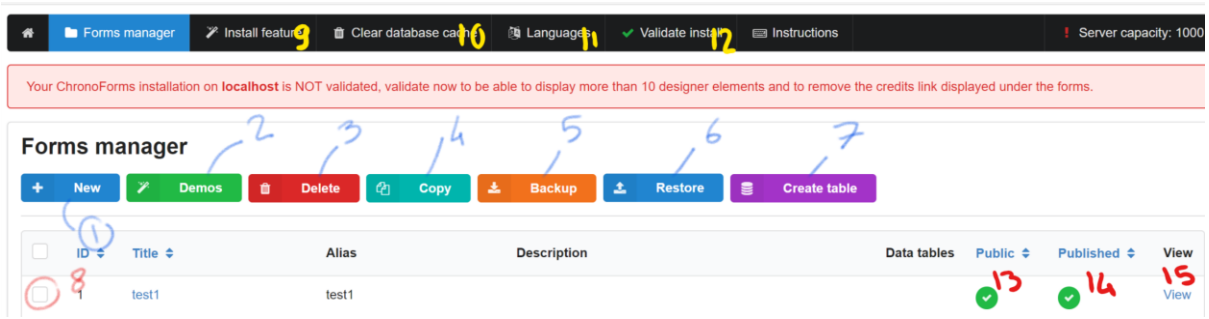
CHRONOFORMS V6 MANUAL

Latest version available on ChronoEngine.com

TABLE OF CONTENTS

THE FORMS MANAGER



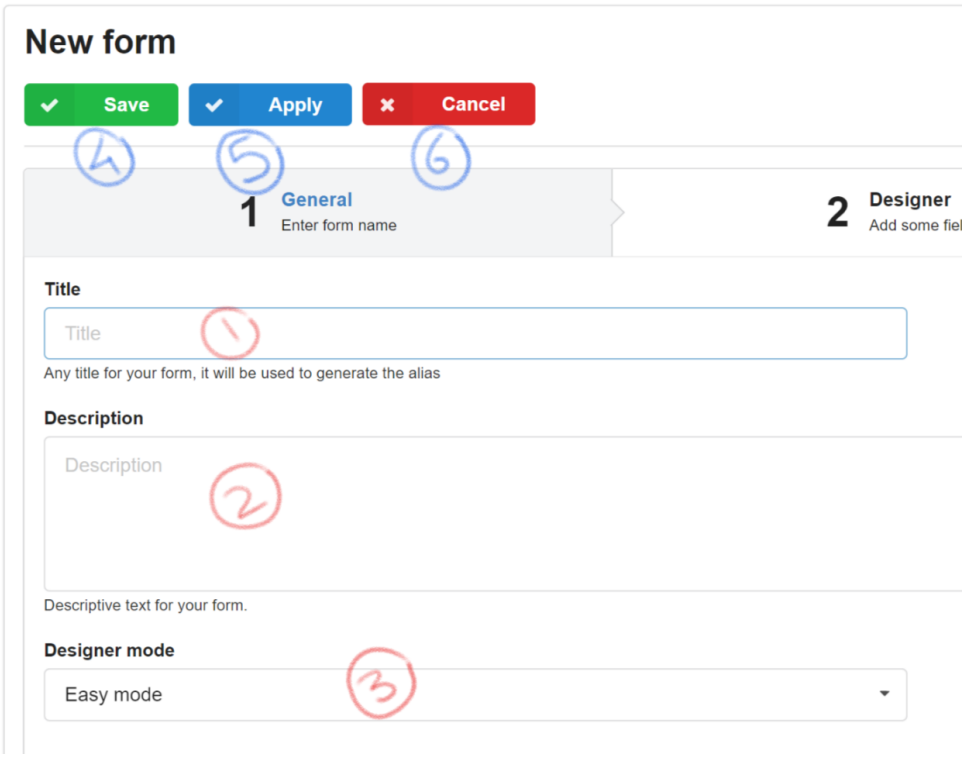
The forms manager provides a lot of functions, here is a list of the available buttons and their descriptions.

1. **New**: Create a new form, your new form can be started in Easy mode or advanced mode.
2. **Demos**: Choose one of the available form demo templates to start building your form with.
3. **Delete**: Select a form from the forms list and click this button to delete it, you may also delete multiple forms.
4. **Copy**: Select one or more forms to copy.
5. **Backup**: Backup the selected form(s), the backup file created can be restored on any other Chronoforms 6 installation, please note that any data tables connected to the form will not be included in the backup file created.
6. **Restore**: Click to choose a valid Chronoforms 6 backup file and restore it, v5 forms backup files can also be restored, but the form will not be fully restored, due to the differences between v5 and v6
7. **Create table**: Select ONE form and click this button to create a database table for this form, you may add, edit or remove any fields.
8. **Selector**: This checkbox is used for selecting a form from the list.
9. **Install feature**: Open the installer page which is used to install more Chronoforms plugins/addons.
10. **Clear Cache**: Clears the Chronoforms cached database tables structure and CSS files, you should clear the cache whenever you modify any data table connected to your forms using a database tool like phpmyadmin.
11. **Languages**: Allows you to translate the Chronoforms admin interface to your desired language.
12. **Validate install**: Validate your Chronoforms installation to remove the credits link and any other functionality restrictions.
13. **Public form**: Enable viewing this form on the website frontend, otherwise the form is only usable in the admin area.
14. **Published**: Enable or disable the current form.
15. **View form**: Display the form on the frontend.

BUILDING A FORM IN EASY MODE

In this section we are going to demonstrate how to create a “bare minimum” form using the new Chronoforms v6.

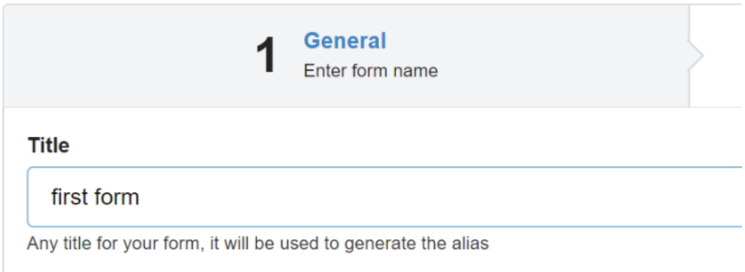
To create a new form, click **New** then click Easy form



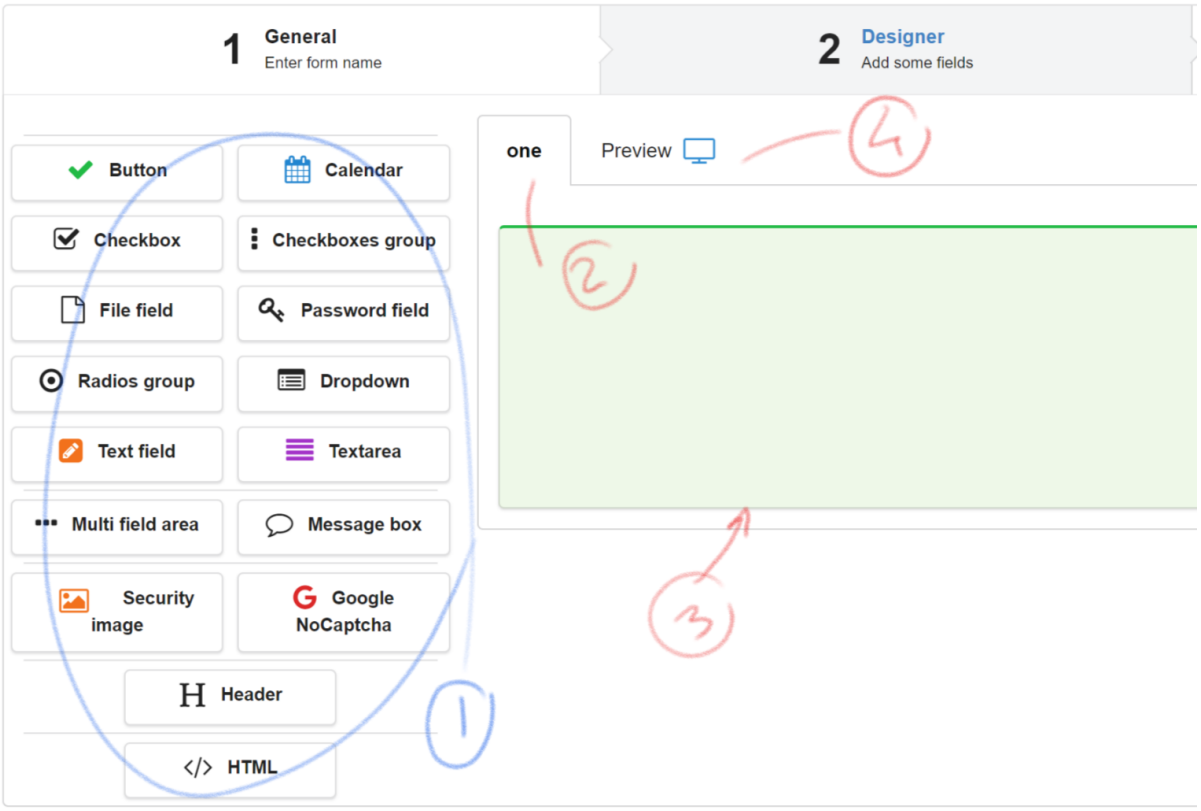
In the new page displayed, you have the following options:

1. **Title**: Enter your form title, the title is required and should be unique and should not be similar to any other form title you have.
2. **Description**: Just any text to describe your form, this is optional and is only for you or the other site admins.
3. **Designer mode**: The current form’s editor mode, you can switch it to advanced mode anytime to start editing your form in advanced mode later.
4. **Save**: Save the form and redirect to the forms manager.
5. **Apply**: Save the form but keep it opened for further changes.
6. **Cancel**: Leave the form page and ignore any changes made.

Let’s go ahead and enter a title for our new form, we will call it “first form”.



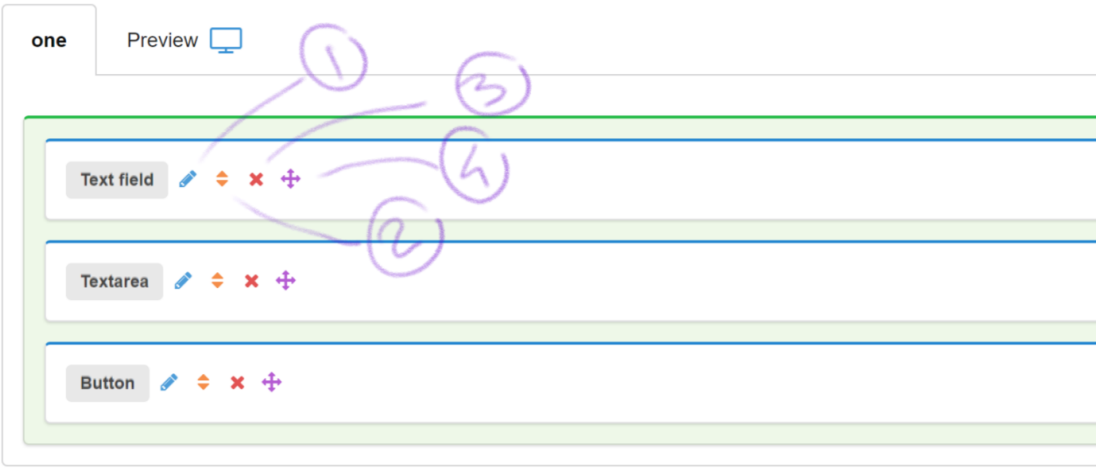
Now it’s time to design our form and add some fields, so we need to open the “Designer” tab:



As shown the designer has few key elements:

1. **The form elements**: These are the list of elements you can add to your form interface, each element has a different display and function when the form is rendered, elements can be dragged using the mouse to the green area on the right side (3).
   Some elements, called “Areas” can host other elements, for example, the “Multi field area” can host multiple elements and display them in one horizontal row, as long as there is enough space for them.
2. **The section name**: The form designer can have multiple sections, each section can host multiple elements, a new form has 1 single section by default, and in most cases you would not need to create more than one, but it’s possible to create more sections.
3. **The section body**: This is where you drag the elements from the left side, elements inside the section can be sorted.
4. **Preview section**: When you have some elements in your section, you can click the preview tab to check how they are going to be displayed on your form exactly.

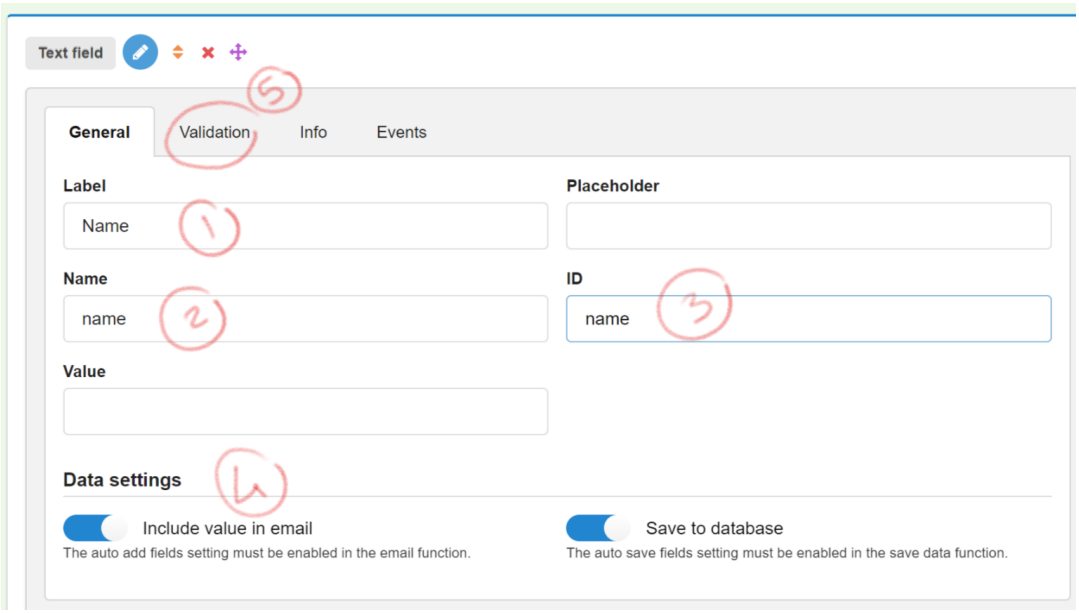
Let’s go ahead and add few elements to our section, we will add 1 text field, 1 textarea and a button.



The new elements have some control icons as shown in the image, each of them has a different function:

1. **Edit**: Click the edit icon to open the element settings area.
2. **Sort**: Click and drag to move the element up or down inside the same holding section.
3. **Remove**: Click to remove the element.
4. **Move**: Click and drag to relocate the element to a different section or area.

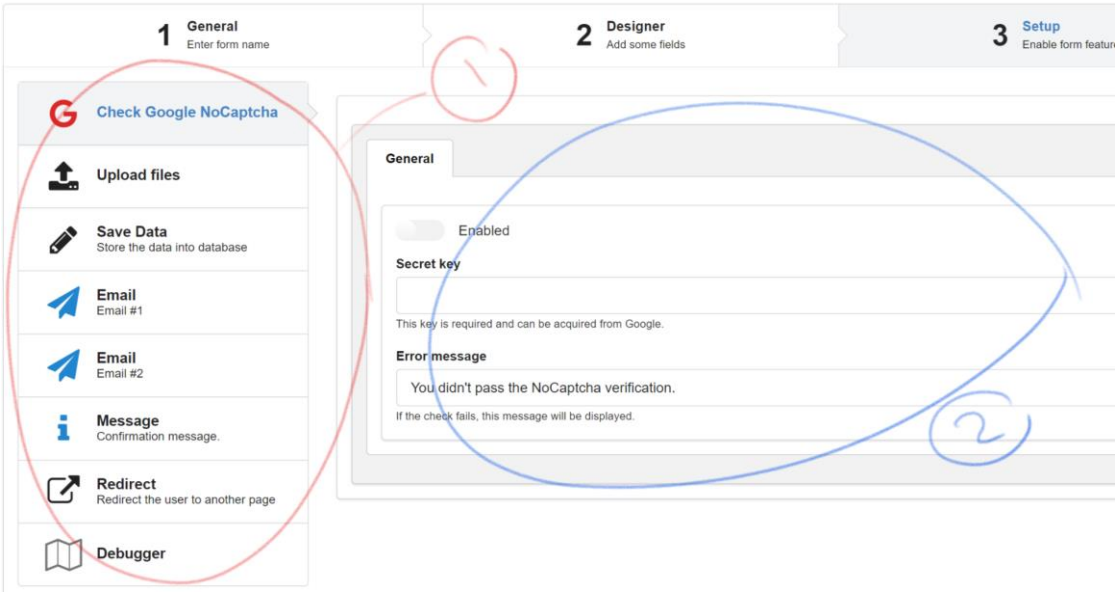
Let's edit some of the text field settings to make it a "name" field:



The settings which have been changed are:

1. **Label**: The field label displayed on the form.
2. **Name**: The field name used in the form code, although this may not be visible anywhere, its advised to use a meaningful value here to represent the function of the field, as you usually use this name to reference the field in different other places.
3. **ID**: The field id is less important than the field name but has the same rules as above.
4. Make sure the field is set to be included in the email, unless you do not want to.
5. Optionally you may set the field as a required field.

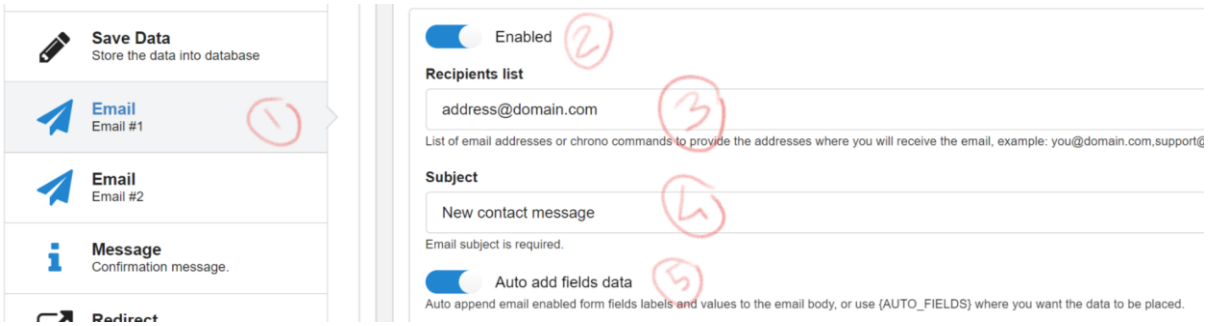
It is the time to move to the form setup section:



As shown, we have 2 parts in the setup section:

1. The form actions on the left side, all functions are disabled by default, you may click the name of any of them and that will display the action settings on the right side (2).
2. The action settings, you need to enable any action for it to work, some actions have some minimum requirements to function properly, for example, the “Check Google reCaptcha” requires a secret key, and requires “Google recaptcha” element in the designer in order to display the recaptcha test.

