
        </div>
    </div>
    <div class="field form-group" id="field-desciption">
        <label for="desciption" class="col-sm-3 control-label"></label>
        <div class="col-sm-9">
            <input type="text" data-bind="value: desciption" name="desciption" class="form-control" v
        </div>
        <div class="col-sm-offset-3 col-sm-9">
            <div class="help-block error"></div>
        </div>
    </div>
</form>
```

Each input field contains a Knockout binding via `data-bind="value: {{name}}"` to track changes. Those changes are pushed to the server by a `POST` request to `message/id` on focus-out. There is a defined communication protocol that the server end point at `message/id` needs to implement.

If server-side validations result in an error, a flash message will be rendered inside `<div class="statusarea"></div>`. If the response contained a `msg` it will be displayed inside `<div class="help-block error"></div>` beneath the input field that was just saved.

As you can see the generated HTML contains CSS classes compatible with Bootstrap, thus including the Bootstrap CSS is enough to make this form look pretty.

Customizing form fields

If you want to display a label next to each input field, declare `title` as required and to use a textarea for `description`, you can call `form.load` with an additional options dict like

```
var form = new ajja.Form("form")
form.load("message/1", {
    title: {"label": "Title", "required": true},
    description: {"label": "Body", "template": "form_text"}
});
```

Initializing form without AJAX request

Instead of loading data from an REST endpoint you can also provide the JSON data directly to the `load` function

```
var form = new ajja.Form("form")
form.load(
    {"title": "My brand new form", "description": ""},
    {
        title: {"label": "Title", "required": true},
        description: {"label": "Body", "template": "form_text"
    }
});
```

Note, that you will need to provide a *save url* in order to make the automatic pushes on field change work.

Rendering a Collection

It is assumed, that you already *installed ajja*. Add a placeholder inside your DOM

```
<div id="my_collection"></div>
```

Initialize the collection (in this case a *ListWidget()*) and load current state from server

```
var collection = new ajja.ListWidget(
    '#my_collection',
    {collection_url: '/messages.json',
     default_form_actions: [],
     form_options: {
        'title': {label: 'Title'},
        'description': {label: 'Body'}
}});
collection.reload();
```

The response from the server is described in the *protocol section*.

On `reload` the placeholder will be replaced by the following HTML

```
<ul id="collection" class="list-group list-collection nav nav-stacked">
    <li id="item_" style="min-height: 50px;" class="list-group-item">
        <span class="actions btn-group badge">
            <a href="#" class="edit btn btn-default btn-xs" data-action="edit">
                <span class="glyphicon glyphicon-edit"></span> Edit</a>
            <a href="#" class="del btn btn-default btn-xs" data-action="del">
                <span class="glyphicon glyphicon-trash"></span> Delete</a>
        </span>
        <span class="content">
            <dl>
                <dt>title</dt>
                <dd>The title</dd>
                <dt>description</dt>
                <dd>The description</dd>
            </dl>
        </span>
    </li>
</ul>
<div id="form-actions">
    <a href="#" class="btn btn-default btn-sm add">
        <span class="glyphicon glyphicon-plus"></span> Add
    </a>
</div>
```

Each item has two default actions: `edit` and `delete`. The collection has the default action `add`. Add and edit both create a bootstrap modal dialog containing a *ajja.Form* form.

As you can see the generated HTML contains CSS classes compatible with Bootstrap, thus including the Bootstrap CSS is enough to make this form look pretty.

The *form_options* argument can be used the same way as *options* for a *Form()* to customize the look and behaviour of the form that is used for adding and editing collection items.

Collection types

ListWidget The *ListWidget()* renders items as HTML lists. List items are rendered as HTML definition lists. Please refer to the section *collection initialization* for details about the default list widget.

GroupListWidget The *GroupListWidget()* behaves similar to the *ListWidget* except that it groups items by a defined attribute.

```
var collection = new ajja.GroupListWidget(
    '#my_collection',
    {group_by_key: 'title',
     group_title_key: 'title',
     collection_url: '/messages.json',
     default_form_actions: [],
     form_options: {
        'title': {label: 'Title'},
        'description': {label: 'Body'}
}});
collection.reload();
```

Groups are created dynamically and items sorted into those groups by *group_by_key*. The title for the groups is taken from the attribute *group_title_key*.

TableWidget The *TableWidget()* renders items in a HTML table.