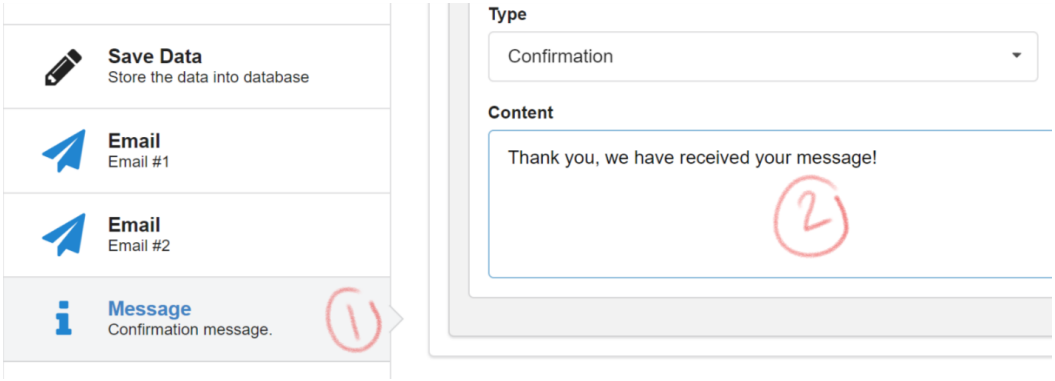
We want our basic form to send an email to the admin, and display a confirmation message to the user, so we will select the email action:



The settings shown are the bare minimum to get an email sent:

1. Select the Email action.
2. Enable the action.
3. Enter your email address.
4. Enter an email subject.
5. Make sure the “**Auto add fields data**” setting is enabled in order to have some email body content generated by Chronoforms.

Let’s configure to the **Message** action:

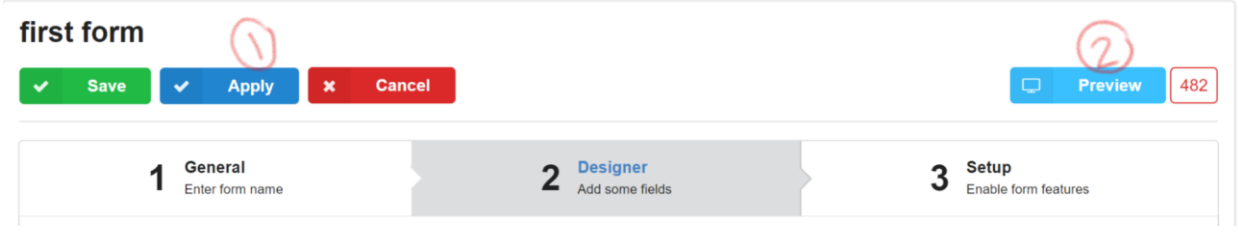


The minimum settings here are:

1. Select the message action.
2. Write the message text.

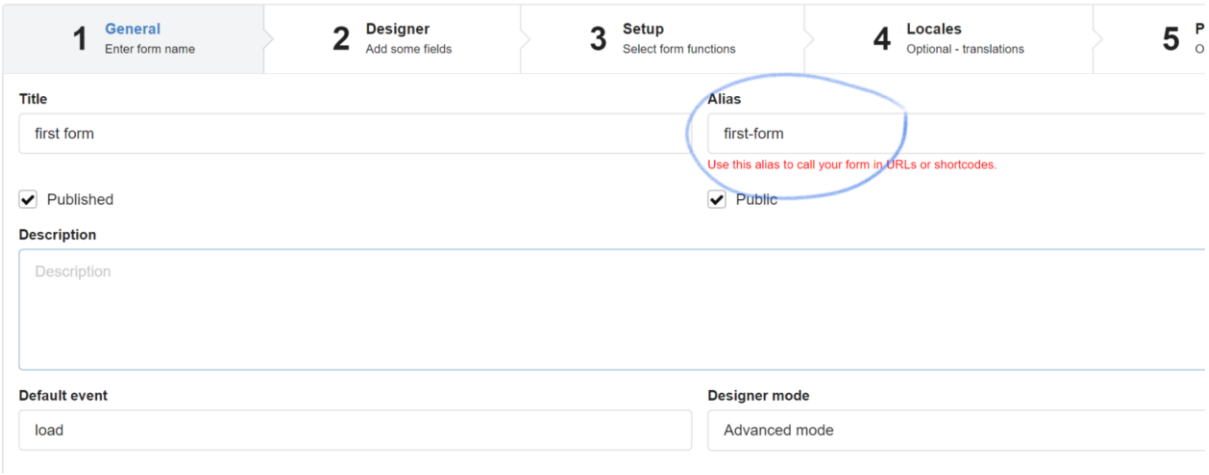
Optional step: in some cases, you may want to enable the “Debugger” before testing the form, the debugger can show helpful data in case something is not working as expected.

You may now save the form (1) and test it using the Preview button (2) at the top right corner:



THE ADVANCED MODE

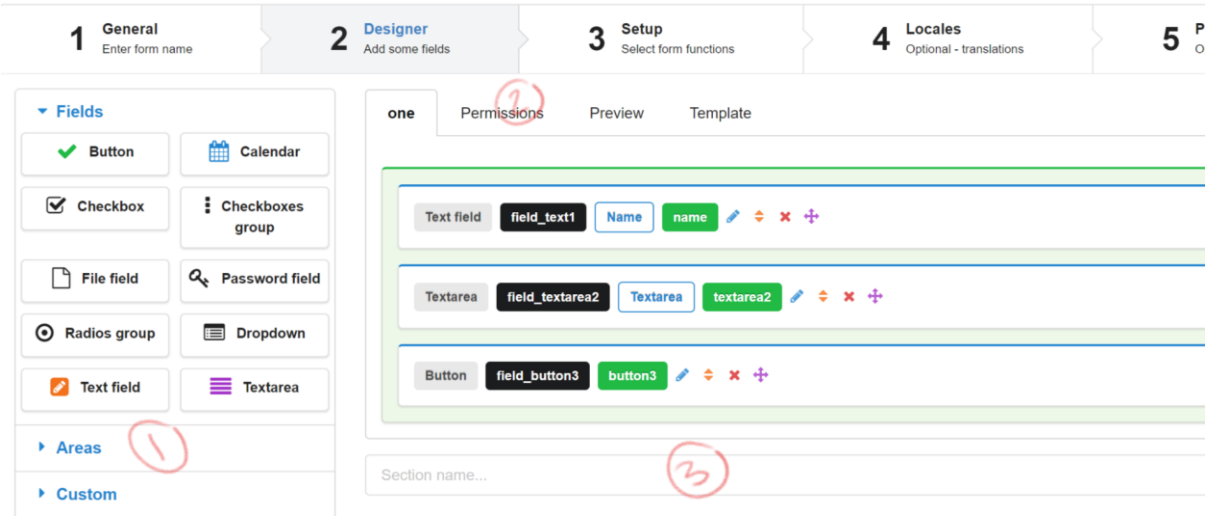
We can switch the "**Designer mode**" to "**Advanced**" under the "**General**" tab, in order to demonstrate the differences and the new features available.



The form Alias is a unique string used for identifying the form, this string is automatically created when the form is saved, you need to use this string when you want to link the form to a Menu item or call it using the plugin shortcode: {chronoforms6}first-form{/chronoforms6} on Joomla OR [Chronoforms6 chronoform="first-form"] on WordPress.

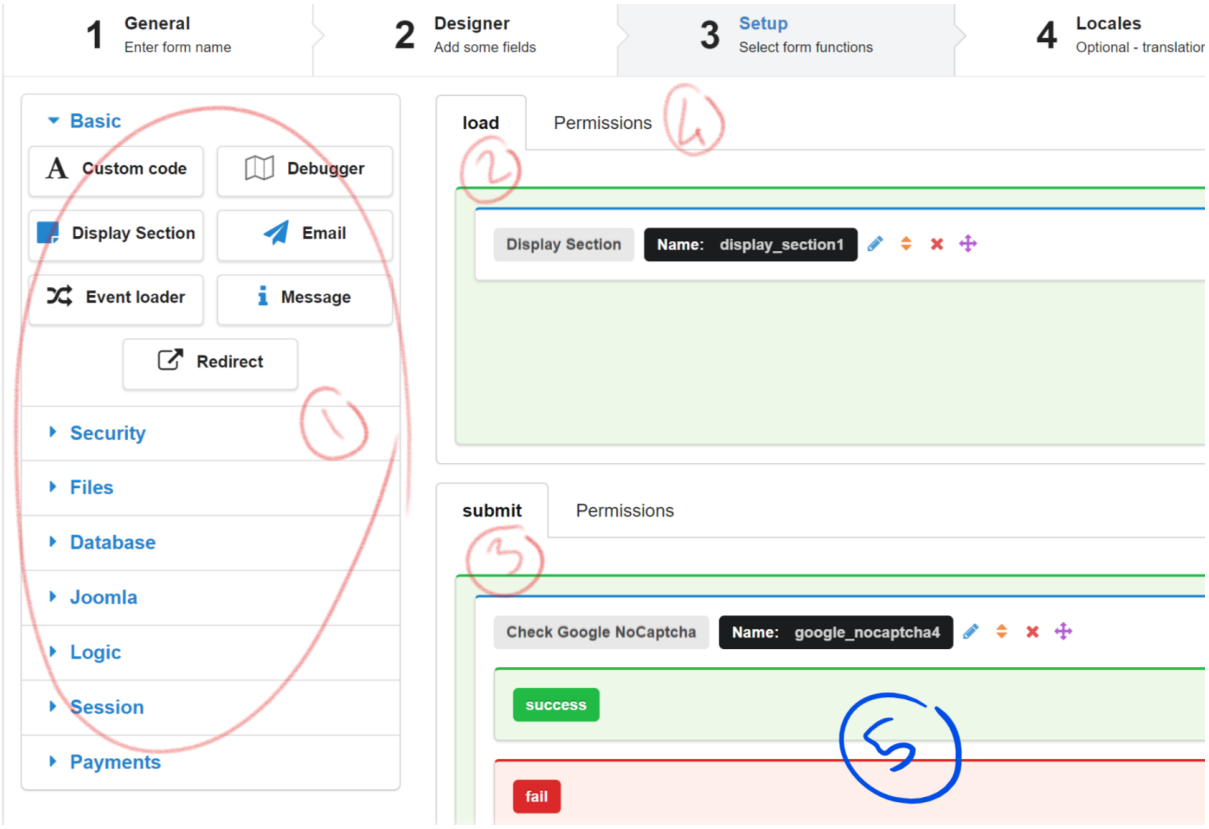
In most cases you do not need to change the alias, but if you do then it should not contain any spaces or special characters.

Switching to the "**Designer**" section displays new features and settings:



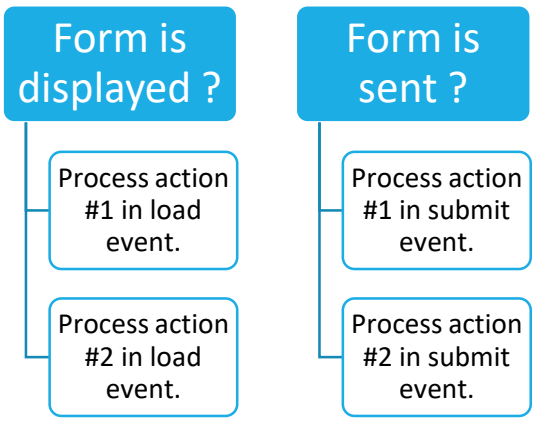
1. More elements are available and they are categorized into groups, click the group blue heading to expand the group and display all the elements inside.
2. Control who can access each section.
3. Add more sections to your form.

Click the "**Setup**" tab to open the form setup section.

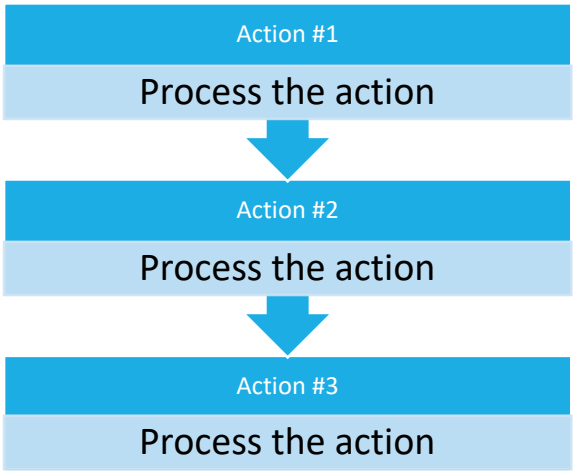


The setup under advanced mode is totally different compared to the easy mode, here are the top 5 differences:

1. Many more actions are available, actions are categorized into groups just like elements, and you drag them to the "Events" on the right side.
2. The form events are accessible, a form event is simply a form page, each form has usually 2 events, one event is processed when the form is loaded, and another event is processed when the form is sent or submitted.
   The load event usually has the "**Display section**" action which simply renders the form section(s), so in the image above, we have the form setup to display the main section when the form is loaded, but you can add more actions to do more stuff when the form is loaded.
3. The setup event is processed when the form is sent or submitted, usually you want to add an email action here along with a message action and may be you can also add some security actions to block any unnecessary spam or filter specific results.



4. Actions are processed in the order displayed, so you usually want to have the anti-spam actions BEFORE any other actions.

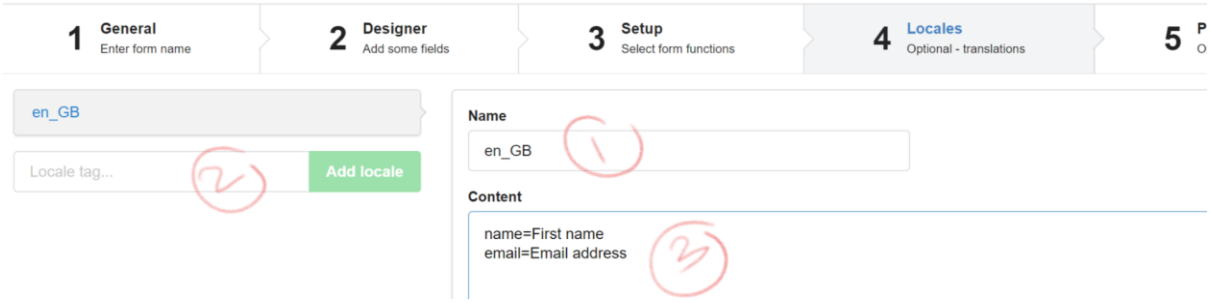


5. You can control which user groups can access each event.
6. Some actions have their own private events, so for example, the "**Check Google reCaptcha**" action has 2 events, one is processed when the captcha test succeeds and another when the captcha test fails, if the events are left empty then the form will proceed to the next action in the queue, but what we usually need to do is to interrupt the form processing when a fail event occurs, and this is usually done using an "Event loader" action, which halts the current processing and loads a different form event (usually the load event in order to redisplay the form).

LOCALIZATION

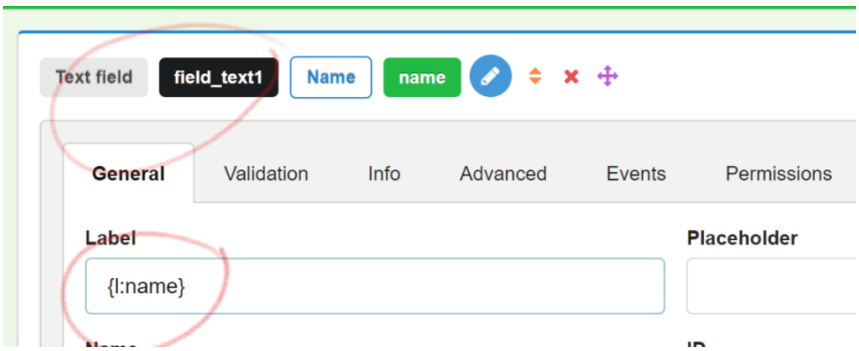
Sometimes its required to translate some of the form interface text or some action setting, the translations are made in the "**Locales**" tab and then called anywhere in your form designer or setup using the language shortcode.

First, you need to go to the Locales area and define your locales tag along with the translated string:



1. Create your desired locale, the name should match the language of your website, in the image we are translating the strings for the users displaying the website with the active language set as English - United Kingdom.
2. You can create more locales, just make sure the locale name is set correctly.
3. Define your translations in multi-line format as **string=Translation**

Now we can call our newly translated string anywhere in the form using the language shortcode, for example, we can translate one of the fields labels:



The localization shortcode is **{l:STRING}**, we have used {l:name} to get the "name" string translation value and set it as the field Label.

DATA SHORTCODES

In Chronoforms v5 it was possible to get the value of form fields or request parameters by placing the field name between 2 curly brackets and use that in the email template, an example would be {field_name}, although this was good enough in most cases, it had so many limitations, and it was not possible to capture different types of data sources, and so a new way of capturing data has been introduced in version 6.

It's very important to understand this section as there are many usages for the shortcodes feature in the new Chronoforms and ChronoConnectivity.

REQUEST AND FORM DATA

You can get the value of any form field or request parameter (parameters passed in the url) using this syntax:

{data:**form_field_name**} or {data:**request_parameter_name**}

If, for some reason, you need a default value to be retrieved if the field value does not exist then you can pass a default value:

{data:field_name/**0**}

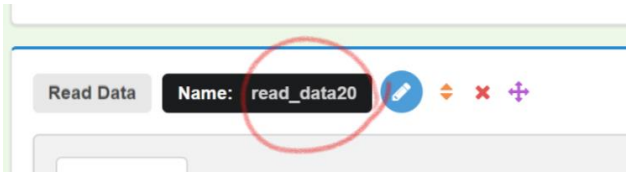
You may also set the request parameter value using a similar call:

{data.**set**:field_name**$value**}

ACTIONS DATA

After any action is processed, a result is returned, for example, after an email action, there is a true or false result, and after "Read data" action, the data table rows are returned.

This result is stored under a variable named after the name of the action, the action name is displayed inside a black label at the top of the action body



In order to use the result of the action, we need to use the variable matching the action name:

{var:function_name} e.g.: {var:read_data20}

If you are not sure about the value of some variable, then you can print the value of the variable in a readable format by using the following shortcode call in a "**Custom code**" action:

{var.**pr**:read_data20}

DATA PROCESSING

There are some helpful functions to apply to the variable data before they are returned, this becomes useful in many different situations as explained:

1. count: for array variables, return the number of items in the array instead of the array items.
2. sum: for array variables, return the summation of the array items.
3. join: for array variables, join the array items into a string a return it, a join character can be supplied: {data.join[,]:array_var_name}

4. empty: for any variable type, check whether the variable is empty or not and return true or false, usually used in switch based actions.
5. trim: for string variables, remove spaces from the start and end of the variable.
6. br: for string variables, replace new lines with html line break characters, useful for textarea fields data variables.
7. split: for string variables, split the string value and return an array of items instead, a splitting character can be supplied: {data.split[,]:string_var_name}
8. jsonen: encode a variable in json format and return the encoded string, usually used when storing an array of items into a single database table field.
9. jsonde: decode a json encoded variable, usually used with data retrieved from a database table when the data has been json encoded, returns an array of items.
10. pr: for all variables, print a readable output of the variable contents.

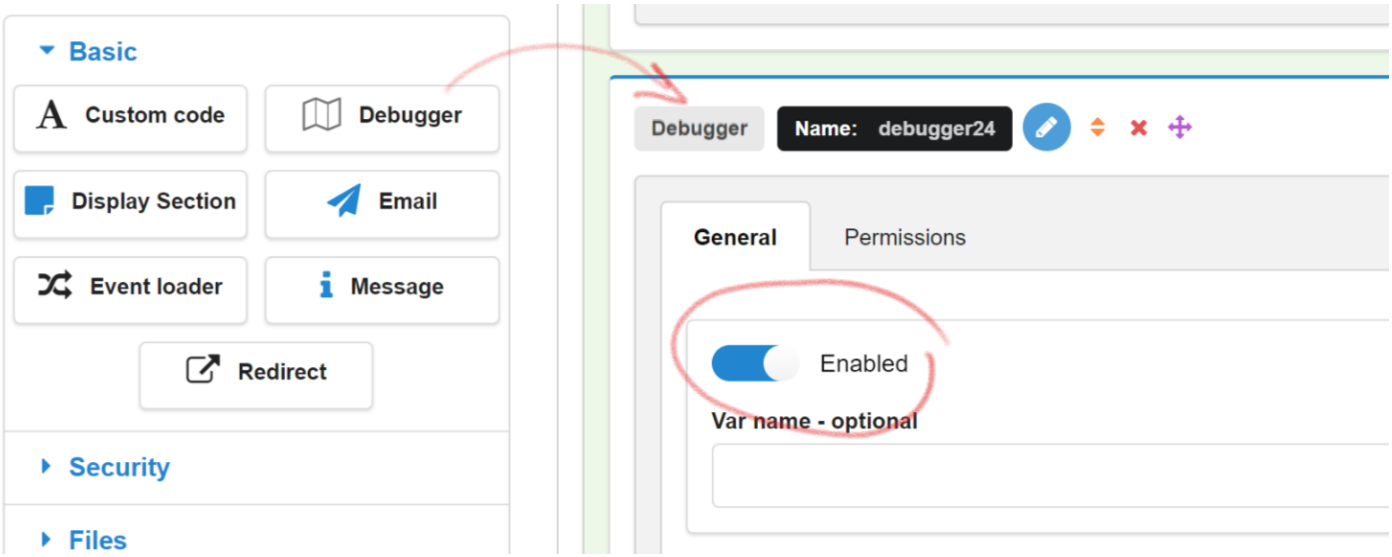
EXTRA UTILITIES

There is more information which you can get easily using the included shortcodes, here is a quick list:

1. {ip:}: get the user IP address, e.g: 192.168.1.0
2. {rand:}: get a random number, e.g: 4123456443
3. {uuid:}: get a unique hash string.
4. {user:**property**}: get the current user property, e.g.: {user:id} = 123 or {user:username} = admin
5. {date:**format_string**}: get the current date in the format passed, e.g.: {date:Y-m-d} = 2017-06-27

DEBUGGING

Use the “Debugger” action to get helpful information about the actions processed up to the Debugger step, the debugger should display a list of the request data available followed by the results of all processed actions, the later can be really helpful to find out the correct shortcode variable name to use or the variable data path.



The “Debugger” must be enabled in order to display the debug data.

An example debug output is shown below:

```
Array
(
    [option] => com_chronoforms6
    [cont] => manager
    [chronoform] => first-form
)
```

```
Array
(
    [email25] => Array
        (
            [recipients] => Array
                (
                    [0] => me@my-domain.com
                )

            [subject] => Hello
            [body] => My email content...
            [from_name] => My domain name
            [from_email] => from@my-domain.com
            [result] => the Mail could not be sent.
            [var] =>
        )

)
```

The most important points are:

1. The data array which includes all the request data and the form fields data (names and values).
2. The actions debug information, their names and their results.
3. Please note that the actions section is indexed by the actions names.
4. The result of each action is available under the "var" key, this is the value which can be retrieved using *{var:action_name}.*

SECURITY AND SPAM PREVENTION

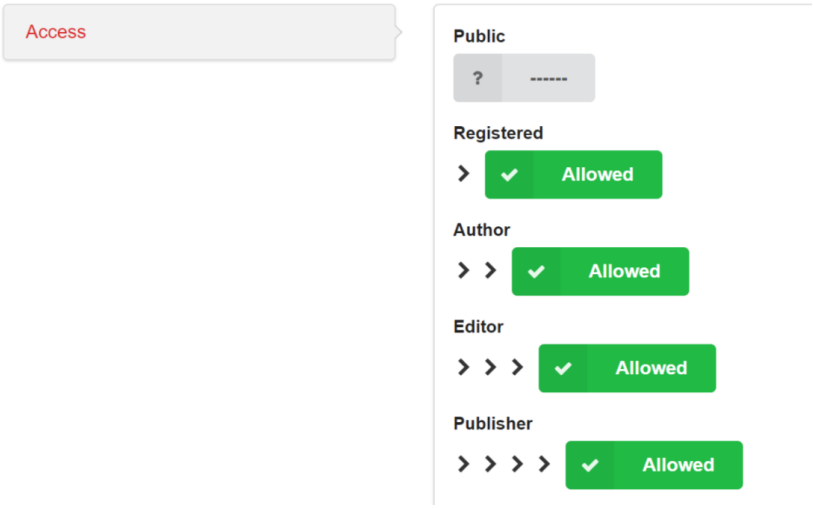
There are many ways to protect your form in Chronoforms v6, either by blocking access to specific user groups or by applying a security check.

ACCESS PERMISSIONS

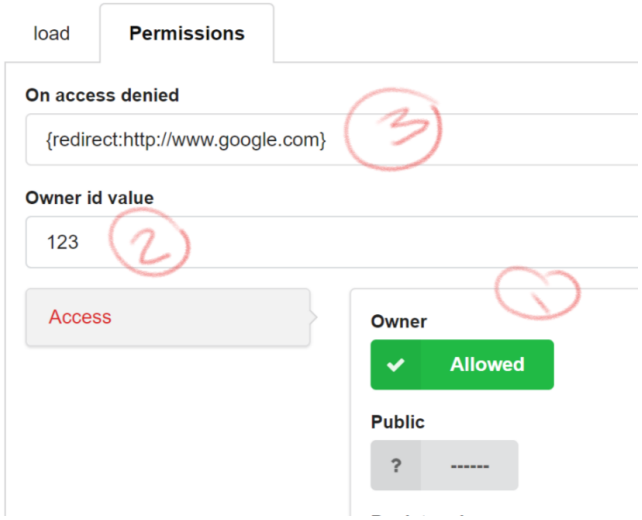
You can control access to the form at the "**Permissions**" tab, set the permitted groups to "**Allowed**" and "**Deny**" or "**Ban**" any groups you do not want to permit access.

By default, the permissions are not applied, unless at least one of the groups permissions has been set, only then the permissions are effective and must be set to either allowed or denied, leaving any in empty state means that its inheriting the permissions of the parent group.

In the image below, the public group is not allowed access (because it's not set and does not inherit access from any parent groups) but the children groups are allowed access.



Some permissions areas support owner access:

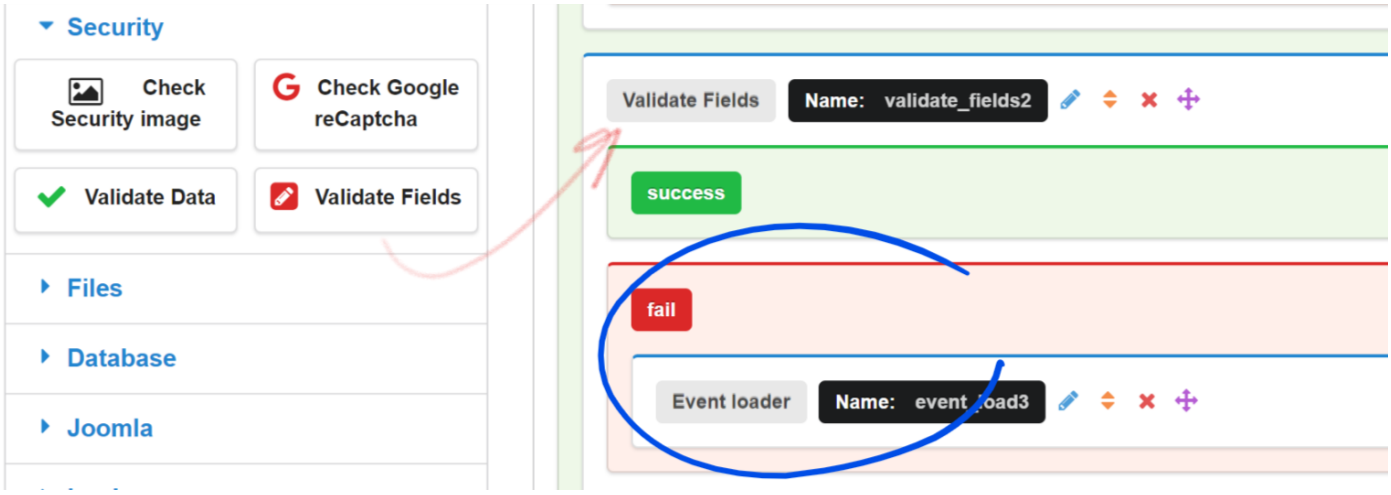


in this case you can:

1. Allow/Deny the owner.
2. Set the owner id which should match the logged in user id in order for the user to gain access to the resource, this owner id can be simply the user id integer value or a shortcode variable returning this value, this is useful if you need to control access based on a value retrieved from another resource like a data table row.
3. Set the action taken when the access is denied, in the example above, the user will be redirected to www.google.com where they can search the internet.

AUTOMATIC DATA VALIDATION

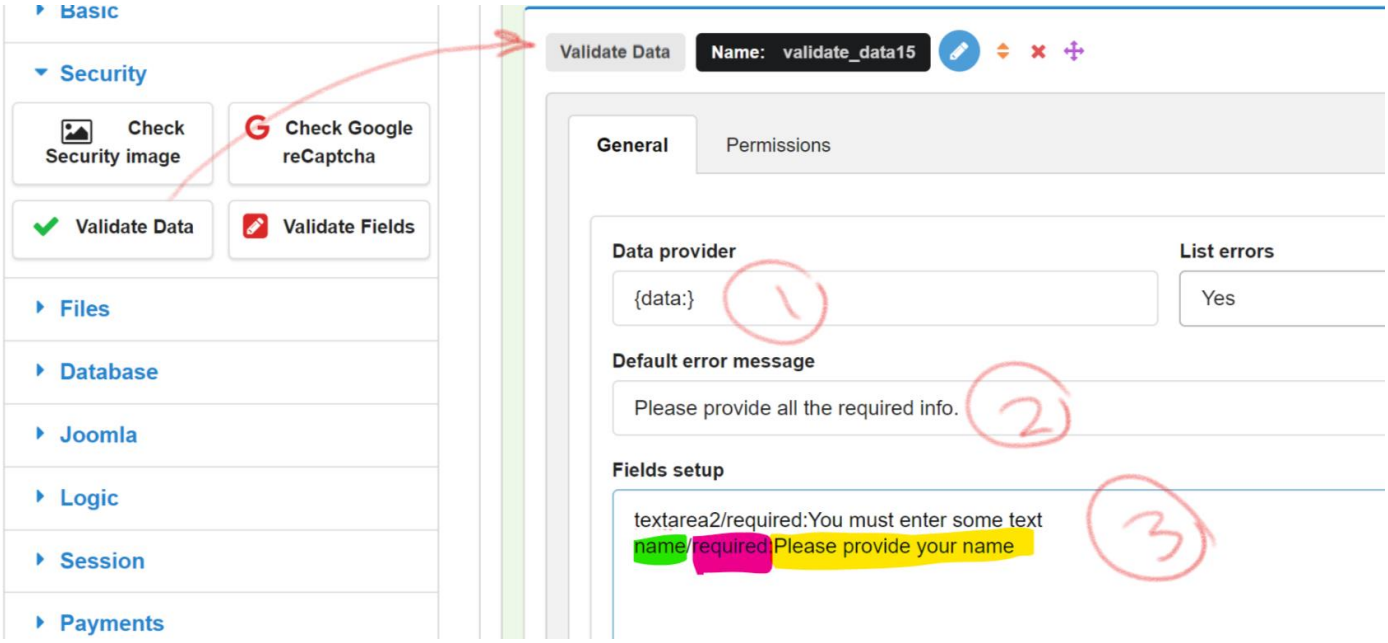
You can automatically validate the form fields data (on server side) and deny empty form submission or direct access to the submit url by using the “Validate fields” action, this action ensures that the form has been displayed by the current user before the submit url is accessed, the setup is fairly easy, just drag the “Validate fields” to the top of the “submit” event in your form and make sure that there is an “Event loader” action in the “fail” event in order to redisplay the form when an error occurs.



CUSTOM FIELD VALIDATION

In some cases, it may be desired to run validation rules which are not defined under the field settings (Field settings rules are checked by the "**Validate fields**" action [Automatic data validation]), this can be done using the "**Validate data**" action.

The "**Validate data**" action can apply the rules to fields inside any data source, a basic setup is as shown:



1. the "**Data provider**" is set to {data:} (here we are trying to validate fields the request data set).
2. Default error message to be used for fields rules which do not have an error message defined.
3. The fields/rules setup, the format is **field_name/rule_name:error_message**.
   you can find a list of all supported rules below.

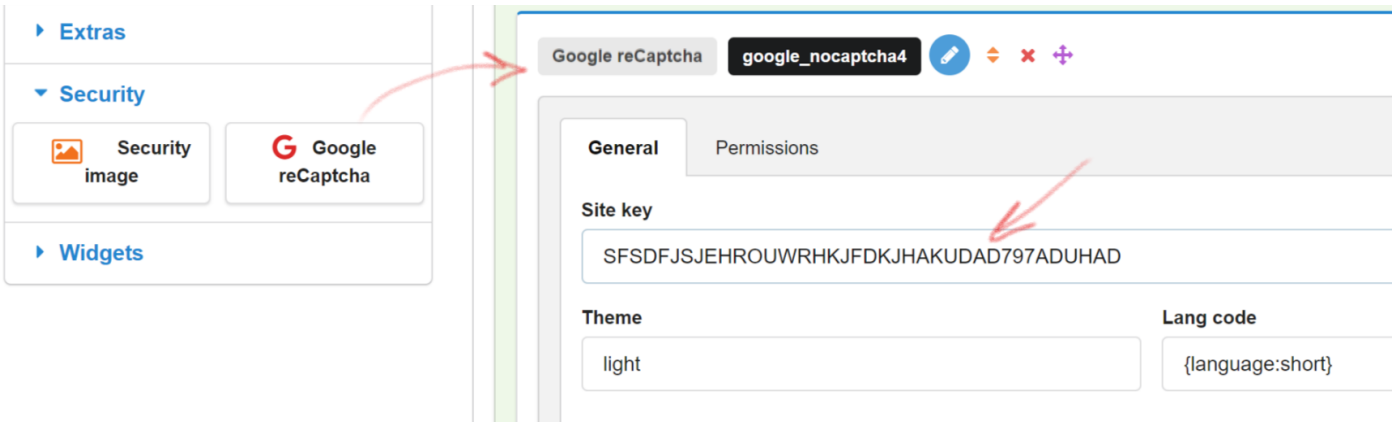
Validation rules supported are, rules names are in bold and are in lower case:

1. **required**: Field name exists and is not empty.
2. **is_empty**: Field value should be empty.
3. **no_spaces**: Field value should not have any spaces.
4. **is_integer**: Field value is an integer.
5. **decimal**: Field value is a decimal number.
6. **number**: Field value is a number.
7. **email**: Field value is a valid email address.
8. **url**: Field value should be a valid URL.

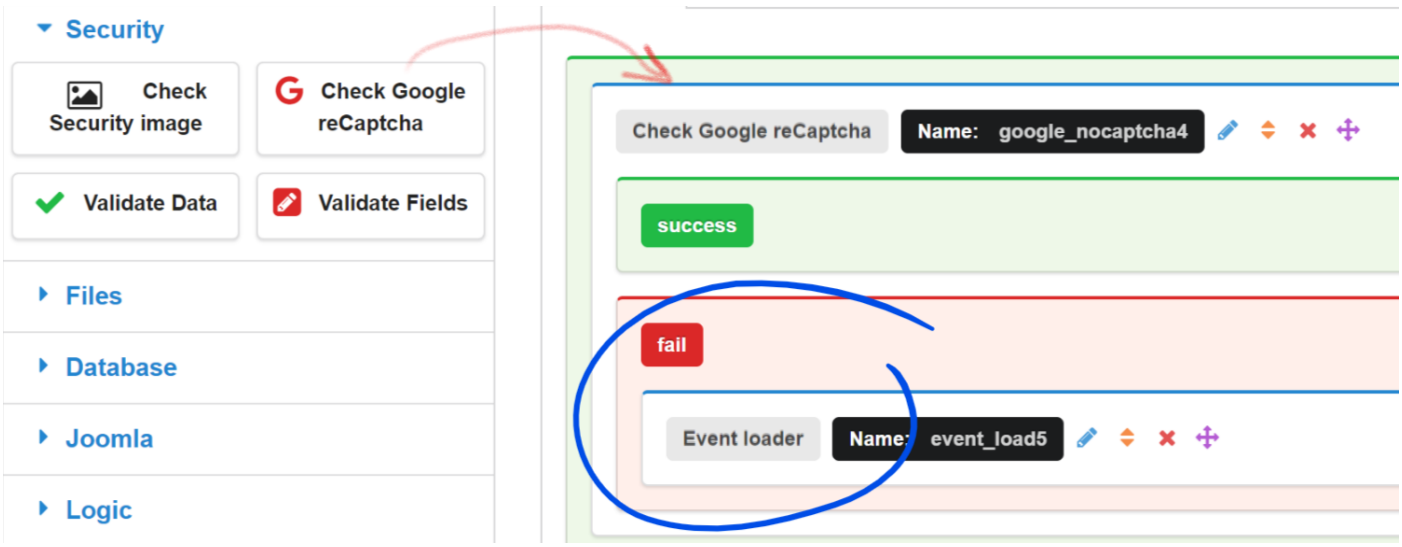
GOOGLE RECAPTCHA

Protect your forms against spammers by integrating the powerful **Google reCaptcha**.