```
var collection = new ajja.TableWidget(
    '#my_collection',
    {collection_url: '/messages.json',
     default_form_actions: [],
     form_options: {
        'title': {label: 'Title'},
        'description': {label: 'Body'}
}});
collection.reload();
```

Customizing the HTML output

It is possible to change the rendered HTML by overriding the default templates. Please refere to *ajja.register_template()* for information about how default templates are customized.

The following default templates are used by *ListWidgets*:

list The main template for the list collection.

list_item_wrapper Wrapper template for each item of the collection.

list_item Template for the content of an item.

list_item_action Template for an item action (edit, delete).

list_item_edit Template for add or edit form (modal dialog) of an item.

GroupListWidgets use these templates in addition:

group The main template for the group collection.

group_item Template for a group item. Contains one *ListWidgets*.

TableWidgets just use these templates:

table The main template for a table collection.

table_head Template for head part of the table.

table_row Template for a row of a table. Contains data and actions.

list_item_edit The same template as for *ListWidgets*.

2.2.2 Server protocol

Communication between server and client consists of asynchronous calls to load and save data. The server is expected to provide a RESTful interface using resource-oriented URLs that can be accessed by HTTP methods GET, POST, PUT and DELETE, respectively. Data serialisation for transport uses the JSON format.

Loading data

Data can be loaded in one of two ways: by requesting it from the resource URL, or by passing it as an argument to the form object upon initialisation.

Either way, the data format is a mapping of field names to values. Values may be any data type known to the JSON format, or appropriate serialisations that can be JSON-encoded, see *the explanation of datatypes*. If the form object is initialised with a URL instead of a field-value mapping, an XHR GET request is sent to the URL and the full response is expected to be the JSON-encoded field-value mapping:

```
{firstname: "Jack", lastname: "", send_notifications: true, messages: 0}
```

Storing changes

Request

With the current client implementation, each field will send its own value to the server separately after it lost focus. To do so, a JSON-encoded mapping of field name to value (or an appropriate representation thereof) is sent to the resource URL by an XHR POST request:

```
{firstname: "John"}
```

In fact, this is an instance where the current implementation doesn't strictly assume a RESTful URL layout; there may be different URLs for loading and saving form data. The URL for saving is specified by the `save_url` form option; if the form object was initialised with a load URL instead of initial data, the `save_url` option may be omitted and will default to the load URL. The latter special case of a single resource URL does imply a RESTful server.

Since the request body contains a mapping from field name to value, this allows for more than one field value to be stored at the same time. While the stock implementation doesn't store the value of more than one field through the same request, custom widgets may do so; also, a *CSRF token* may be sent along under a configurable name. A server implementation must assume any number of fields to be sent within one save request.

Response

The HTTP response code may be 200 if everything went fine technically (even though validation errors may have occurred), or some error code if a server or connection error occurred. In that case, a notification is displayed to the user and the save request is queued to be retried later.

The body of a response with status 200 must be a JSON-encoded mapping that contains at least a status flag, signifying whether validation succeeded. Upon success, the minimal body should read:

```
{status: "success"}
```

In the case of validation errors, the response is expected to convey an error message suitable to present to the user, e.g.

```
{status: "error", msg: "Not a valid email address"}
```

The error message will be displayed by the widget that initiated the save request.

Updating sources

Furthermore, the response body for a successful save request may contain updated sources for object fields within the same form. This is to help with interdependent widgets where the value of a field determines the set of possible choices of one or more dependent fields.

If a saved value results in updates to any sources for other fields within the form, the response body contains another key `sources` whose value is a mapping from field names to new source values. Source values have the same format (lists of objects having a token and title) as when *configuring them statically through the form options*:

```
{status: "success",
 sources: {subcategories: [
     {token: "sub-a1", title: "Subcategory A.1"},
     {token: "sub-a2", title: "Subcategory A.2"}
 ]}
}
```

Collections

The data format to load a collection is a list of items. Items are objects containing a resource url, which points to the RESTful interface of that item, and an object with item data, that will be displayed inside the collection:

```
[
    {resource: 'message/1',
     data: {'title': 'The title', 'description': 'The description'},
    {resource: 'message/2',
     data: {'title': 'Another title', 'description': 'Another description'}
]
```

The collection protocol is identical for all collection widgets.

2.2.3 Datatypes

The type of the input field is based on the data type of each attribute and defaults to a simple text input for empty / null values. The following data types are recognized out of the box:

string If the value for a field is a string, an input field of type *text* is rendered (Template-ID: *form_string*).

boolean If the value for a field is boolean, a an input field of type *checkbox* is rendered (Template-ID: *form_boolean*).

number If the value for a field is a number, an input field of type *number* is rendered (Template-ID: *form_number*).

object If the value for a field is an object, a selectbox is rendered. You will need to *define a source* for this type as well. (Template-ID: *form_object*).

You can customize the templates that are rendered for you fields by *providing a template for your field or overwriting the default templates*.

Defining a source for objects

Primitive datatypes like strings or boolean values can be easily handeled with the above auto recognition. For more complex types like objects you need to define a source from which the rendered selectbox gets its values. Not all values are submitted on field save but the token of the selected value and values accordingly for multiselect. An example will demonstrate the behaviour.

```
var form = new ajja.Form("form");
form.load(
    {title: []},
    {save_url: 'message/1',
     title: {
        label: "Title",
        source: [{token: 'mr', title: 'Mr.'},
                 {token: 'mrs', title: 'Mrs.'}]}});
```

Its also possible to define a multiselect field. Just pass the attribute *multiple* to the fields options.