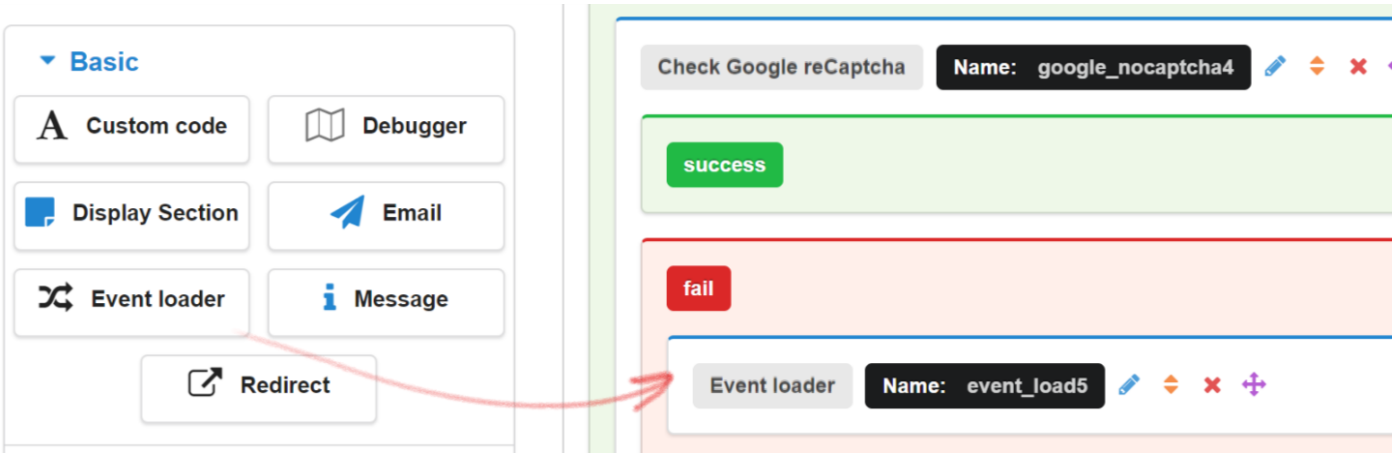
The first step is to add the Google reCaptcha element to your form and insert the site key of your current domain, you get 2 keys from Google when you apply for reCaptcha, a site key and a secret key, the site key should be placed in the reCaptcha element under the designer:



The next step is to add the "**Check Google reCaptcha**" action to your form's submit event in order to get the reCaptcha test result from Google, we also need an "**Event loader**" in the "fail" event in order to redisplay the form when the user fails to pass the reCaptcha test:



The "Event loader" action is under the basic actions group in the setup section.

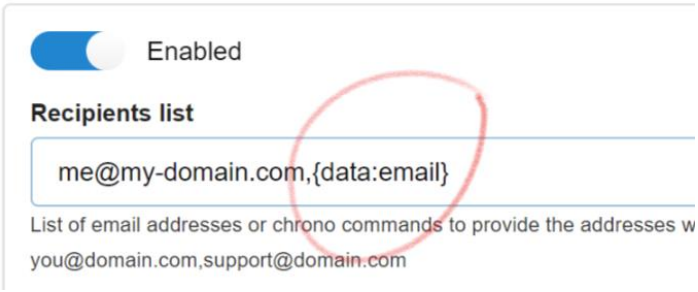


EMAILS

Sending emails using Chronoforms is very easy, the "**Email**" action has many strong features.

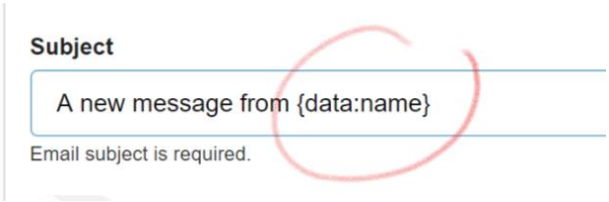
In order to have the email working, your email action needs to have at least the following 3 settings configured: *Recipients*, *Subject* and *Body*.

The "Recipients" setting takes an email address, or a comma separated listed of addresses, you may also use a variable shortcode to provide the address(es).



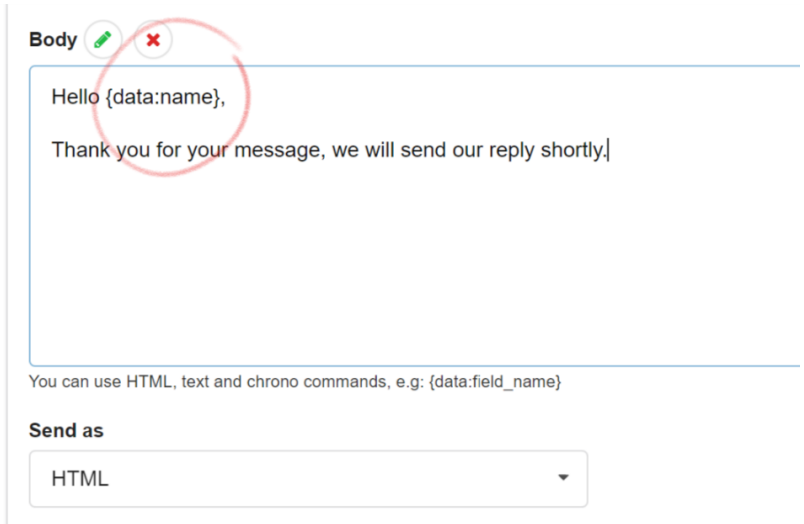
In the example above, we are sending the email to 2 recipients, the static address me@my-domain.com AND the address provided in the form field named "email".

Your email subject can be anything, and may contain variables calls too.

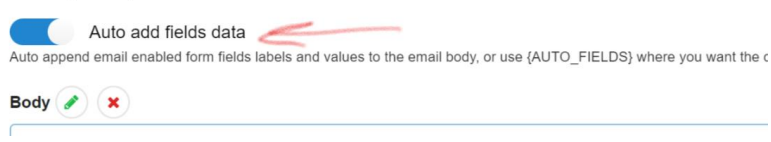


According to the example, the subject would be something like "A new message from XYZ", assuming that the value provided in the form field named "name" was XYZ.

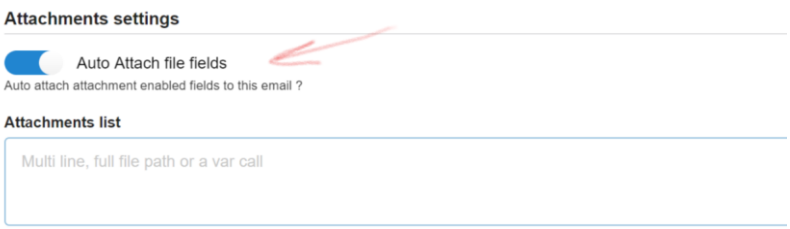
The "Body" setting should contain the intended email content, you may use the variable shortcodes to retrieve the form fields values:



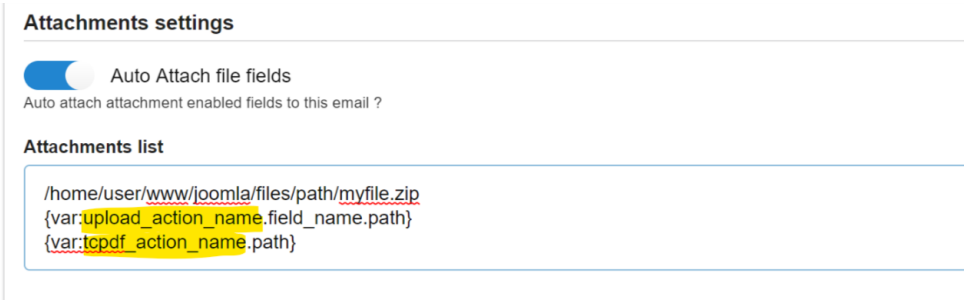
If you need to have the body content built automatically then just enable the "*Auto add fields data*" setting:



If you want to have the form file fields attached to the email, then the easiest way to do this is to enable the "*Auto attach field fields*" setting:



In some situations, you may need to attach more files to the email, before doing so, ensure that the files to be attached are saved on the same server of your website, then provide either the full path to the file or a variable which contains the full path to the file in the "*Attachments list*" box:



UPLOADING FILES

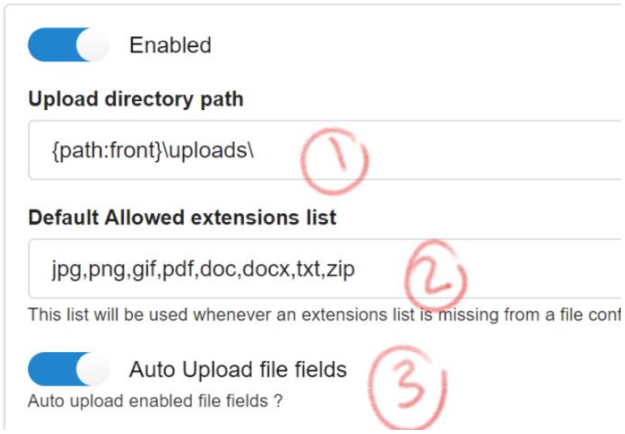
You can have a file upload feature in your form, files should be uploaded to your website before they can be attached to emails.

In order to upload the files selected in the form file fields, you need to use the "**Upload files**" action:

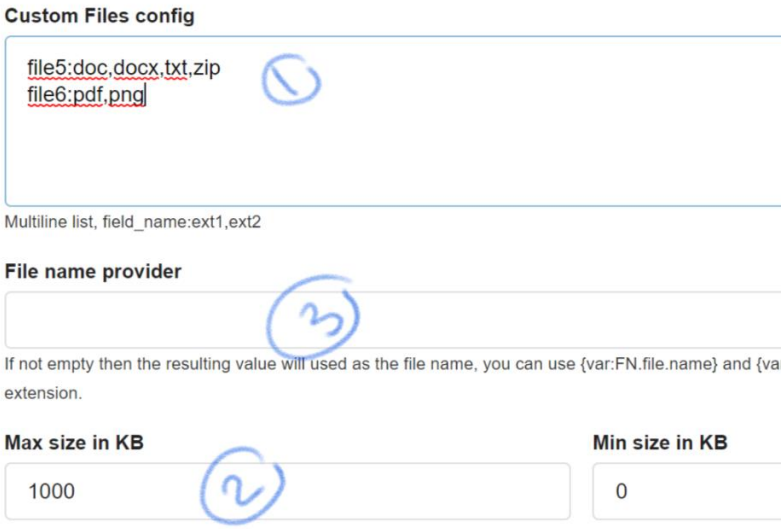
You should also have the "**Event loader**" action in the "fail" event of the "**Upload files**" in order to reload the form when an error occurs with the upload.

Assuming that you have form file fields configured to save the files to server, the upload files action should be ready once dragged from the actions list, but here are the main settings to change:

1. You should add the uploads path, this should be a full server path, in the example shown we use the {path:front} variable which returns the full server path to the Chronoforms folder.
2. You should have a comma separated list of permitted extensions, if the user tries to upload a different file type then an error will be raised.
3. Enable the "*Auto upload file fields*" setting if you have configured your form file fields to be saved to the server.



In some cases, you may need to manually configure the file fields, you can do this using the "*Custom files config*" box:



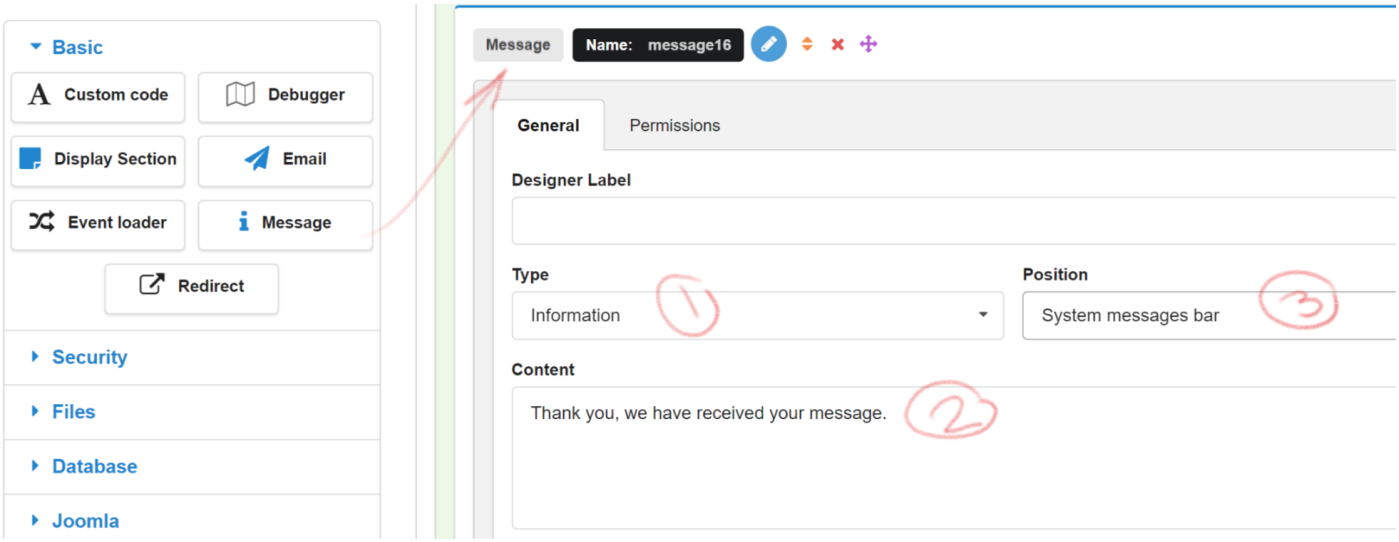
1. Insert a multi-line list of file fields names followed by a colon then a comma separated list of permitted extensions for each file, you can omit the extensions list here but the default extensions list configured before will be used.
2. Set the maximum permitted file size in Kbytes, 1Mega bytes = 1024 Kbytes.
3. Optionally, and for advanced usage, you may want to rename the upload files using a specific convention, use the "*File name provider*" for that.
   By default, an uploaded file named "test.png" will be saved as "timestamp-test.png", but you can change this by using the proper shortcodes in the "*File name provider*" field.

ALERTS AND CONFIRMATION MESSAGES

You can display an alert or a form message using the "**Message**" or "**Custom code**" actions included.

MESSAGE

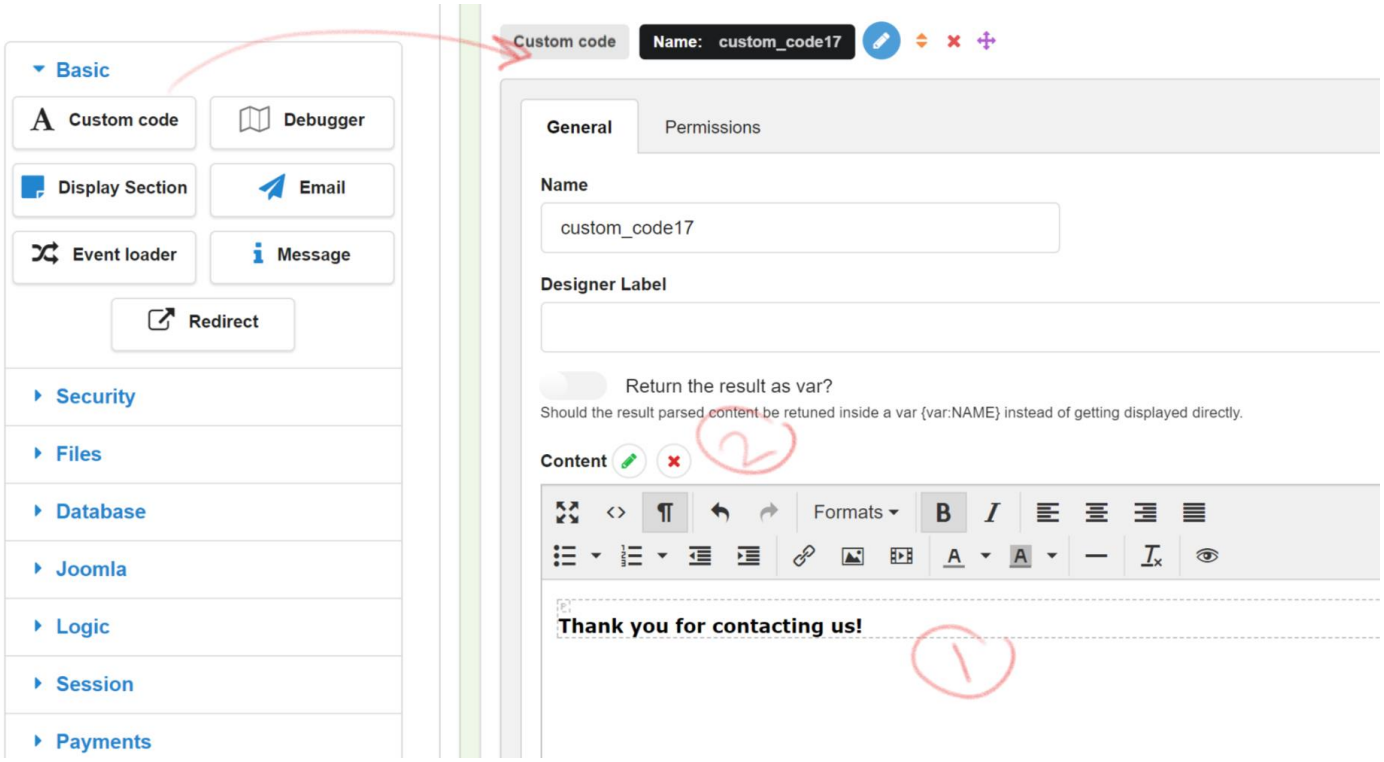
The message action supports four types of alerts, confirmation (green), error (red), warning (yellow) or info (blue), the setup is straight forward, just select the alert type and enter the message text, you may also select where on the page the message will be displayed.



1. Choose one of the 4 alert types.
2. Write your message, where you can use shortcodes to get the data of any form fields or request parameters, example: {data:**form_field_name**}
3. Select the message position on the page, in AJAX forms this may need to be set to "Body".

CUSTOM CODE

Write your own form content using HTML or PHP code in the "**Custom code**" action content, any PHP code should be included between **<?php ?>** tags.



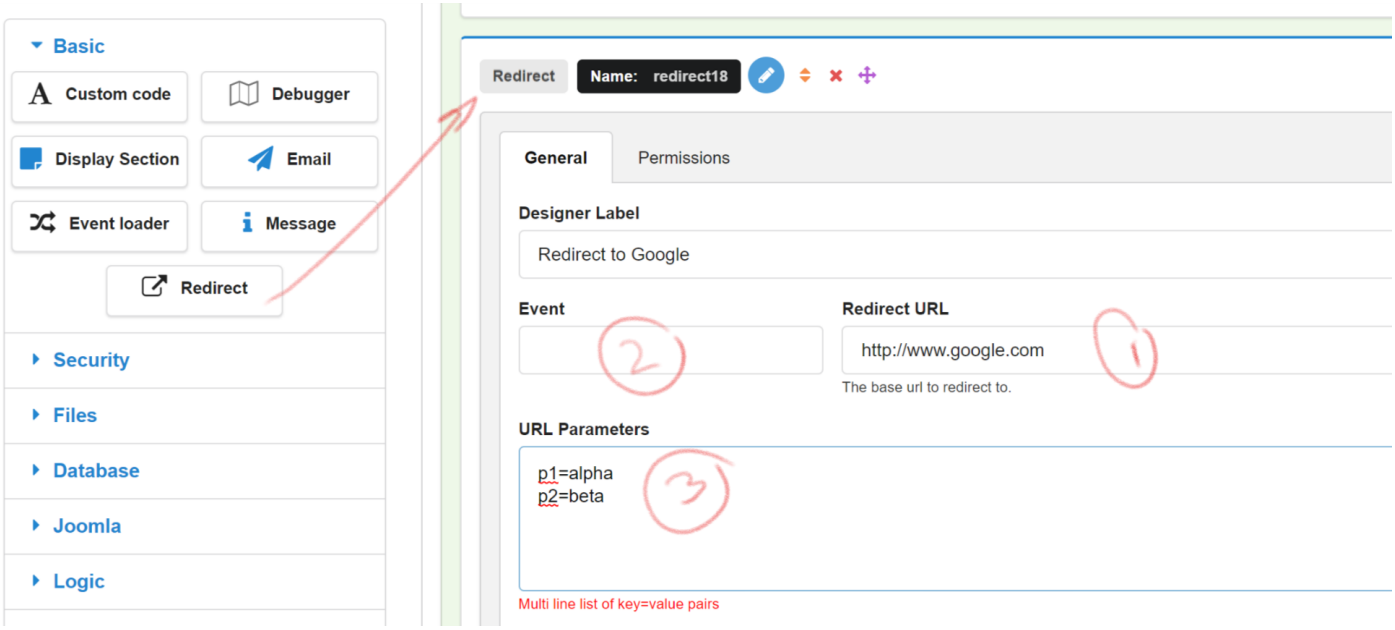
1. Enter the data you want to display in the "Content" box.
2. Optionally enable the wysiwyg editor to format the text, but this will remove any PHP code.

REDIRECTION

You can redirect the user to a different url at any point in the form processing using the "**Redirect**" action.

Once the redirect action has been triggered, any subsequent actions will NOT be triggered, this means that the Redirect action should always be the last action in the actions sequence, placing actions after it has no meaning.

The **Redirect** action setup is as following:



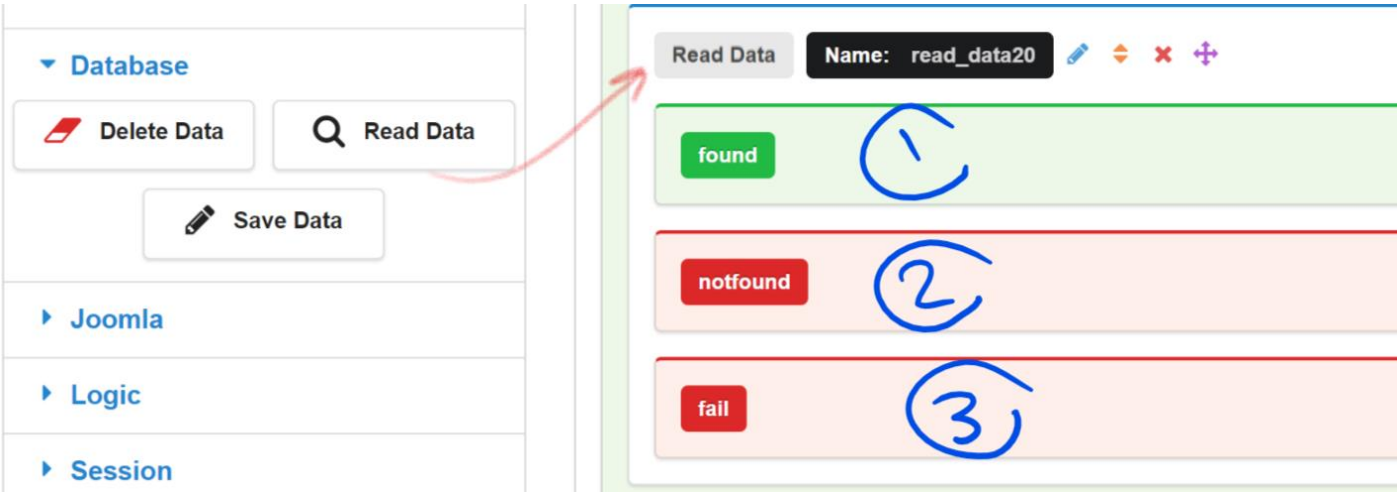
1. Insert the FULL url you want to redirect to, including the protocol (http or https).
2. Optionally you can add a form event name, this is useful only when you are redirecting to a different event in your form.
3. Optionally you can provide url parameters to be passed in the url, the result url according to the settings above would be: **http://www.google.com?p1=alpha&p2=beta**

DATABASE MANAGEMENT

Chronoforms version 6 comes with different actions to allow your form to read, update and delete data in the local database or in external databases.

READING DATA

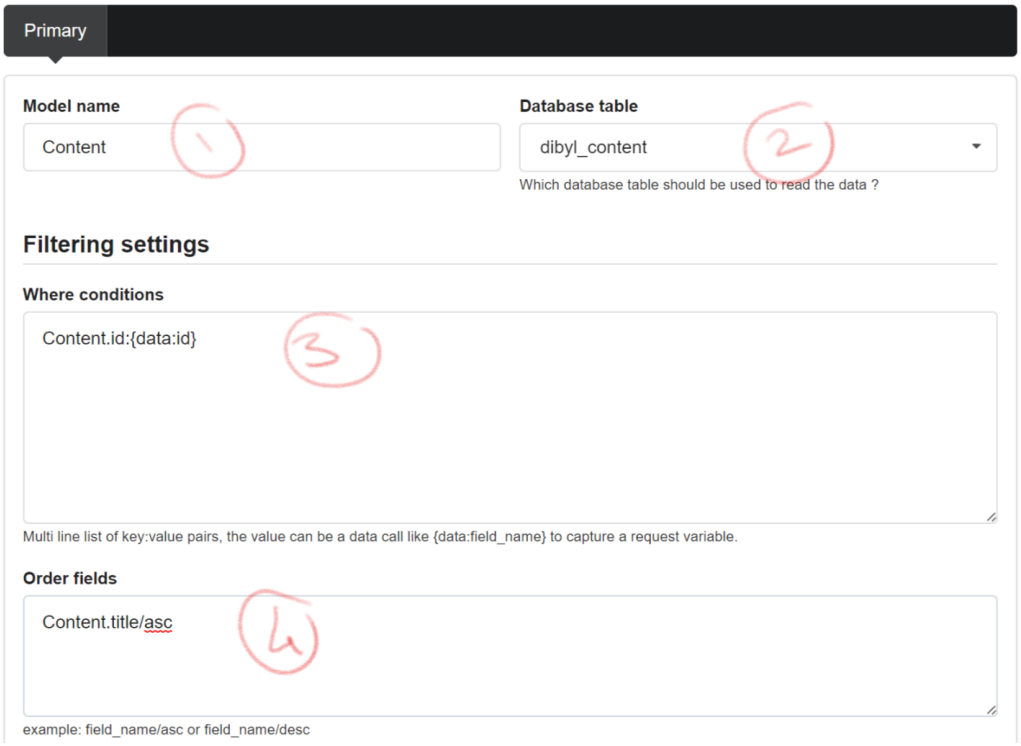
Start by adding a "**Read data**" action to your form



The "**Read data**" action has 3 events:

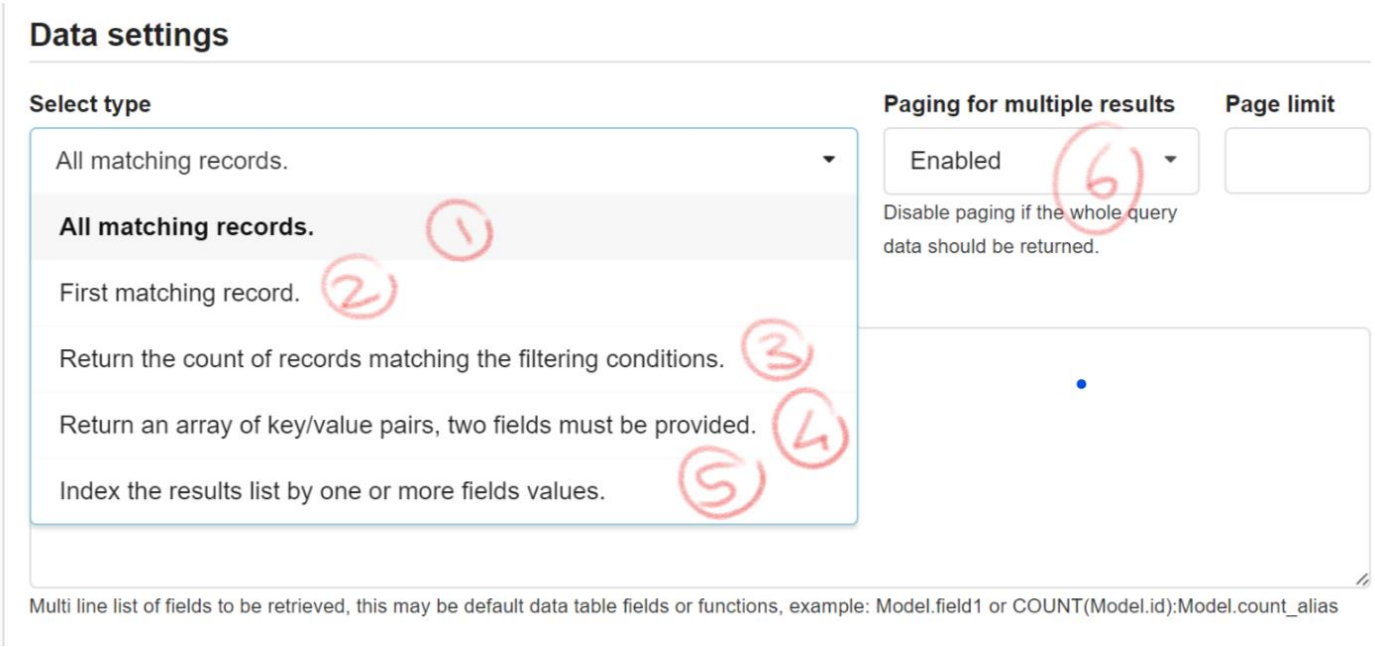
1. **Found**: triggered when a record has been found matching the read criteria.
2. **notfound**: triggered when no records have been found matching the read criteria.
3. **fail**: triggered when some error occurs during the reading operation.

Without configuration, the "Read data" action will not return any results, the basic configuration is shown:



1. Enter the **Model name**, the model name is an alias for the database table name used in reading operation and data associations, this should be a single word, no spaces or special characters.
2. Select the **Database table** to read the data from.
3. Optionally, set the reading **conditions**, if left empty, all the table rows will be read or the first row only, based on the **select type**.
4. Optionally, set the results **order**.

Additionally, you can control the number and format of the results returned using the "**Select type**" setting:



1. **All matching records**: return ALL matching records, an indexed array of multiple records is returned.

```
(
    [0] => Array
        (
            [Article] => Array
                (
                    [id] => 1
                    [title] => About
                    [state] => 1
                )

        )

    [1] => Array
        (
            [Article] => Array
                (
                    [id] => 2
                    [title] => Working on Your Site
                    [state] => 1
                )

        )

    [2] => Array
        (
            [Article] => Array
                (
                    [id] => 3
                    [title] => Welcome to your blog
```

2. **First matching record**: Return the first matching record only, return one single record, but it's still in array format.

```
[Article] => Array
    (
        [id] => 1
        [title] => About
        [state] => 1
    )
```

3. **Return the count**: Return the number of matching records, a 0 or integer is returned.
4. **Return an array of key/value pairs**: Return a single dimension array of keys and values, the fields used for keys and values should be set in the "Fields to retrieve" setting, this option is usually used for populating dropdowns from the database.

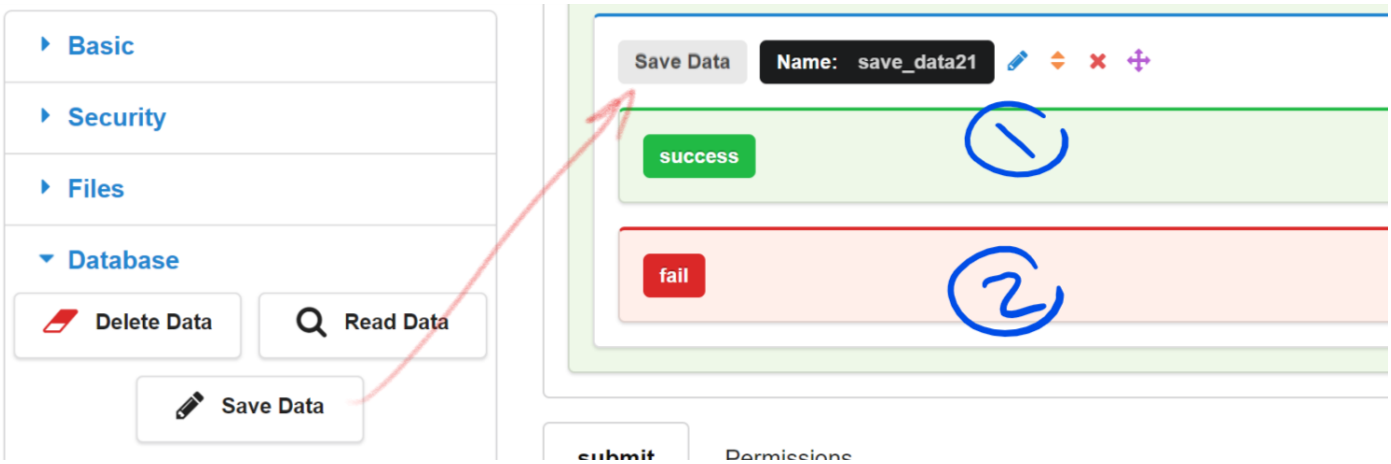

```
(
    [1] => About
    [2] => Working on Your Site
    [3] => Welcome to your blog
    [4] => About your home page
    [5] => Your Modules
    [6] => Your Template
    [7] => test home
)
```

5. **Index the results list by fields**: Return a list of results indexed by the value of a field.
6. **Enable results paging**, if enabled then only the first 30 records are returned, the results can be continued by passing the startat parameter in the url.

In the example above, the results returned by the "**Read data**" are going to be available under the variable {var:read_data20}, the structure of the results is dependent on the "**Select type**".