```
var form = new ajja.Form("form");
form.load(
    {title: []},
    {save_url: 'message/1',
     title: {
        label: "Title",
        source: [{token: 'mr', title: 'Mr.'},
                 {token: 'mrs', title: 'Mrs.'}],
        multiple: true}});
```

If the possible values of a selection depend on the value of another field (subcategories of a category that can be selected from a choice, or a list of search results for a free-text search box), updated source lists can be returned with the server response to the respective save requests, e.g. the requests to save a new supercategory or a new search term. See also *the specification of the server protocol*. This is a generic way to avoid preloading a huge amount of possible source data (which may not even be possible, as in the search-box example) as well as complex update logic inside client code.

Rendering a Yes/No template for boolean fields

By default, a boolean field is rendered as a checkbox. A selected checkbox represends the value true, a not selected one the value false. By using an object template one can get a boolean field which has two radio buttons for Yes and No (or whatever you want to expess with the field).

```
var form = new ajja.Form("form"),
    source = [{token: 'true', title: 'Yes'},
              {token: 'false', title: 'No'}];
form.load(
    {sent: 'false'},
    {save_url: 'message/1',
     sent: {
        label: "Was the message sent already?",
        source: source,
        template: 'form_radio_list'}});
```

2.2.4 Customization

There are various options which can be passed to customize the HTML output and the behaviour of *ajja*.

Providing a save url for the server

One great feature of *ajja* is that it automatically pushes changes in your form fields to the server. By default, changes are sent to the same url that form data was loaded from. You can also specify a separate url to save to:

```
var form = new ajja.Form('form', {save_url: '/save.json'});
```

The server end point at `save_url` is expected to implement ajja's communication protocol.

Customizing the form template

The default behaviour is to simply append every new field as a child to the form tag. If you need to customize the order of your fields or just need different overall HTML for your form, you can use a custom form template with containers for all or just some of the fields:

```
ajja.register_template(
  'form',
  ['<form method="POST" action="{{action}}" id="{{form_id}}" class="ajja form-horizontal">',
   '  <div class="statusarea"></div>',
   '  <table><tr><td><span id="field-firstname" /></td>',
   '  <td><span id="field-lastname" /></td></tr></table>',
   '</form>'].join('')
);

var form = new ajja.Form('form');
form.load({firstname: 'Max', lastname: 'Mustermann'});
```

This will replace the `span` containers with ids `firstname` and `lastname` with the appropriate `input` fields.

CSRF token

In order to prevent Cross-site request forgery (CSRF), *ajja* can handle CSRF tokens and always submit them with every save request. The token needs to be generated on the server and injected into the DOM in a hidden input field with the id `csrf_token`.

```
<input type="hidden" id="csrf_token" value="secure-random" />
<div id="form"></div>
```

The token will be sent with every request under the key `csrf_token`.

Customizing field widgets

You can either customize widgets by their type (e.g. all fields rendered for strings) or customize single widgets by their name.

Customization by field type

You can overwrite the default templates by registering your own templates prior to form initialization:

```
ajja.register_template('form_boolean', '<bool_template_html />');
ajja.register_template('form_string', '<string_template_html />');
var form = new ajja.Form('form');
```

For every string value, your input template would be rendered instead of the default input text field. Same for lists and boolean values.

Customization by field name

Imagine you want checkboxes instead of a select field. You can use the *form_radio_list* template for that purpose:

```
var form = new ajja.Form('form');
form.load({kind: ''},
          {kind: {source: [
                      {token: 'dog', title: 'Dog'},
                      {token: 'cat', title: 'Cat'},
                      {token: 'mouse', title: 'Mouse'}],
                   label: 'What kind of animal do you love?',
                   template: 'form_radio_list'}});
```

You can pass the *load* method a JS object containing customizations for each field. One of these customization options is the name of the registered template, which results in rendering two checkboxes instead of the default select box.

Rendering readonly widgets

If you need to make a field widget immutable, you can pass it the *disabled* flag in the options:

```
var form = new ajja.Form('form');
form.load({kind: 'Immutable'},
          {kind: {label: 'Immutable text',
                  disabled: true}});
```

Its possible to render the whole form with immutable fields, too:

```
var form = new ajja.Form('form', {disabled: true});
form.load({name: 'John Doe', gender: 'male'},
          {name: {label: 'Name'},
          gender: {label: 'Gender',
                    'source': [{token: 'unknown', title: 'Not specified'},
                               {token: 'male', title: 'Male'},
                               {token: 'female', title: 'Female'}]}});
```

2.2.5 Internationalization

***ajja* supports i18n, but only translations for english (default) and** german are provided. Therefore we seek your help: To add a new language fork ajja on GitHub and add a new file to src/localizations. To enhance predefined translations, extend existing files of that folder.

To use a predefined localization, initialize *ajja* with the `language` option:

```
var form = new ajja.Form("form", {language: "de"});
form.load({topic: 'Internationalization'}, {topic: {label: 'Topic'}});
```

This way all notifications and hints will be rendered in german, instead of english. The language is chosen via it's file name, so `language: "de"` will effectively read `src/localizations/de.js` for translations.

2.2.6 Behaviour on Connection Issues

If for any reason the server does not save requests made by *ajja* (e.g. because there was a connection issue), those saves are retried once the server is handling saves again. This happens as soon as the first save was successful.

Furthermore, when saving the whole form all fields that were not saved on change are saved immediatly. This makes sure that there are no field validation errors left to be corrected by the user. There won't be a "save all" request containing all field values. The backend must make sure that the single requests for each field are saved.

2.3 API

2.3.1 Form

`$.fn.form_submit_button()`
Make a form submit button an ajax submit button. This makes sure that when clicking submit, all fields are saved via ajax.