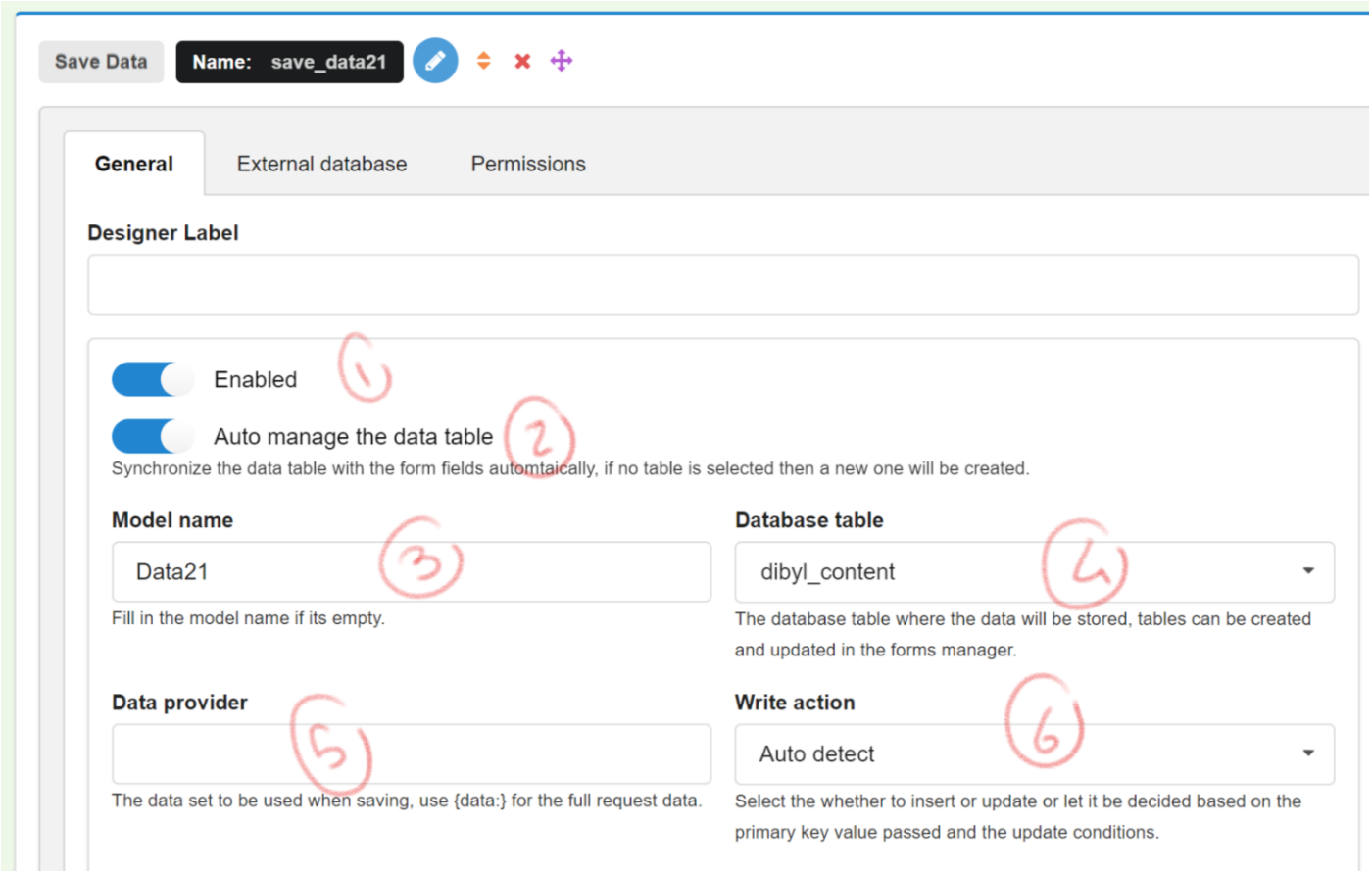
SAVING DATA

Any data can be saved or updated to the database easily using the "**Save data**" action:



The "**Save data**" action has 2 events:

1. success: triggered when the data has been saved or updated successfully.
2. **fail**: when the saving process has failed.

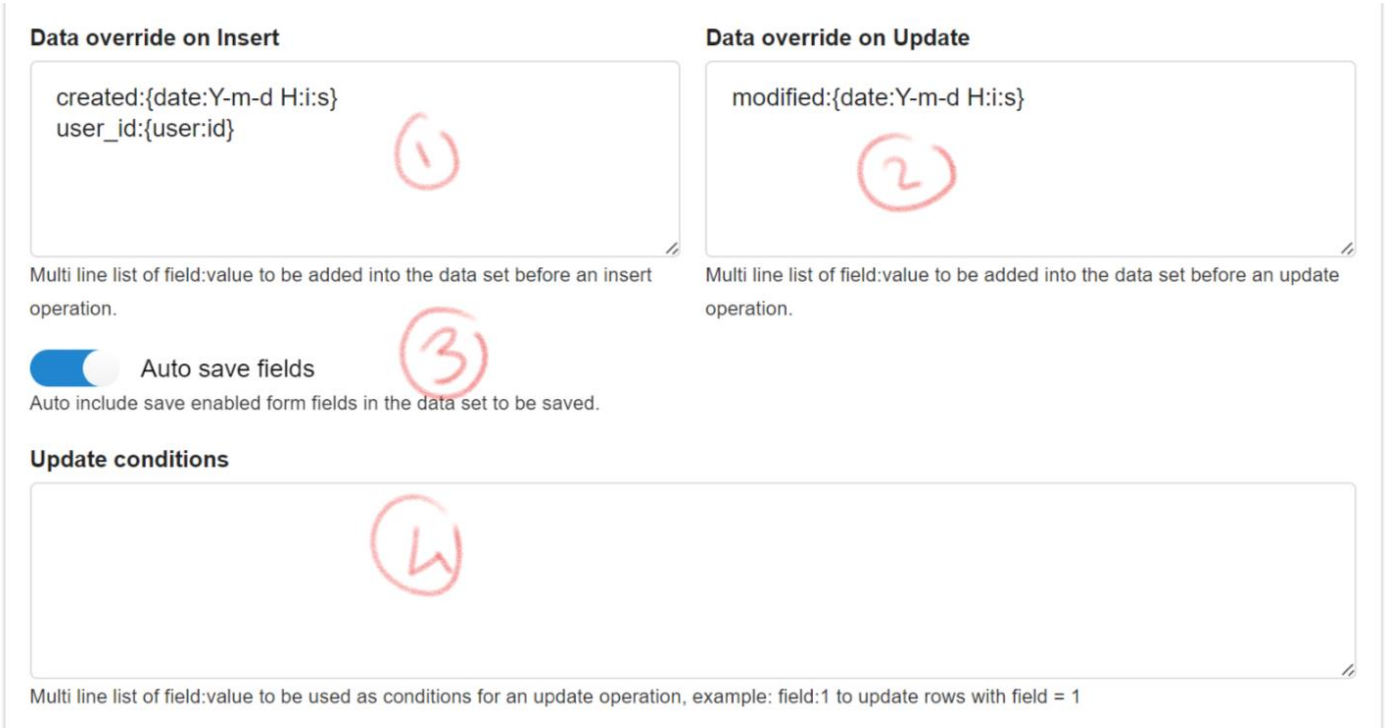


Few settings have to be configured for the "**Save Data**" action to work correctly:

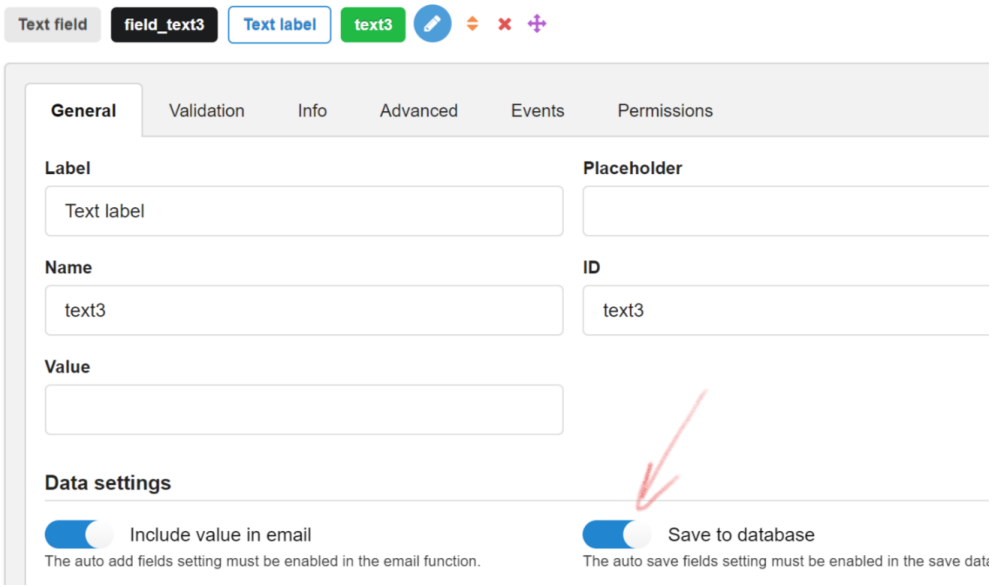
1. The action must be enabled.
2. Optionally, enable the table synchronization, this makes sure that the form has a database table available to store the form data, and that this table has fields matching the form fields names, if you are saving to a custom table and do not need to have all the form fields saved then you should disable this setting.
3. Set the **Model name**, no spaces or special characters.
4. Select the **Database table** in which the data will be inserted or updated.
5. Set the "**Data provider**", if left empty then an empty array will be used*.
   *leave this empty IF the "**Auto save fields**" setting is enabled, or set it to any data set you want to store in the database, the dataset fields should match the table fields names.

6. Optionally, select the write operation to be performed, either insert or update, Chronoforms detects this automatically but in some cases you may need to set it manually.

Few more advanced settings can also be configured:



1. Update the "**insert**" data set, the data here will be merged with the provider dataset before a "**new**" record is inserted.
2. Update the "**update**" data set, the data here will be merged with the provider dataset before an existing record is updated.
3. Auto update the data set with the data of the fields marked to be saved into the database (in the field settings area).



4. Add your update conditions, this is used when an update operation is performed, example: id:5

WRITING CONDITIONS

All the 3 database actions can take some conditions in order to read, update or delete data, the conditions can be provided in the following formats:

MULTI LINE SYNTAX

Just enter the field name followed by a colon then a value, the value can be just any value or a shortcode, examples:

- id:4
- id:{data:parameter}

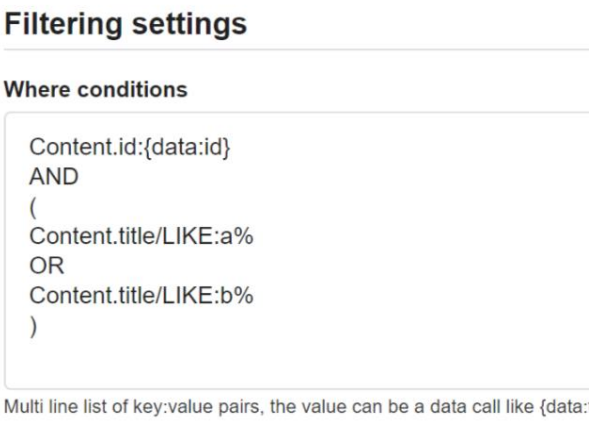
You may provide a arithmetic operator sign after the field name in this format:

- id/>:4 which means select records with id value greater than 4.
- title/LIKE:%test% which means the title field should be like “test”.
- id/<>:5 which means select fields with id value not equal to 5

Another cool feature is ability to skip parameters or block the operation:

- id:{data:parameter}/- this means that if {data:parameter} result is an empty value then skip this parameter from the conditions.
- id:{var:name}/+ means that if {var:name} results is empty then abort the whole action and return an error, this is useful if your action must have a specific condition met.

You may also use logical operators and/or group conditions together, as in the example below.



PHP SYNTAX

You may use PHP to build the conditions list, here is the equivalent php code to the example above:

return [[“Content.id”, “{data:id}”], “AND”, “(“, [“Content.title”, “a%”, “LIKE”], “OR”, [“Content.title”, “b%”, “LIKE”], “)”];

DATA ITERATION

Chronoforms version 6 has different Actions and elements which can iterate over data arrays in order to help you run actions or display content for each element in the array, this was not easily possible in any previous version.

Examples of this are the **Loop** and the **Loop event** actions, and the “**Repeater**” area element.

All the data iteration actions and elements have the same basic setup and workflow, you need to set a data provider which should be an array, the action or element then iterates over the values and processes the content provided.

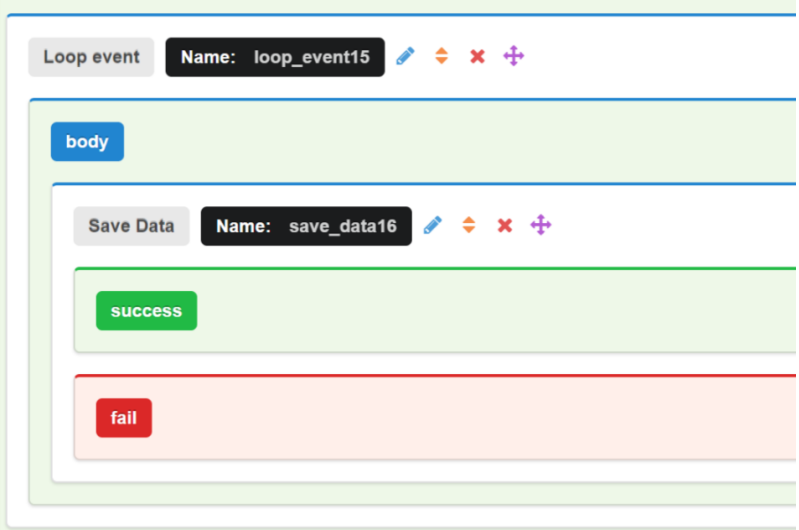
Inside each iteration, 2 variables are available:

1. {var:iterator_name.key} : Holds the key of the current iteration in the array.
2. {var:iterator_name.row}: Holds the value of the current iterated item/object.

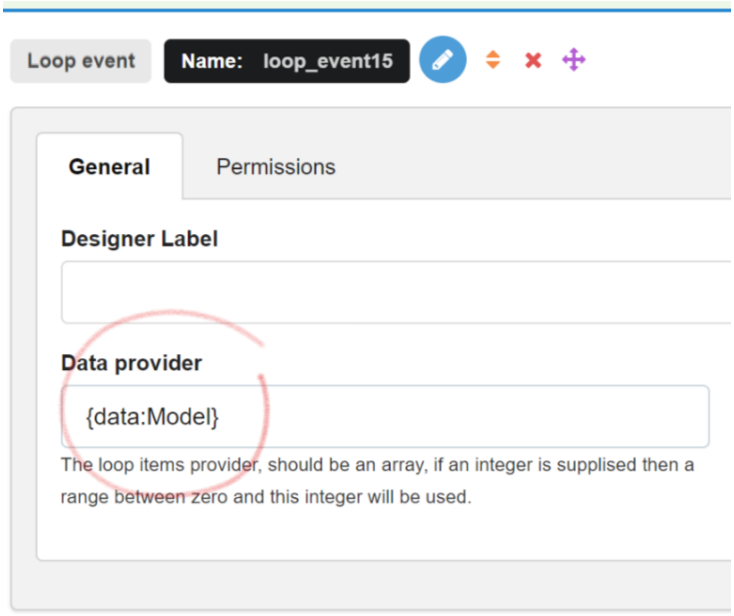
LOOP EVENT

The Loop Event action receives an array as the data provider and runs a set of actions provided in the “Body” event times the number of the elements in the array.

A frequent usage of this action is when we need to save multiple items into a database table, we have to add a “Save data” action inside the “Body” event of the “Event loop”.



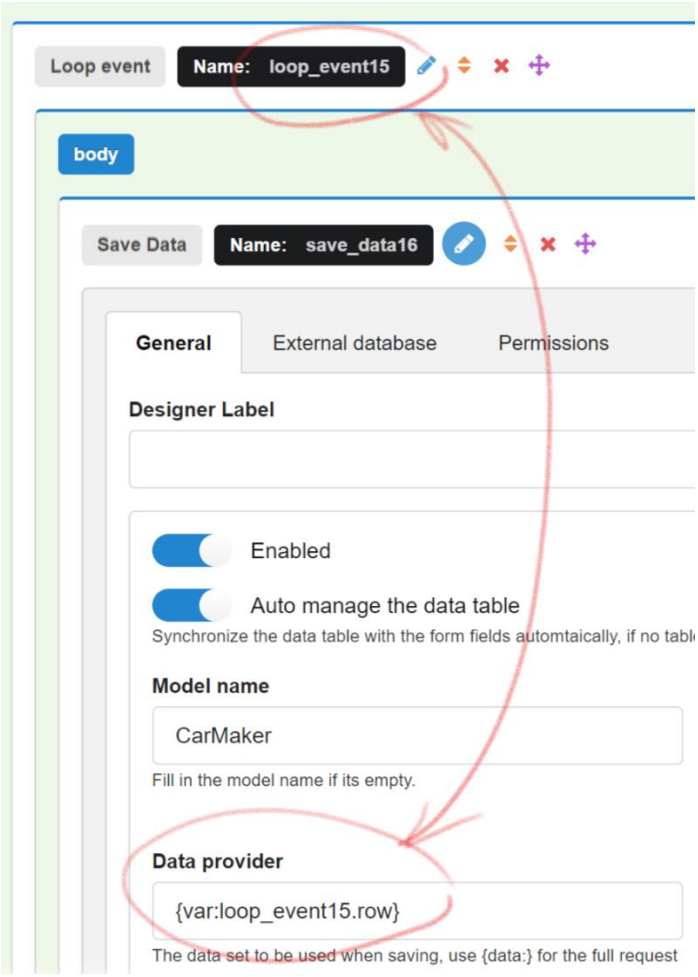
The **Loop event** data provider should be set to a variable holding an array



The image below shows the data array and the data structure of the “Model” key we are using as the data provider

```
Array
(
    [option] => com_chronoforms6
    [cont] => manager
    [chronoform] => first-form
    [event] => example
    [Model] => Array
        (
            [0] => Array
                (
                    [brand] => BMW
                    [model] => M6
                )

            [1] => Array
                (
                    [brand] => Lamborghini
                    [model] => Aventador
                )

            [2] => Array
                (
                    [brand] => Citroen
                    [model] => Survolt
                )

        )

)
```

The “**Data provider**” of the “**Save data**” action can be set to the row value of the current iteration, {var:iterator_name.row} where the “**iterator_name**” is the name of the “**Loop event**” action, or as per the example it should be {var:loop_event15.row}.



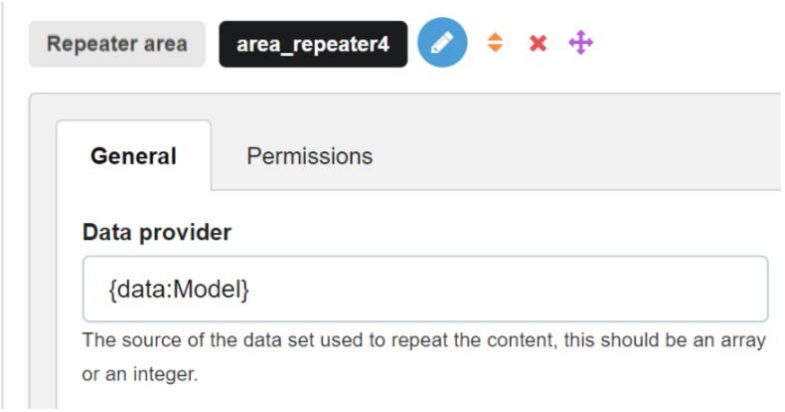
According to the example above, at the first iteration, the {var:loop_event15.key} will hold “0” and the {var:loop_event15.row} will hold the BMW array set shown below.

```
(
    [brand] => BMW
    [model] => M6
)
```

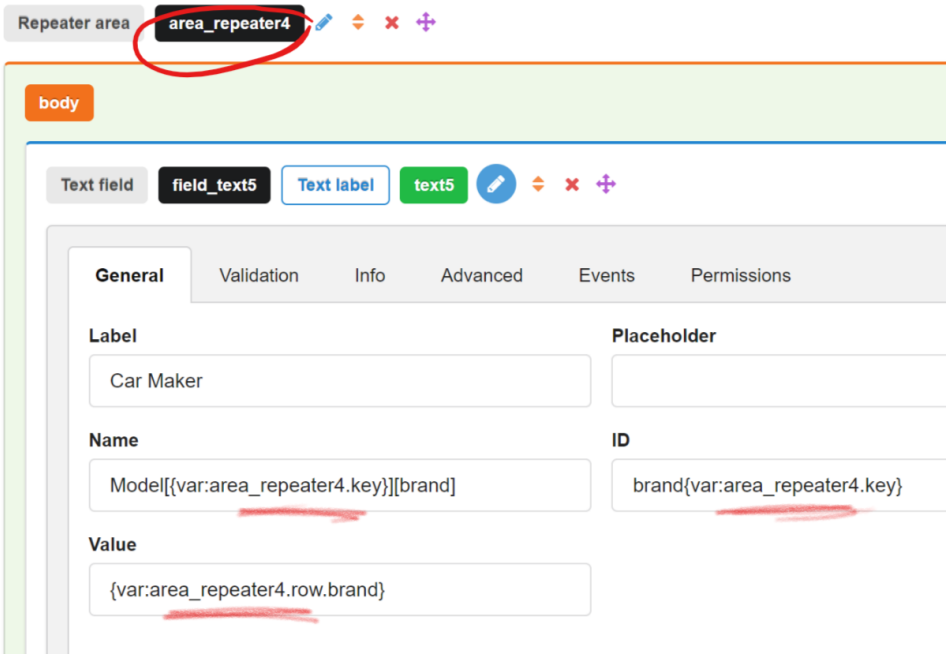
If our database table used in the “**Save data**” action has the fields “brand” and “model” then 3 new entries will be added to the table after the Loop event has finished processing.

REPEATER AREA

Similar to the Loop Event action, the Repeater area uses the Data provider array elements to repeat the elements in the “Body” section, following the same steps above but using a “**Repeater**” area instead of an “**Event loop**” we can have 3 text boxes for the 3 big car makers as shown.



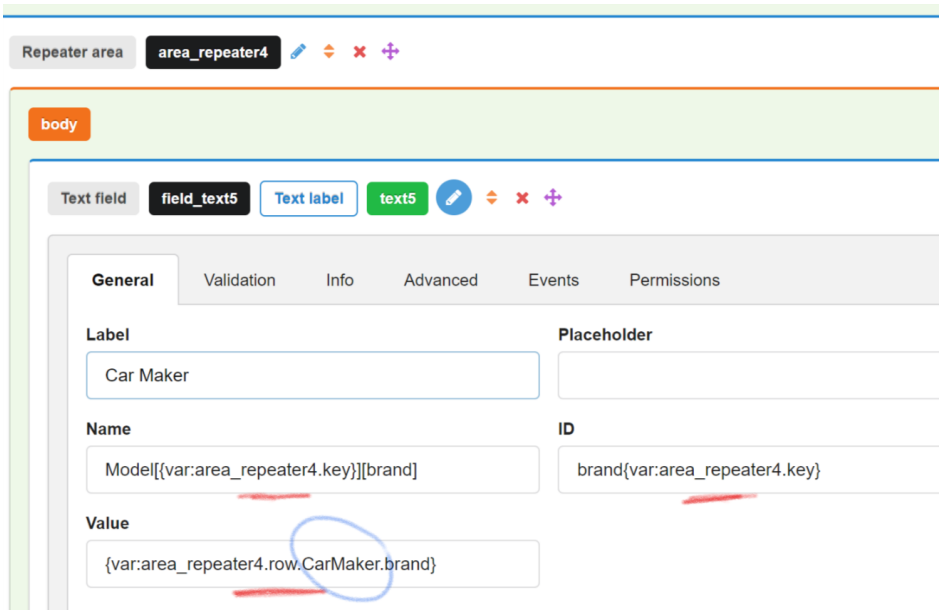
And since we need to use the “iterator_name” when calling the array items values, the text field setup would be like the following according to the data array structure:



However, when working with “**Read data**” actions, the results are retrieved in this structure:

```
(
    [0] => Array
        (
            [CarMaker] => Array
                (
                    [brand] => BMW
                    [model] => M6
                )

        )

    [1] => Array
        (
            [CarMaker] => Array
                (
                    [brand] => Lamborghini
                    [model] => Aventador
                )

        )

    [2] => Array
        (
            [CarMaker] => Array
                (
                    [brand] => Citroen
                    [model] => Survolt
                )

        )

)
```

And if we assign this new data set to the “Data provider” of the “Repeater” then the text field setup would be the same but we would need to change the variable path used to get the value since there is an extra “CarMaker” key in the path:



In other words, the {var:iterator_name.row} has a different value now:

```
[CarMaker] => Array
    (
        [brand] => BMW
        [model] => M6
    )
```

And we have to account for this when calling the brand or model values.

CONDITIONAL LOGIC AND SWITCHING

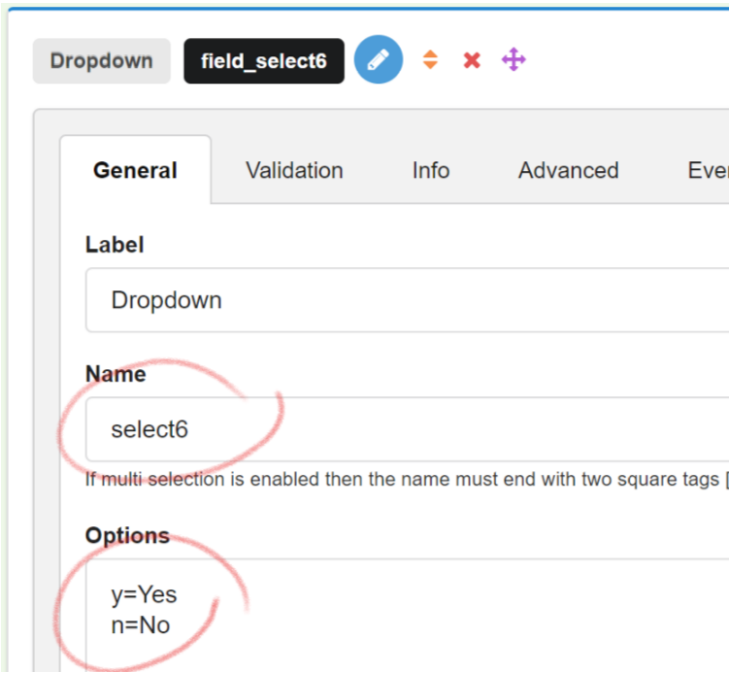
In Chronoforms version 6 it’s possible to conditionally switch content or actions based on a data source.

For example, in some cases, you may want to trigger a specific action when the user makes a specific selection, this can be done using the "**Event switcher**" action.

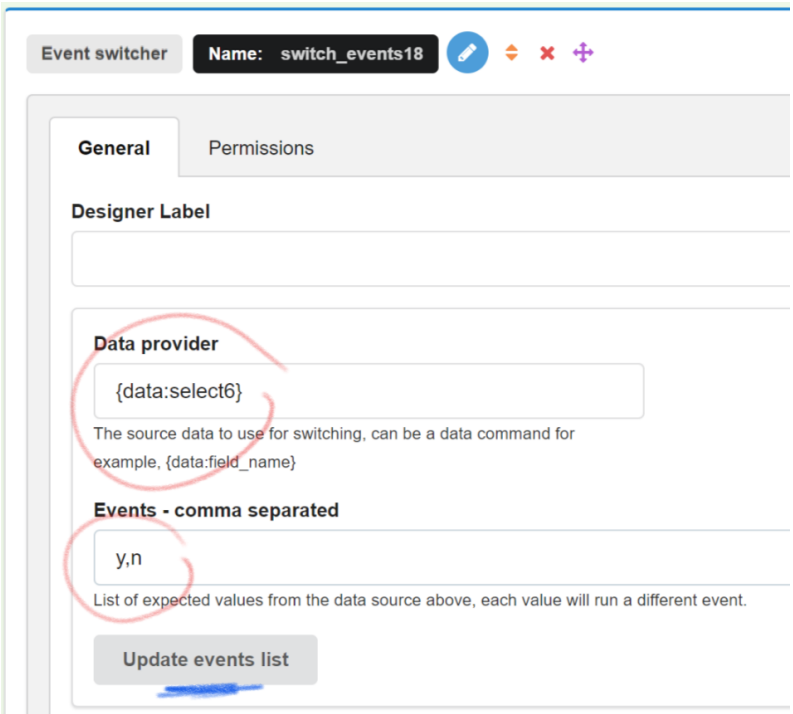
EVENT SWITCHER

The Event switcher matches the value of the provided data source against the names of provided events, when the value of the data source matches the event name, this event is triggered.

For example, having a dropdown with 2 options yes and no, we can do different actions based on the choice made by the user

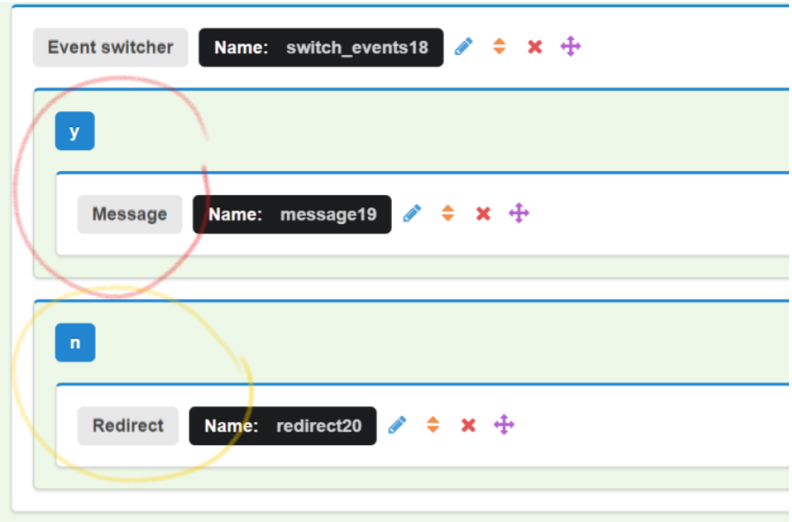


The "*Data provider*" of the "**Event switcher**" should get the dropdown value, and the events names should match the values we expect



You should click the "*Update events list*" button whenever you change the events list in order for new changes to take effect.

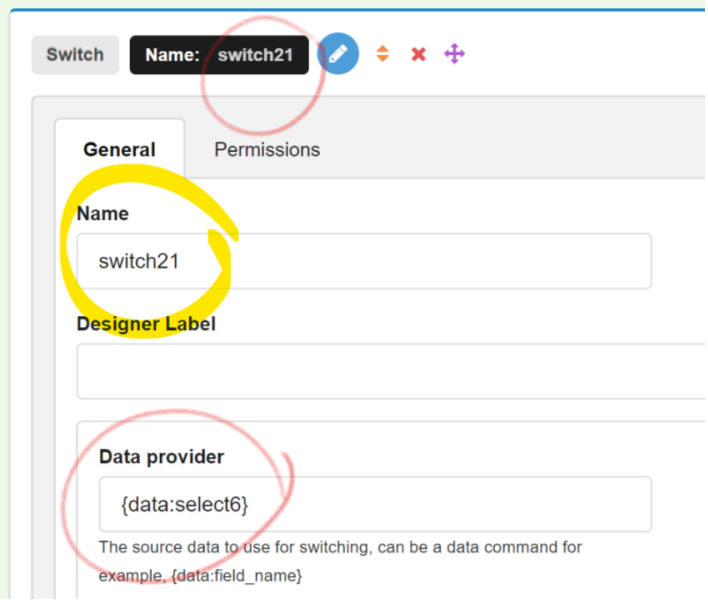
Now let's say that we want to display a message if the user selects "Yes", but to redirect the user if they select "No", we can do this easily using the following setup:



SWITCH

Alternatively, in some cases, you may need to just switch a specific content inside an element or inside the email body, or even switch the recipient of an email, this can be done using the "**Switch**" action.

Similar to the "**Event switcher**", you need to set the "*Data provider*", let's use {data:select6} again to capture the value of the dropdown field.

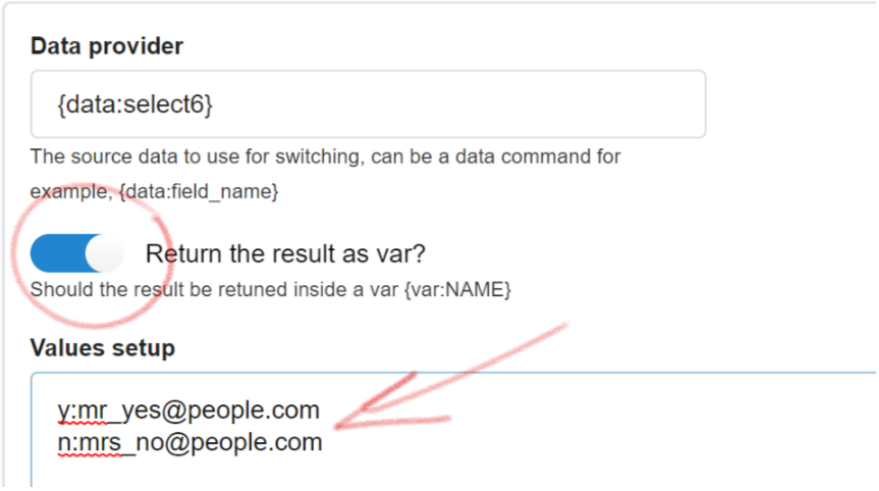


The "**Switch**" action is one of the actions which will most probably be called using a shortcode {var:action_name}, these actions usually have a setting to change their names, we have also set the "*Data provider*" to the dropdown field value.

Next we need to add the switching logic, the important point here is the "*Return result as var*" setting, and it can either be:

- Disabled, the default value, this is suitable if you want to output the switching result instantly after the action is processed, for example when the result is supposed to be displayed on the web page.
- Enabled, this is necessary when we need to store the result for later use until we call it using the {var:action_name} shortcode, for example in the email body or recipients box.

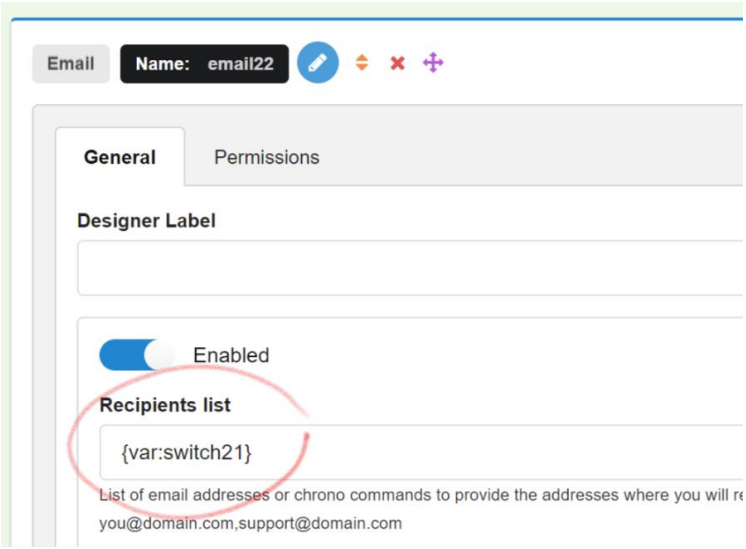
Let's enable the "*Return result as var*" setting and use 2 different recipient addresses based on the select dropdown value



The values setup should be in multiple lines, 1 expected value followed by a colon then a return value.

After the "**Switch**" action has run, we will have one of the 2 addresses in the {var:switch21} variable based on the selection made by the user.

We can now use this shortcode in the "*Recipients*" box of an email action:

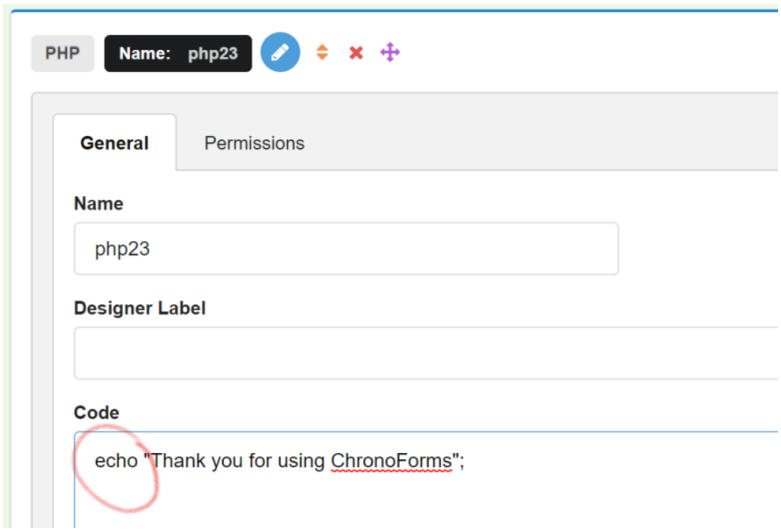


PHP CODE

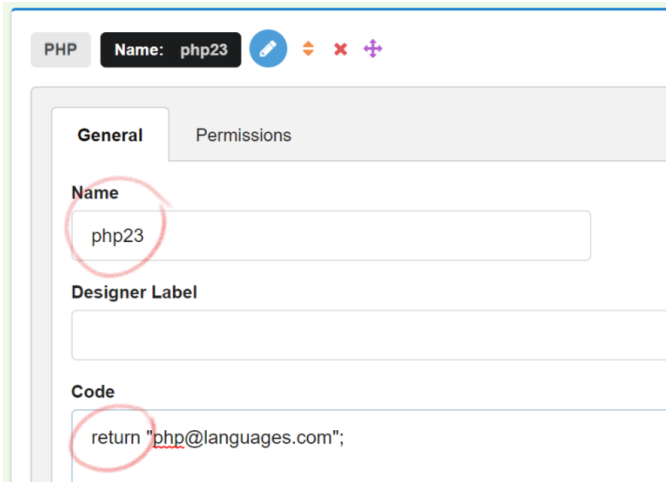
In some cases, you may have your own switching logic which you need to apply, this can sometimes be possible only using the **PHP** action.

In the **PHP** "*Code*" box you need to use your PHP code with OUT the php tags <?php or ?>

Similar to the "**Switch**" action, your code can either display a result directly by using the "echo" function or set the result in var named after the action name by using the "return" function, for example, the following code would display a message once the PHP action has been processed:



But the code below would not display anything, instead, the address returned is available inside the {var:php23} variable.