```
$('#form input[type=submit]').form_submit_button()
```

class `Form` (*id*[, *options*])

Extends *TemplateHandler()*

Arguments

- **id** (*string*) – The id of the DOM node where the form should be rendered.
- **options** (*Object*) – An object containing options for the form.
- **options.save_url** (*string*) – The url where data changes are propagated to. Should return a dict with either `{"status": "success"}` or `{"status": "error", "msg": "Not an eMail address."}`.
- **options.action** (*string*) – The url the form will submit to (if intended). Will become the action attribute in form.
- **options.language** (*string*) – 2-char language code. Default is *en*.
- **options.disabled** (*boolean*) – Only render disabled fields in the whole form if true.

Returns The form instance.

Return type Object

```
$(body).append('<div id="form"></div>');
var form = new ajja.Form('form');
```

alert (*msg*)
Show a message to the user. (Alert box)

Arguments

- **msg** (*string*) – The message to display.

Return type void

clear_field_error (*name*)
Clear announcement of an field error during save.

Arguments

- **name** (*string*) – The name of the field.

clear_saving (*name*, *msg_node*)
Clear announcement of save progress for a given field.

Arguments

- **name** (*string*) – The name of the field.
- **msg_node** (*string*) – The node where a saving progess message is displayed.

clear_server_error()
Clear any announcement of an HTTP fault during an ajax call.

clear_status_message(*msg_node*)
Clear given status message.

Arguments

- **msg_node** (*Object*) – DOM Node as returned by *status.message*.

collect_sources()
Collect sources from options and make them ovservable.

Return type void

create_form()
Wires the form DOM node and object.

Return type void

create_model()
Create a knockout model from self.data.

Note: Needed for bindings and oberservation.

---

expand_form()
Expands the form_template into the DOM.

Return type void

field(*name*)
Get the DOM node for a field.

Arguments

- **name** (*string*) – The name of the field.

Returns The DOM node of the field as a jQuery object.

Return type Object

finish_load(*tokenized*)
After load handler. Save data retrieved from server on model.

Arguments

- **tokenized** (*Object*) – The data returned from the ajax server request.

get_widget(*name*)
Retrieve the widget for a field.

Arguments

- **name** (*string*) – The name of the field.

highlight_field(*name*, *status*)
Highlight field with status.

Arguments

- **name** (*string*) – The name of the field.

- **status** (*string*) – The status to display. Should be one of 'success', 'info', 'warning' or 'danger'.

init_fields()
Initialize fields from self.data.

Note: Guess the type of data for each field and render the correct field template into the DOM. Invoke the knockout databinding via auto-mapping data into a model (thanks to ko.mapping plugin) and invoke observing the model for changes to propagate these to the server. Appends fields into the form if no DOM element with id name like field is found.

---

is_object_field(*name*)
Check whether field is an object field.

Note: Object fields are either select boxes or radio lists.

Arguments

- **name** (*string*) – The name of the field to check.

Return type boolean

label(*name*)
Return the label for a field.

Arguments

- **name** (*string*) – The name of the field.

Returns The label of the field.

Return type string

load(*data_or_url*[, *options*[, *mapping*]])
Invokes data retrieval and form field initialization.

Arguments

- **data_or_url** (*string*) – The url to a JSON View returning the data for the form or the data itself.
- **options** (*Options*) – Options for each data field.
- **options.<field_name>** (*string*) – Options for the field.
- **options.<field_name>.label** (*string*) – The label of the field.
- **options.<field_name>.template** (*string*) – The id of a custom template for this field.
- **options.<field_name>.required** (*boolean*) – Whether this is a required field or not.
- **options.<field_name>.source** (*Array*) – The source for a select field. Contains objects with 'token' and 'title'.
- **options.<field_name>.multiple** (*boolean*) – For object selection, whether to do multi-select.

- **options.<field_name>.placeholder** (*string*) – Placeholder to the empty dropdown option.
- **options.<field_name>.disabled** (*boolean*) – true if field should be disabled.
- **mapping** (*Object*) – An optional mapping for the <ko.mapping> plugin.

Return type void

```
form.load({'firstname': 'Robert', 'is_baby': true});
form.load('/data.json', {is_baby: {label: 'Is it a baby?'}});
```

notify_field_error(*name*, *msg*)
Announce error during save of field.

Arguments

- **name** (*string*) – The name of the field.
- **msg** (*string*) – The message to announce.

notify_saving(*name*)
Announce that save of a field is in progress.

Arguments

- **name** (*string*) – The name of the field.

notify_server_error()
Announce HTTP faults during ajax calls.

observe_model_changes()
Observe changes on all fields on model.

reload()
Invokes data retrieval from server and reloads the form.

render_widget(*id*)
Render one form widget (e.g. an input field).

Arguments

- **id** (*string*) – The name of the field.

resolve_object_field(*name*, *value*)
Save tokens from value in object fields.

Arguments

- **name** (*string*) – The name of the field.
- **value** (*Array|string*) – Tokens from object field if multiple of one token.

Returns Returns either an array of source objects if field is multiple or exactly one source object.

Return type Array|Object

retry()
Retry saving the form.

save(*name*, *newValue*[, *silent*])
Schedule saving one field's value to the server via ajax.

Arguments

- **name** (*string*) – The name of the field.
- **newValue** (*string*) – The new value of the field.

- **silent** (*boolean*) – Do not notify the user about saving field.

save_and_validate(*name*, *newValue*)
Validation of the field and newValue

Arguments

- **name** (*string*) – The name of the field.
- **newValue** (*string*) – The new value of the field.

Returns A jQuery promise.

Return type Object

save_remaining()
Save all fields that were not saved before.

Note: Fields are saved silently.

---

start_load()
Invokes data retrieval if needed.

Note: After retrieval (which may be asynchronous), self.data is initialized.

---

start_save(*name*, *newValue*[, *silent*])
Actual work of preparing and making the ajax call.

Note: May be deferred in order to serialise saving subsequent values of each field.

Arguments

- **name** (*string*) – The name of the field.
- **newValue** (*string*) – The new value of the field.
- **silent** (*boolean*) – Do not notify the user about saving field.

Returns A jQuery promise.

Return type Object

status_message(*message*, *status*, *duration*)
Create a status message for the given duration.

Arguments

- **message** (*string*) – The message to display.
- **status** (*string*) – The status to display. Should be one of 'success', 'info', 'warning' or 'danger'.
- **duration** (*number*) – How long should the message be displayed (in milliseconds)

Returns The created message as jQuery DOM node.

Return type Object

subscribe(*name*)
Subscribe to changes on one field of the model and propagate them to the server.

Arguments

- **name** (*string*) – The name of the field.

t (*msgid*)
Translate a message into the language selected upon form initialization.

Arguments

- **msgid** (*string*) – The message id from the localization dict.

Returns The translated message.

Return type string

tokenize_object_fields (*name*, *value*)
Get tokens from value in object fields.

Arguments

- **name** (*string*) – The name of the field.
- **value** (*Array|string*) – The selected values if field is multiple else the selected value.

Returns The selected tokens if field is multiple else the selected token.

Return type Array|string

update_bindings ()
Add or update knockout bindings to the data.

Note: This is where all the magic starts. Adding bindings to our model and observing model changes allows us to trigger automatic updates to the server when form fields are submitted.

---

update_sources (*data*)
Update sources from data. Called on form reload.

Arguments

- **data** (*Object*) – The data returned from the ajax server request.

when_saved (*retry*)
Handle save retries if connection to server is flaky or broken.

Arguments

- **retry** (*boolean*) – Chain retries? (default: true)

Returns A jQuery promise.

Return type Object

2.3.2 Template

ajja.**register_template** (*id*, *template*, *description*)
Allows you to register your templates or change the default templates.

Arguments

- **id** (*string*) – The id of the template.

- **template** (*string|function*) – The template. Will be saved as a compiled Handlebars template. Can be a precompiled Handlebars template, the template as raw HTML or the id of a DOM node containing the HTML of the template.
- **description** (*string*) – A description for the template.

Return type void

```
ajja.register_template('my_template', '<dl><dt>{{name}}</dt><dd>{{value}}</dd></dl>');
var form = new ajja.Form('form');
form.load({title: 'Sebastian'}, {title: {template: 'my_template'}});
```

```
$('body').append(
    '<script type="text/html" id="reference"><b>{{value}}</b></script>'
);
ajja.register_template('my_template', '#reference');
var form = new ajja.Form('form');
form.load({title: 'Sebastian'}, {title: {template: 'my_template'}});
```

```
var compiled = Handlebars.compile('<p>{{value}}</p>');
ajja.register_template('my_template', compiled);
var form = new ajja.Form('form');
form.load({title: 'Sebastian'}, {title: {template: 'my_template'}});
```

class TemplateHandler()

Helper class for handling templates within *ajja*.

```
var handler = new ajja.TemplateHandler();
```

get_template(*id*)

Get the template for the given *id*.

Arguments

- **id** (*string*) – The id of the template.

Returns The template as precompiled Handlebars template.

Return type function

```
handler.get_template('form_boolean')({name: 'asdf'})
'<input type="checkbox" name="asdf" data-bind="checked: asdf" />'
```

list_templates()

List all registered templates.

Returns A list of objects containing the id, description and compiled template.

Return type Array

```
handler.list_template()[0]
{id: 'form', description: 'The base `ajja.Form` template', template: function}
```

2.3.3 Collection

class GroupListWidget(*node_selector*, *options*)

Group items of a list by class written in data attributes.

Note: Each list item must provide `data-{{self.group_by_key}}` and `data-{{self.group_title_key}}`. Those are needed to decide, in which group the item is placed and what title that group will get.

Extends *ListWidget()*

Arguments

- **node_selector** (*string*) – The selector of the DOM node where the widget should be rendered.
- **options** (*Object*) – An object containing options for the widget.
- **options.group_by_key** (*Array*) – By which data-key are items grouped.
- **options.group_title_key** (*Array*) – Specify what key leads to the title of the group.
- **options.[collection_url]** (*string*) – The url to a JSON View returning the data for the collection.
- **options.[form_options]** (*Object*) – An object containing options for the edit *Form()* as described under *load()*.
- **options.[item_actions]** (*Array*) – Additional item_actions besides *edit* and *del*.
- **options.[default_item_actions]** (*Array*) – Set to an empty Array to hide *edit* and *del*.
- **options.[form_actions]** (*Array*) – Additional form_actions besides *add*.
- **options.[default_form_actions]** (*Array*) – Set to an empty list to hide *add*.

Returns The widget instance.

Return type Object

```
$(body).append('<div id="my_list_widget"></div>');
var list_widget = new ajja.GroupListWidget(
    '#my_list_widget',
    {collection_url: '/list.json'}
);
```

get_collection (*item*)

Return the container DOM node of item.

Note: The grouping is done here on the fly.

Arguments

- **item** (*Object*) – An item as returned by the collection JSON view.

Returns jQuery DOM node to items container.

Return type Object

class ListWidget (*node_selector*, *options*)

Turn any DOM elements matched by node_selector into ListWidgets.

Extends *TemplateHandler()*

Arguments

- **node_selector** (*string*) – The selector of the DOM node where the widget should be rendered.
- **options** (*Object*) – An object containing options for the widget.
- **options.collection_url** (*string*) – The url to a JSON View returning the data for the collection.
- **options.form_options** (*Object*) – An object containing options for the edit *Form()* as described under *load()*.
- **options.[item_actions]** (*Array*) – Additional item_actions besides *edit* and *del*.
- **options.[default_item_actions]** (*Array*) – Set to an empty Array to hide *edit* and *del*.
- **options.[form_actions]** (*Array*) – Additional form_actions besides *add*.
- **options.[default_form_actions]** (*Array*) – Set to an empty list to hide *add*.

Returns The widget instance.

Return type Object

```
$(body).append('<div id="my_list_widget"></div>');
var list_widget = new ajja.ListWidget(
    '#my_list_widget',
    {collection_url: '/list.json'}
);
```

add_item()
Add an new item to the collection.

apply_item_actions(*node*)
Bind a click handler to each action of the given item.

Note: The callback, that was specified in *item_actions*, is binded here.

Arguments

- **node** (*Object*) – The jQuery DOM node of the item with the actions.

close_object_edit_form(*ev*, *object_form*, *form_dialog*)
Handler, that closes the edit form after save or cancel.

Arguments

- **ev** (*Object*) – The close event.
- **object_form** (*Object*) – The *Form()* instance.
- **form_dialog** (*Object*) – The jQuery DOM node of the form.

del_item(*node*)
Delete an item from the collection.

Arguments

- **node** (*Object*) – The jQuery DOM node of the item to be deleted.

edit_item(*node*)
Render an edit *Form()* and provide it to the user.

Note: The {FormOptions} object provided on initialization of the ListWidget is used to render the form.

Note: Only fields with a label (provided in {FormOptions}) are rendered in this form.

Arguments

- **node** (*Object*) – The jQuery DOM node pointing to the item.

get_collection(*item*)
Return the container DOM node of item.

Arguments

- **item** (*Object*) – An item as returned by the collection JSON view.

Returns jQuery DOM node to items container.

Return type Object

get_collection_head(*items*)
Return the rendered HTML of the widgets header.

Arguments

- **items** (*Array*) – The items as returned by the collection JSON view.

Returns HTML ready to be included into the DOM.

Return type string

reload()
Reload the widget. Retrieve data from the server and render items in DOM.

Returns The widget instance.

Return type Object

render(*items*)
Render items in the DOM.

Arguments

- **items** (*Array*) – The items as returned by the collection JSON view.

Return type eval

render_form_actions()
Render the form actions and bind a click handler to them.

render_item(*item*)
Render an item into the DOM.

Arguments

- **item** (*Object*) – An item as returned by the collection JSON view.

Returns jQuery DOM node to the rendered item.

Return type Object

render_item_content(*node*)
Render the content of an item (the part next to the actions in DOM).

Arguments

- **node** (*Object*) – The jQuery DOM node pointing to the item.

class TableWidget(*node_selector*, *options*)
Show list of items in a table.

Extends *ListWidget()*

Arguments

- **node_selector** (*string*) – The selector of the DOM node where the widget should be rendered.
- **options** (*Object*) – An object containing options for the widget.
- **options.collection_url** (*string*) – The url to a JSON View returning the data for the collection.
- **options.form_options** (*Object*) – An object containing options for the edit *Form()* as described under *load()*.
- **options.[omit]** (*Array*) – Specifiy attributes taht should not be rendered as columns in the table.
- **options.[item_actions]** (*Array*) – Additional item_actions besides *edit* and *del*.
- **options.[default_item_actions]** (*Array*) – Set to an empty Array to hide *edit* and *del*.
- **options.[form_actions]** (*Array*) – Additional form_actions besides *add*.
- **options.[default_form_actions]** (*Array*) – Set to an empty list to hide *add*.

Returns The widget instance.

Return type Object

```
$(body).append('<div id="my_list_widget"></div>');
var list_widget = new ajja.TableWidget(
    '#my_list_widget',
    {collection_url: '/list.json'}
);
```

get_collection_head(*items*)
Return the rendered HTML of the widgets header.

Note: Only fields with a label (provided in {FormOptions}) are returned as columns of the table.

Arguments

- **items** (*Array*) – The items as returned by the collection JSON view.

Returns HTML ready to be included into the DOM.

Return type string

render_item(*item*)
Render an item into the DOM as a table row.

Arguments

- **item** (*Object*) – An item as returned by the collection JSON view.

Returns jQuery DOM node to the rendered item.

Return type Object

translate_boolean_cells(*node*)
Render boolean cells as proper glyphicons.

Arguments

- **node** (*Object*) – The jQuery DOM node pointing to the table row.

2.3.4 Helpers

ajja.**declare_namespace**(*value*)
Helper to declare namespaces (e.g. *ajja*).

Arguments

- **value** (*string*) – The namespace to be declared.

```
ajja.declare_namespace('ajja');
ajja.foo = 'bar';
```

ajja.**isUndefinedOrNull**(*value*)
Check whether value is undefined or null.

Arguments

- **value** (***) – A value.

Return type boolean

```
ajja.isUndefinedOrNull('foo');
```

ajja.**or**(*value1*, *value2*)
Simple OR function. Returns `value1` if its defined else `value2`.

Arguments

- **value1** (***) – A value.
- **value2** (***) – A value.

Returns value1 or value2

Return type

-

```
ajja.or(null, 'asdf');
```

2.4 Contributing

All open-source projects live through their community, so does *ajja*. You are very welcome to contribute bug fixes and feature proposals. Please note the following guidelines:

- *ajja* is hosted on github: http://github.com/gocept/ajja

- Use github's issue tracker to submit bug fixes or feature proposals.
- Whenever possible, include a test demonstrating the issue.
- It is good practice to fork a project and make your changes through your own fork.
- Always send a pull request with a reference to the github ticket.
- Include enough information within your commit messages and pull request to help us understand your changes.

Don't be discouraged if a pull or feature request gets rejected. We will always provide helpful comments or information explaining any issues with your code and how to fix them.

A good and welcome start working on the project would be fixing or extending the documentation.

2.5 Changelog

2.5.1 4.0.1 (2016-03-17)

- Fix wrong dependency url in *bower.json*.

2.5.2 4.0.0 (2016-03-17)

- *gocept.jsform* has a new name: *ajja*
- There are also some API renamings that are incompatible to older versions:
  - The base CSS class name for ajja forms is now *ajja* and not *jsform*.
  - The jQuery function to initialize a submit button was renamed from *jsform_submit_button()* to *form_submit_button*.
  - If you include *ajja's sources manually (which is not recommended), make sure to now include 'form.js* instead of *jsform.js*.

2.5.3 3.0.1 (2016-03-17)

- Introduce new API for template handling.
  - *get_template(id)* returns the compiled version of the template.
  - *register_template(id, template, description)*
    - allows you to register a template under the given id
    - can handle precompiled templates, plain html or an id selector to a dom node containing html
  - *list_templates()* returns a list of templates with id and description
- Massively improved documentation.
- Start measuring code coverage.

2.5.4 3.0.0 (2016-02-03)

- Serve handlebars templates precompiled in src/templates.js to complete bower integration. (#23)

  Rename template source files. They are now accessible via *gocept.jsform.templates* variable in the application. This is an backwards incompatible change.

- Update version pinnings.

  Depending on concrete pinnings for jquery(-ui) is not necessary as gocept.jsform supports jquery 1.x and 2.x. Also add explicit version pinnings for knockout and knockout-mapping as updating there by accident may break gocept.jsforms model behaviour.

2.5.5 2.8.0 (2015-12-09)

- Add group widget and table widget which use list widget to render items in groups / as table. (#38)
- Add new template to render numbers. (#15)
- Throw an error during __init__ if selector for form node did not match. (#32)
- Throw error when ListWidget.add_item receives malformed AJAX response. (#33)
- Add *form-control* CSS class to textarea template.

2.5.6 2.7.2 (2015-12-04)

- Refactor radio button template to use the same source API as object templates.

2.5.7 2.7.1 (2015-12-04)

- Fixed syntax error.

2.5.8 2.7.0 (2015-12-04)

- Suppress success messages in UI when saving all remaining fields.
- Add new template than renders a list of radio buttons.

2.5.9 2.6.3 (2015-11-18)

- Add some more translations.
- Disable placeholder for select fields which are required.

2.5.10 2.6.2 (2015-11-17)

- Unify node where form data is saved to.

2.5.11 2.6.1 (2015-11-16)

- Fix initialization of sources if form data is provided directly instead of retrieving it via load url.
- Make sure that *after-load* always fires asynchonously.

2.5.12 2.6.0 (2015-11-12)

- Add option to switch one field or the whole form to readonly mode.

2.5.13 2.5.1 (2015-11-11)

- Use *bower-knockout-mapping* in bower package instead of serving mapping plugin directly. (only effects bower package of *gocept.jsform*)

2.5.14 2.5.0 (2015-11-06)

- Add list widget which uses jsform to display edit form of list items.
- Remove version pin of Handlebars, since error related to Handlebars 4 is specific to the application that uses gocept.jsform.

2.5.15 2.4.0 (2015-09-08)

- Pin version of Handlebars to 3.0.3, since switching to a new major version should be done on purpose, rather by chance.

2.5.16 2.3.0 (2015-07-31)

- Display status messages 3 seconds instead of 1 second. This hopefully will fix test failures in Jenkins.
- Introduced, that a save call to a widget can manipulate the source values of another widget. This is usefull for dropdowns, where the list of available values is dependend from the selected value of another dropdown.

2.5.17 2.2.0 (2015-06-17)

- Fix display of validation error messages.
- Make it possible to display custom HTTP error messages when *message* was provided in the JSON response.

2.5.18 2.1.0 (2015-04-09)

- If ajax result is HTML instead of JSON data, display HTML (which is in most cases a 500 error or login page). (#11838)

2.5.19 2.0.0 (2015-03-26)

- Render the token as the `<option value>` attribute, so we get the best of both worlds: internally we deal with objects, but the DOM looks "normal" (and other DOM-based libraries like select2 can interface with it).
- Package the browser resources as a bower package.
- Switch from json-template.js to Handbars.js for templating. (#13804)
- Fixed a bug where using select fields with dynamic options (sources) triggered a save request with empty value upon creation in the UI.

2.5.20 1.2.0 (2014-10-22)

- Improved handling of object and multiselect fields such that the values loaded are actually just the values, with select options being passed as the `source` field option. This is an important backwards-incompatible change.
- Added a field option `multiple` that, when truthy and the selected type is object, makes the field be considered a list of objects. When using the default templates, this turns a select widget into multi-select.
- When loading values into a form via the JS API rather than via HTTP as JSON, assume full objects, not tokens to be contained.
- Add textarea template.
- Added the concept of required fields which cannot be saved when blank.
- More Bootstrap 3 compatibility.
- Simplified HTML mark-up of form elements.
- Removed the package metadata's implicit dependency on the test extra.
- Use classy classes.
- Consider HTTP error responses unrecoverable errors. This distinguishes them from time-outs and connection errors.
- Add `loaded` property on Form, a Deferred so clients can check whether loading has finished. This is mainly helpful for tests, e.g.:

```
selenium.waitForEval(
    '$("#jsform").data("form") && '
    '$("#jsform").data("form").loaded.state()', '"resolved"')
```

- Expose the `get_template` function for reuse outside the Form class.
- If an empty string is specified as `form_template`, just use the existing form DOM node without applying any template.
- Add jQuery plugin `$().jsform_submit_button(callback)` that sets up a button that saves the jsform and calls a function after the successful save.

2.5.21 1.1 (2014-04-07)

- Propagate save message of server using `after-save` trigger.
- Added infrastructure for displaying and clearing status messages, use it for announcing HTTP faults during AJAX calls.
- Display and clear field-related errors both at the widget and via status messages.

- When saving data, count anything but a JSON response with a status value of "success" as an error. Give up completely after an incomprehensible response, retry on connection errors either after the next successful server access for any other field, or when requested by a call to `retry()`.
- Added an API method `save_remaining()` that causes any fields to be saved that have not been modified so far. While this should only save the initial values loaded from the server, it is useful to apply the same validation and error handling to all fields.
- Added an API method `when_saved(retry)` that returns a promise that aggregates any pending and completed save calls, either propagating the retry behaviour of single fields (the default) or failing on server errors.
- Provide a visual clue and status message while saving a field.
- Some refactoring to improve code readability.
- Made code pass jshint check.
- Made default and currently not overrideable status message behaviour compatible to bootstrap 3
- Properly unsubscribe change handlers when updating fields.
- Added simple localization.

2.5.22 1.0 (2013-12-13)

- Remove console calls as they are not understood by all browsers.

2.5.23 0.8 (2013-12-10)

- Fixed: jsform did not render in IE8 if form template started with line break.

2.5.24 0.7 (2013-12-03)

- Add ability to send a CSRF token with every request. This token must be available via the id *csrf_token* (can be customized) in the DOM.
- Added minified versions of javascript resources.

2.5.25 0.6 (2013-09-06)

- Bugfix: Use `indexOf` instead of `startsWith`, which is not available on all browsers.

2.5.26 0.5 (2013-09-06)

- Declare `for` attribute on form labels.
- Store "save on change" subscriptions so they can be cancelled.
- Ignore `null` values for data fields. (#1)

2.5.27 0.4 (2013-08-27)

- Made it possible to define templates as template files on file system.

2.5.28 0.3 (2013-08-27)

- Add events `after-load` and `after-save`.
- Fix `JSON` serialization to be able to handle Knockout observables.
- Added `reload` functionality to the form class.

2.5.29 0.2 (2013-08-26)

- Made it possible to preselect values in arrays when the form is rendered.
- Changed form submit behaviour:
  - Default submit type is not `POST` instead of `GET`. (Change it with the `save_type` option)
  - Data is now submitted as `JSON` type.

2.5.30 0.1 (2013-08-17)

initial release

Symbols

A

C

D

E

F

G

H

I

L

N

O

R

S

T

U

W