The following functions calls can be handy when working with PHP code in your form:

Code

```
$this->data("field_name", "default_value"); // get request or field value
$this->data("field_name", "value", true); // set request or field value
$this->get("var_name", "default_value"); // get var value or action result value
$this->set("var_name", "value"); // set var value
```

#MULTI PAGE FORMS

You can have multi page forms in Chronoforms v6 using the "**Multi page**" action.

FIELDS SETTINGS

NAME

The field name is what identifies your field data after the form is submitted, its strongly advised to always use a meaningful name, representing the function of your field, the field name, should be, in most cases, unique and should not be the same as any other field name in the form.

The field name should not contain any spaces or special characters, apart from dashes and underscores, – or _.

You will need to use the field name when you want to get the value of some field after the form has been sent using the data shortcode {data:field_name}

If your field supports the “Multiple” setting (Dropdowns, checkboxes and file fields) then you will need to add 2 square brackets to the end of the field name: field_name[]

If you are going to use the form for data saving AND data reading, then its advised to use the following naming convention: Model[field_name] as it makes reading data into the form fields easier.

ID

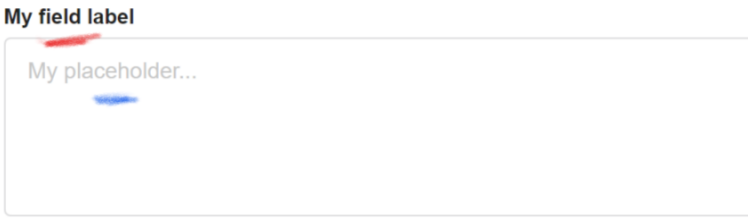
The field id is usually needed for JavaScript code identification purposes, it should be unique with no spaces or special characters, but it does not have to be meaningful, unless you are going to use it in custom JavaScript coding.

LABEL

The field label is the text displayed beside the field to clarify the field function for the end user.

PLACEHOLDER

The field placeholder is the text displayed “inside” the field to clarify its function, the placeholder will disappear once the user starts typing into the field.



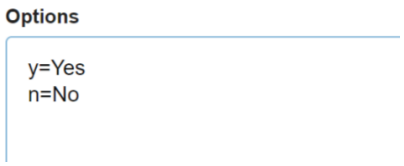
VALUE

The field value is the field’s initial value when the form is first displayed.

OPTIONS

Dropdowns, Checkboxes and Radios have the “*options*” field, which is the list of values and texts of the selections displayed by this field.

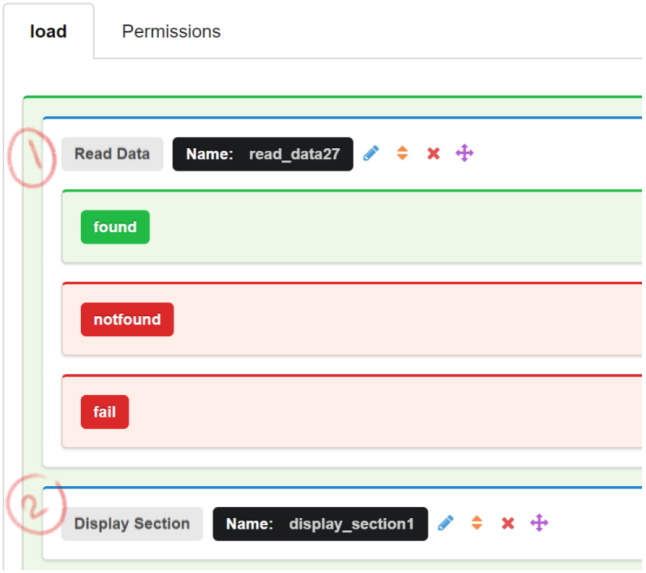
The options should be added in the format below:

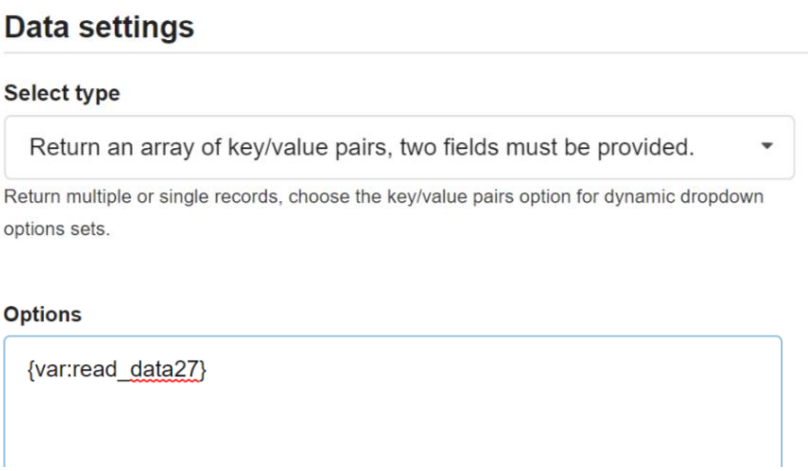


To have an empty option as the first dropdown option, add “=Select” to the top of the list.

Please note that the selection value is what is going to be available in the data array after the form is sent, in the example above, **y** or **n** will be available, but “Yes” and “No” will be displayed to the user in the form.

You can load the options from the database by having a “**Read data**” action BEFORE the “**Display section**” action and having the “*Select type*” set to “*return an array of key/value pairs*”, you should then call the result var of the “**Read data**” in the “*Options*” box as shown below.





GHOST

Some fields may not appear in the data array after the form is submitted if they did not have any selections or changes made, to avoid this issue which has some negative side effects, there is a “Ghost” feature available for these fields, it should be enabled by default and in most scenarios you should keep it enabled.

The ghost value is set to an empty string by default, which means that if no selection is made then send an empty string under this field name, but you can change it to something else if you have a specific requirement.

VALIDATIONS

EXTRA ATTRIBUTES

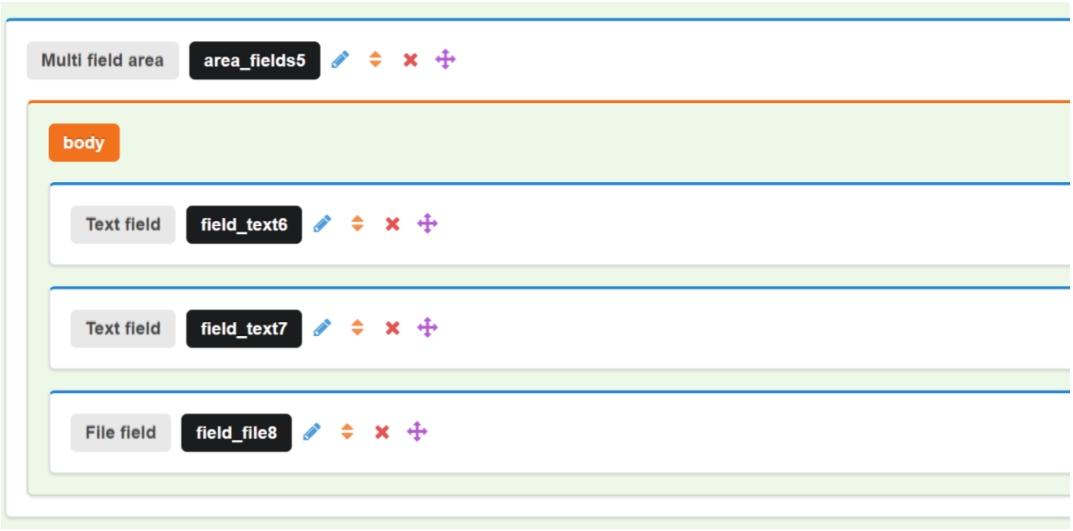
ADVANCED FORM STRUCTURE

AREAS

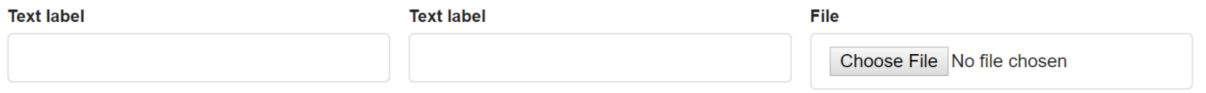
Areas are form elements which can host other forms elements, each area has a unique function and will render the children elements differently on the form.

MULTI FIELD AREA

Drag other elements into the multi field area to display them in a one horizontal row, on small screens the fields are going to stack over each other:



The output should now look like this:



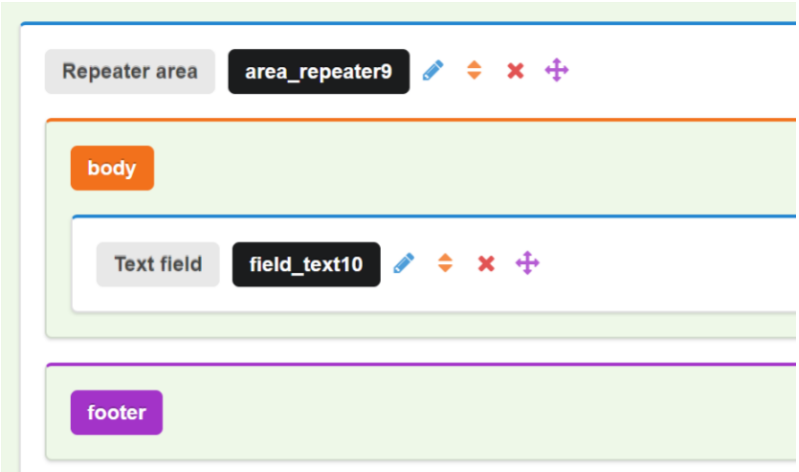
REPEATER AREA

The repeater area can have 2 functions:

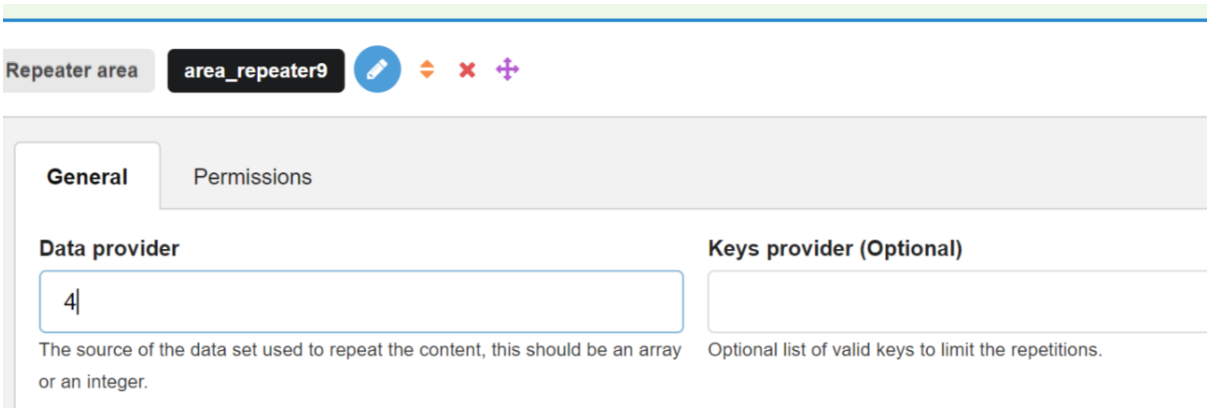
1. **Static repeater**: Repeat some content x number of times (based on the provider array count), this is static repeating and the content is already repeated when the form has been loaded.
2. **Dynamic multiplier**: this is done after the form is loaded and requires user input (a multiply button click).

Static repeating:

Drag a repeater area and drag one or more fields inside the body section of the repeater:



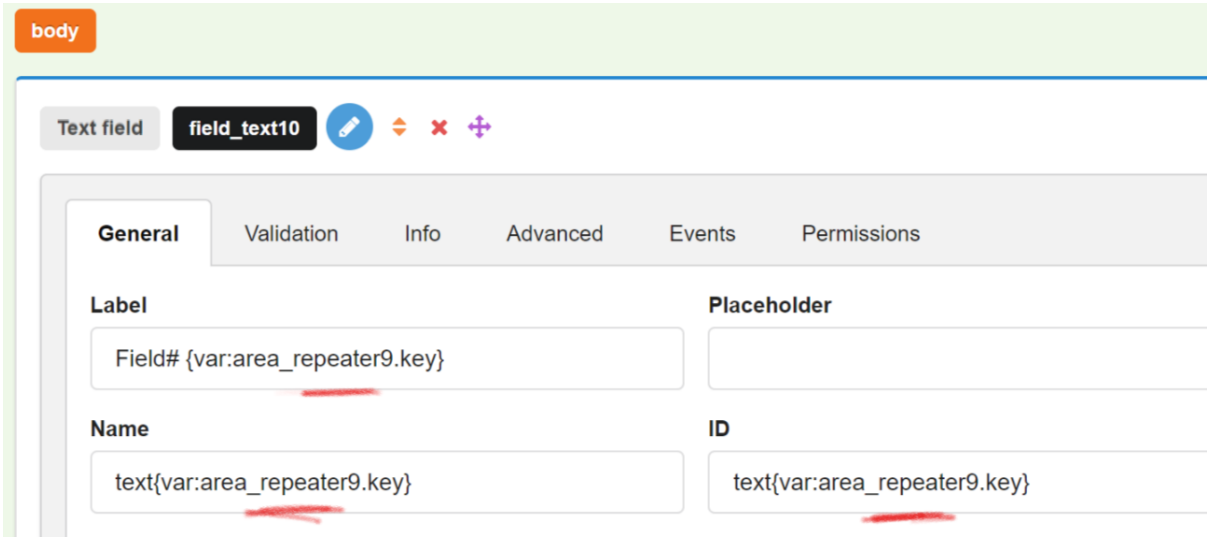
Open the repeater settings and set the data provider to an integer, say 4



The data provider expects either an array to use its elements for repeating the content in the body, but if an integer is provided then an array will be created containing elements from 0 to this integer.

If we save the form now, we will have 5 text fields displayed instead of 1, but all of them will have the same field name and label which is not desirable in most cases.

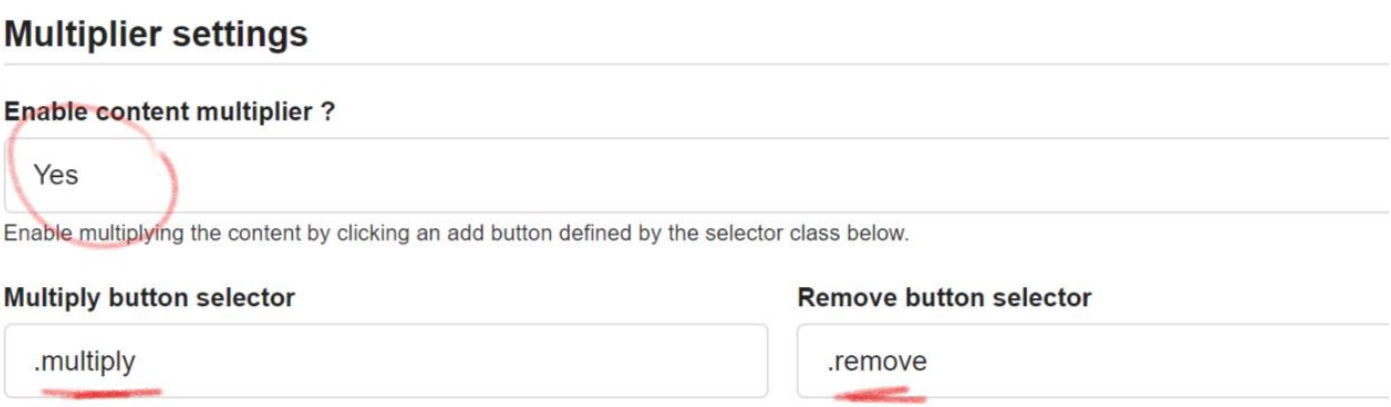
We can use the repeater's iteration key to give the field a unique name in each run, for this we need to use the repeater's shortcode {var:iterator_name.key}, which in this example should be: {var:area_repeater9.key}



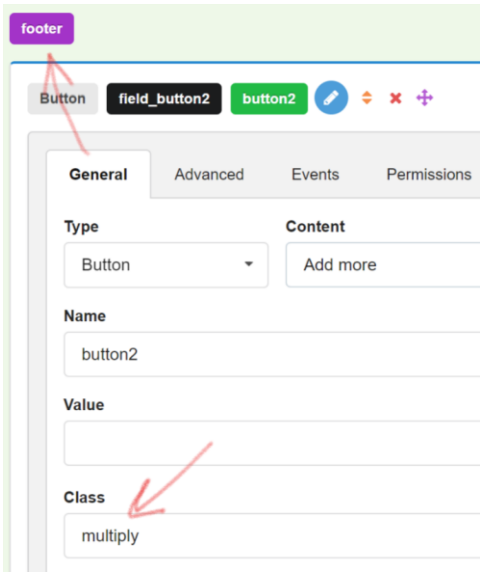
Dynamic Multiplier:

We can also configure the repeater to multiply the body section content when the user clicks a button, this is typically used when we want to let the user add more fields to the form dynamically.

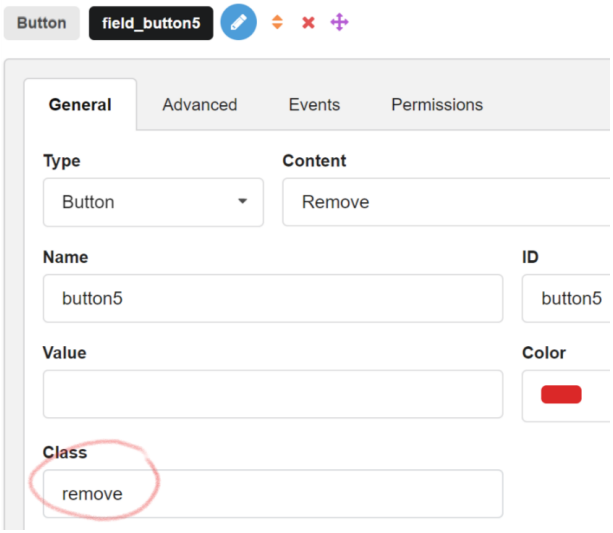
First the multiplier setting must be enabled in the repeater:



Please note the selector class which the multiplying button should have, its “multiply”, this should be added in the Class setting of the button, the button should also be inside the “*Footer*” section of the repeater.



You may also use a remove button with the class “remove”, but this button should be inside the body section of the repeater.



The result should now look like this: