Design Concept for a new Form System

John Liddiard (JohntheFish)

This document presents a design concept for a flexible form system implemented as an addon package for Concrete CMS. The concept has gone back to first principles to eliminate the endemic flaws and limitations of existing forms systems.

It is assumed the reader has some basic knowledge of Concrete CMS.

This design concept is not intended for implementation all in one go. Initial implementation would be targeted at specific applications, providing the components of this system necessary to those applications while ensuring further aspects of the concept are not precluded.

The addon should be extensible by further development within the addon package and by 3rd party */application* and */packages*.

Table of Contents

1 Technical Objectives....3
2 Architecture....4
  2.1 Front end form design....4
  2.2 Back end form handling....5
3 Implementation notes....9
  3.1 Where is the pipeline defined?....9
  3.2 Form blocks....9
  3.3 The overall form....10
  3.4 Submit button....10
  3.5 Form handler pipeline....11
  3.6 Form handlers....11
    3.6.1 Validate input fields....11
    3.6.2 Check spam (multiple handlers)....11
    3.6.3 Send email....12
    3.6.4 Save uploaded files....12
    3.6.5 Save to database....12
    3.6.6 Save to table....13
    3.6.7 Save to log....13
    3.6.8 Save as draft....14
    3.6.9 Save to Express object....14
    3.6.10 Message....14
    3.6.11 Save to document....14
    3.6.12 On error, On success....14
    3.6.13 End/Return....15
    3.6.14 Redirect....15
    3.6.15 Fire event....15
  3.7 Pipeline edit dialogue....15
  3.8 AJAX Forms....15

Copyright John Liddiard 2022

3.9 Dashboard.....15
3.9.1 Documentation.....16
3.9.2 Plugin management.....16
3.9.3 Submit block defaults.....16
3.9.4 Reports.....16
3.10 Supporting front-end functionality.....17
3.10.1 Report blocks.....17
3.10.2 Revision of forms.....17
3.10.3 Notification.....18
3.11 Garbage management.....18
3.11.1 Uploaded files.....18
3.11.2 Extreme Clean.....18

 Copyright John Liddiard 2022

1 Technical Objectives

The front end of forms should be easy to lay out in a page.

Form elements should be easy to design and easy to customise.

It should be easy to mix form components with other content, both for informational purposes and for design of a form.

Options for preserving visitor/user entered data and using it to re-populate further forms.

Full consideration of GDPR. No data held unless user/visitor is informed and consented.

Provision for visitors/users to view forms they have previously entered.

Provision for visitors/users to save and then use draft form data prior to submission.

Provision for visitors/users to edit and resubmit forms they have entered.

It should be easy to add new kinds of form components.

Open to extension with with both */application* code and 3rd party packages.

A basic Concrete CMS developer or theme designer should not need to learn any new skills to extend the form system.

For sites with many different forms, we need a mechanism for reusing groups of form components, the equivalent of stacks for form components.

Multi step forms.

User/visitor extendible lists of field(s), along the lines of ‘add further person’ to an entry list of people.

Data saved should be tangible. Data associated with a form submission should be saved in a single row of a single table. This promotes reliability, speed and ease of further integration.

When a form is submitted, validation and further processing should be flexible. No submission action should be compulsory. Sending emails, saving of data, redirection to another page and any application specific processing should be easily configurable.

Protected against spam.

Easy addition of further form processing. Easy integration with further validation of form data. Easy integration with further destinations for form data, both in-site and to 3rd party systems.

Suitable for future addition/extension for payment processing.

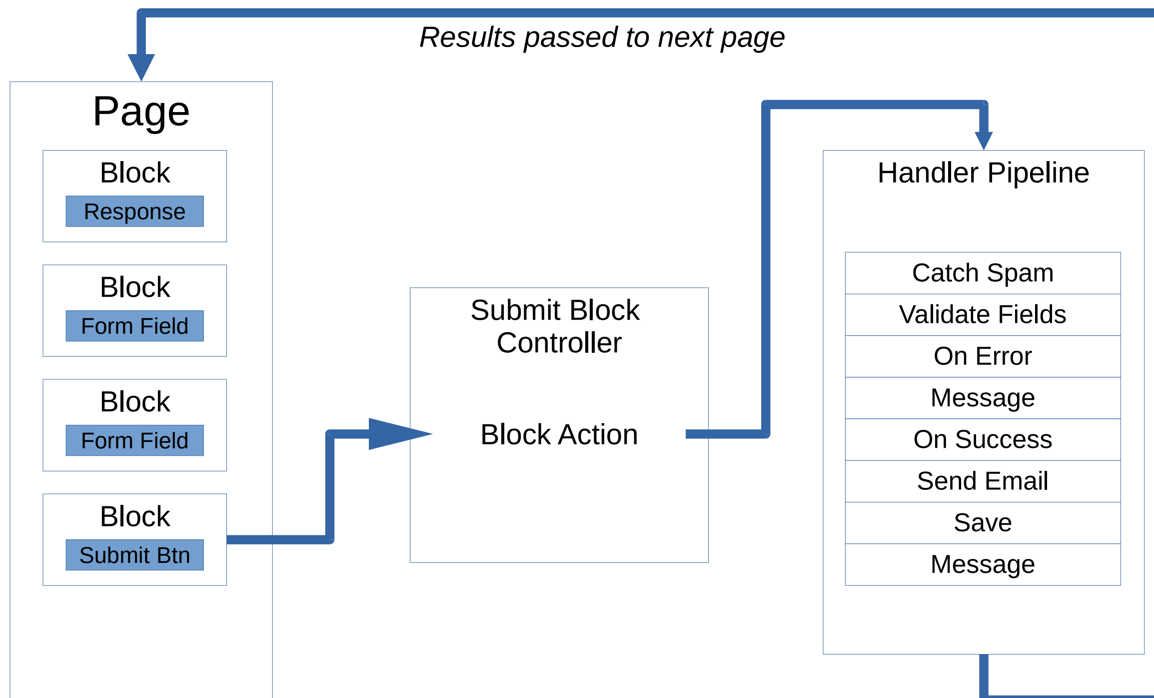
Make use of existing Concrete CMS core and addon structures to both reduce learning curve for editors and developers and to facilitate future expansion.

Adaptable for application specific form elements and processing.

Clearly documented.

Copyright John Liddiard 2022

2 Architecture



2.1 Front end form design

Form systems generally waste considerable code on attempting to re-implement capabilities for adding and editing input fields that already exist in Concrete CMS for adding, styling and laying out blocks.

Lets turn that round.

- If every type of form field is implemented as a block type, we can now create forms by adding a block for each field to a page.

By building a form from an arrangement of form blocks on a page we now have a system that any Concrete CMS site editor is immediately familiar with and can design with.

A new form system will need to provide code for each field type. Providing that code in a series of block types is no more trouble than providing that code through other mechanisms and is in many ways simpler.

Concrete CMS provides excellent functionality for block design, layouts and versions. We can add alternate block templates to form elements as we would with any block type. Blocks are easy for third party addition or extension.

Blocks can be pre-assembled into stacks. Add a stack of form input blocks to a page and we have re-usable sub groups in forms. Insert other types of block between form input fields and we have an extensive capability for presentation. Combine this with Universal Content Puller and we can have

 Copyright John Liddiard 2022

hierarchies of combined form elements.
https://marketplace.concretecms.com/marketplace/addons/universal-content-puller1/

Block templates and are the first thing any Concrete CMS developer learns to code. They are one of the easiest things to code. The underlying mechanisms are well established and reliable.

- Need a different layout or styling within a form field block? Code a new block template.
- Need a specialised form input? Code a block to provide that input.

In addition to the usual form fields, we can have specialised form blocks for:

- Submit buttons.
- Acceptance of degrees of storage of user/visitor data such as in-browser, on server etc. for GDPR compliance.
- Concrete CMS specialised widgets, such as page, file and user selectors.
- File upload.
- Password creation/confirmation, duplicate fields that need to match for validation.
- Authentication codes, such as originating from 2FA systems. https://marketplace.concretecms.com/marketplace/addons/two-step-authentication-advanced/
- Spam traps, such as a honeypot and Captcha.
- Managing visibility of dependant fields or groups of fields.
- Managing dependant field validation.
- Variable repeated individual inputs, number of inputs controlled by add/delete/sort(drag) buttons alongside the inputs.
- Variable repeated groups of inputs, number of groups controlled by add/delete/sort(drag) buttons alongside the groups.

Managing dependant field visibility and validation could be generic up to a point, but may need application specific blocks for anything beyond basic rules. Implementation of both will be a trade off between the complexity of a generic rule engine and the convenience of just coding what is required for an application in JavaScript and php.

The particularly complex issue here is a truly generic variable repeated groups of inputs. This could be implemented by variable repetition of a stack, where the form block provides controls to add/remove copies of the stack from the front end view.

2.2 Back end form handling

At the back end, when a form is submitted, the data goes through validation and then a sequence of actions such as saving the data and dispatching email notifications.

 Copyright John Liddiard 2022

Let’s generalise that into a configurable sequence or pipeline of form handlers, where each handler performs a discrete action in the overall process.

For multiple emails, insert multiple email handlers into the pipeline. For multiple messages, insert multiple message handlers into the pipeline.

The obvious handlers at this point of discussion are:

- Validate all individual form fields.
- Check for spam.
- Send an email.
- Save the data to a database table.
- Save an uploaded file to the file manager.
- Save the data to a user (variants for user attributes, cookie or session) so it can be used to populate subsequent forms.
- Show a message (actually prepare a message to be shown on return).
- Redirect to page (actually prepare to redirect on return).
- Black hole.
- End - return to page (which would also show any messages and handle redirection).

We also need to consider logic in the pipeline of handlers, so we can run different handlers when validation fails. We don’t need complex logic or loops, so a simple conditional jump as a handler would be enough to facilitate linear logic. Something along the lines of.

- On Error, process the following handlers. Else jump forward to On Success and continue.
- On Success, where handling continues if there is not an error.

The pipeline would then be built with handlers for the error, followed by handlers for success. This could be further generalised into conditional logic handlers.

- If, where form field matches condition, process the handlers. Otherwise (false), jump forward to the Else handler (or Else-if or End-if) and continue.
- Else, where the if jumps forward to when the form field does not match a condition.
- Else-if, an alternative for a subsequent condition.
- End-if, where it all resumes, assuming previous conditions have not exited the pipeline.

Such single depth linear logic will not be complex to implement and is already considerably more flexible than anything that can be achieved with handling in existing forms packages.

The second part of making this back end form handling flexible for any forms requirement is to make it extendible with further general purpose and application specific handlers. We make the entire system of handlers extendible, so new handlers can be provided by further addon packages or application code.

 Copyright John Liddiard 2022

Unlike Express, a new handler will be a simple class that provides a method to show edit options and a method to do something with form data.

- Need a complex multi-field validation? Code a handler to provide that validation.
- Need to save data somewhere else? Code a handler to make that save.
- Need more complex logic? Code a handler or series of handlers to embody that logic.
- Need to send data to an external service? Code a handler to forward data to that service.

Which brings us to another consideration. External services may return further data, validation or errors such as a failure to respond. If that return is synchronous, we can continue in the same form handler pipeline, adding any failure to respond to an $error list object.

However, such a return could also be asynchronous, via a callback entry point. Such an asynchronous return is not unusual, especially for payment processors.

This is getting a bit too far ahead for an initial implementation, but we need to make sure the architecture facilitates suspending a handler pipeline and then routing a callback to resume the pipeline with incoming data. In the meantime, the browser submitting the form may need to show a response pending message or page.

- Await response (record the pipeline state, be ready to resume it when the response is routed)
- Response pending (return for now, but also prepare a browser to receive a subsequent message)

Executing both such handlers implies a concurrent fork of the pipeline, so perhaps we need another logic item for concurrent branches, or alternatively the mere presence of an *Await response* after an exiting step can be used to imply such.

A further consequence of asynchronous response is there could be multiple form processing pipelines running concurrently and awaiting responses that could arrive out of sequence or may not arrive at all.

As noted, this is not functionality for an initial implementation, but a possible growth of functionality that must not be precluded by the initial implementation.

A similar asynchronous situation could also be of use internally in a site. For example, a form handler plugin that starts a workflow and awaits a confirmation from an administrator before continuing, or for two-factor identification.

The handler pipeline could get lengthy and there may be opportunities for reuse of groups of handlers within the pipeline. As the handler pipeline is not built out of blocks, we cannot place groups of handlers into a stack. Hence another consideration for the future would be a dashboard interface to create macros of handler sequences as fragments of a pipeline, and a meta-handler plugin to:

- Use macro, to run a previously constructed macro sequence of handlers.

Many form applications will only require a basic pipeline of common handlers. A sequence that essentially replicates only what existing forms systems do. A pre-constructed default sequence to

 Copyright John Liddiard 2022

pre-populate a new form handler pipeline will simplify this for site administrators that do not need to know any more about the pipeline and use this new forms system primarily for its front-end blocks.

Copyright John Liddiard 2022

3 Implementation notes

Having considered the general architecture of our new forms system, here are some thoughts on implementation details.

3.1 Where is the pipeline defined?

Once we consider that a form could have more than one submit button, and consequently more than one handler pipeline, the direct way to associate a handler pipeline with a form is to configure the handler pipeline inside each form *Submit block*. Hence each form *Submit block*, in edit mode, builds the handler pipeline for that form and that specific submit button. In view mode it presents the submit button and, when clicked, will then run the configured handler pipeline.

Contrary to this, initial design thoughts had considered a dashboard page to configure a handler pipeline for a specific form. However, once the concept of a *Submit block* was established, the *Submit block* became the obvious and convenient place for an administrator to define the handling for that submit button.

One or more dashboard pages may still be implemented, for setting defaults and managing named settings as pre-assembled sequences of form handlers.

3.2 Form blocks

As noted, each form element will be implemented as a block. Much of the block infrastructure will be common across many form element blocks, so all form blocks can inherit from a base/abstract form block, which in turn inherits from the core block controller class. This base can also be used to flag that a block is a compatible form block.

In the add/edit dialogue, all (or most) form blocks would provide options for:

- Form name, to associate a field with a form when there are multiple forms on a page. The input element's form="form_name" attribute.
- Input name, the name of the field within the form. The input element's name="input_name" attribute. Behind the page this will also be modified with some application specific encoding.
- Persistence, to enable auto-population of a field with existing data from user/session/cookie/previous form submission, either local to the page or to accept globally from any page in the site.
- Label, an associated html <label> element
- Form state, what form states the block is shown or rendered in, such as after a successful submission or in a multi-step form.

Other reasonably common options

- Place holder text

 Copyright John Liddiard 2022

- Numeric or text input constraints
- Data edit options to manage constraints under which an already saved field may be edited. For example, to lock a submitted field from subsequent editing in the dashboard reports section or when the form is edited by the original visitor/user.
- Field prefix/suffix. This is more theme/framework specific, the equivalent of bootstrap form-group prepend/append.

In addition to the usual add/edit/view methods, the controller for a form block may provide methods that can be called by the handler pipeline.

- *extractSubmissionData*(). For use where the front-end field(s) do not map directly to the form data item, this will return the actual form data item. For example, when a file is uploaded, the data saved with a form would be the fID. The parent (default) method will simply return the field value.
- *validateSubmission*(). A back end validation of the form value, returning an $error object if it fails. The parent (default) method will return null.

3.3 The overall form

The input elements of a form will be rendered by blocks that could be nested several levels deep in structures of other html elements, so simply wrapping as a whole with <form> and </form> tags could result in illegal html.

The most flexible solution is to only provide the <form></form> with the Submit button. All form elements will be left out in the wild and associated with a form="form_name" attribute. With such a strategy, further meta-data for the form structure can also be attached to the submitted data.

When there is more than one form on a page, the *Form name* can be used to associate fields to specific forms.

The data submitted will then need to be separated from meta data when the form is handled on the server. Hence the *extractSubmissionData*() method above.

3.4 Submit button

The block type that renders a *Submit button* will control the overall form processing. The form data data will then routed back to the block controller through the usual Concrete CMS block action mechanism.

The pipeline of handlers for a form will be already configured by the submit block edit dialogue.

The action method in the submit controller will then instantiate block controllers for the individual form elements. The form data, metadata and blocks will then be fed into the form handling pipeline.

Whilst this is described as being within processing from the submit controller, it will actually be structured as a handler pipeline subsystem of classes, so facilitating other entry points to the form handling pipeline such as routing an asynchronous callback.

Copyright John Liddiard 2022

3.5 Form handler pipeline

The plugin system can use the proven plugin system used in other addons such as Universal Content Puller, Omni Gallery, Extreme Clean and many others.

https://c5magic.co.uk/addons/plugin-system

Each handler plugin will provide a *handler_action()* method that accepts:

- Submitted data
- Configuration of handler
- The form pipeline object providing access to the list of handlers, metadata and accumulated $error list

Then return a structure that includes one or more of

- $error response object
- Messages and other data to be returned to the browser
- Instructions to the pipeline handler, such as exit or to jump forward to another handler

The pipeline system will then act on that return and either move on to a subsequent handler in the pipeline or end and return a response to the browser.

3.6 Form handlers

Some design thoughts for various form handlers. This is by no means a complete list. The purpose is to demonstrate how the design concept works and to act as a validation of the architecture by considering the implications of implementing each handler.

3.6.1 Validate input fields

The *Validate input fields* handler will pass each item of form data to the *validateSubmission()* method of the respective block and collate any $error object responses.

At this stage there will already have been in-page Validation through html attributes of form elements and JavaScript. However, with public facing forms such validation cannot be trusted and will as a minimum need to be repeated in the php validation.

3.6.2 Check spam (multiple handlers)

Checking spam will actually be a series of handlers that can be used or not according to processing requirements.

- IP blacklist check
- Spam service check (using the core configured spam service)
- Banned word check

 Copyright John Liddiard 2022

Notably not in this group is a captcha. Captcha will be implemented as a form field block and validation of the captcha handled by that block's *validateSubmission*() method.

A honeypot would also be handled in a similar way in the associated block's *validateSubmission*() method.

3.6.3 Send email

Most form submissions will use two instances of a *Send email* handler. The first to notify an administrator and the second to acknowledge the submitter of the form.

The edit dialogue (as shown when configuring a submit block) will allow entry of email subject, body, from, to, cc fields etc. Each such field will accept *{{place_holders}}* for named items from the form and other site and user data, so when a form field is named *my_email_address*, the 'to' field of the response email can be filled with *{{my_email_address}}*.

Email will then be sent using the Concrete CMS mail connection. The new forms system does not directly implement any mail interface.

3.6.4 Save uploaded files

When a form includes uploaded files, these will need to be saved to the Concrete file manager and an fID noted prior to further saving of data.

The edit dialogue will provide options for structuring name, description, location, file set and attributes in a similar manner to Snapshot
https://marketplace.concretecms.com/marketplace/addons/snapshot

3.6.5 Save to database

Most form submissions will involve saving data. The default *Save to database* handler will differ from other forms systems by saving all data to a single table. The default form data table will have fields for

- Form name
- cID
- Submit date-time and user ID
- Update date-time and user ID
- Form data, as a JSON data type or as a string of JSON encoded data

MySQL supports a JSON data type that can be searched and queried within the JSON. We don't need to be constrained by a preconfigured set of columns for actual form data or to spread the data across multiple tables for each type of data.

https://dev.mysql.com/doc/refman/8.0/en/json.html

Where a database does not support the JSON data type, data can be serialized as JSON and saved as a text string.

 Copyright John Liddiard 2022

Thus data storage and subsequent searching will be:

- Fast. One record equals one read or one write.
- Collated. All the data for a form submission is in one row of one table.
- Atomic. Even without enclosing in a transaction, data it is either saved or it is not.
- Tangible. A site admin can look at the database in phpMyAdmin and actually see a list of data entries or export a list of entries.
- Reliable. No route for skewed or otherwise broken data.
- Searchable. No messing about with separate search indexes or complex table joins. All the data is in one place.
- Easy to use and work with in further development.

An option would be to replace into the table, so a user would be updating an existing record rather than appending a new record to the table.

A special case will be saving of an uploaded file, where the file will be saved to the file manager and the fID saved with the form data.

3.6.6 Save to table

This is an additional or alternate save strategy, where saving is routed to a specific table, not the general purpose table.

Whilst such a handler could be engineered as a single handler with the table name as a parameter, that would leave the configuration open to dangerous mistakes such as writing to core tables.

In practice, *Save to table* would be a base class for further handlers to be coded from, where an application could implement a handler *Save to My Application Specific Table*.

Such a table schema will need to correlate with form fields added to the page, so we also need a strategy for saving with:

- Columns missing data. Provide a default value or note an $error.
- Data that does not map directly to a column. Such data can either be discarded or we can have a generic everything else column in JSON, or again note an $error.

This category of save handler may be a good case for integration with Entity Designer
https://marketplace.concretecms.com/marketplace/addons/entity-designer

3.6.7 Save to log

An additional or alternative save strategy. Serialise the form data as JSON or as name/value pairs and save it as text to a channel in the Concrete Log.

Copyright John Liddiard 2022

3.6.8 Save as draft

Saves form data somewhere private to a user such as an attribute, session or cookie, for re-population later. This could be a reasonable use for a submit handler pipeline that does not fully validate.

3.6.9 Save to Express object

As the fundamental limitations of Express based forms is one of the reasons for this new forms concept, the idea of Save to Express object seems an oxymoron. Nevertheless, where Express is used for purposes other than forms, perhaps there could be a application for such a handler.

3.6.10 Message

The *Message* handler is largely similar to the *Send email* handler, but the message will be returned to the browser for display. Display will be delayed until handling is complete, so can be shown on the original submission page or on a redirected page.

For the *{{place-holders}}*, this will also need *{{error-list}}* for cases where it is used to show an error.

The edit dialogue will provide configuration options for the level of message such as information, success, primary, secondary, warning, error.

Such levels are obviously biased towards bootstrap based themes, but can also be mapped to encompass common notification categories for other themes and frameworks.

The actual rendering of the notification will be managed by the *Message display* block. Hence styling of *Message* for a theme can be accommodated by a block template for the *Message display* block.

3.6.11 Save to document

A combination of *Show message* and save, the concept of this handler would be to create a document such as a pdf file based on a rich text template and *{{place-holders}}*, fill the template from the form submission and then save that document to the File Manager.

The document could then subsequently be attached to a message with a *Send email* handler.

3.6.12 On error, On success

The *On error* handler simply needs to check the ongoing $error object for messages. If error messages are present then pass them forward to whatever is next, likely a *Send email* handler or *Message* handler.

If no errors are present, jump forward in the handler pipeline to the next *On success* handler.

 Copyright John Liddiard 2022

3.6.13 End/Return

The form handler pipeline is finished, so return to the form page. All accumulated handler response instructions will be returned and processed. This will act on those instructions, such as displaying a message, redirecting and setting new form states.

3.6.14 Redirect

Pick a page to redirect to when the handler pipeline ends.

3.6.15 Fire event

Create and fire a Concrete CMS event, then accept what is returned by the event.

Firstly, this is to facilitate an alternate form of extension / integration, for developers that are happier coding event handlers than coding from a boilerplate for a plugin.

Secondly, this could be used to fire an already established Concrete CMS event and hence trigger any associated event handlers.

3.7 Pipeline edit dialogue

The edit dialogue for a pipeline will be shown within the dialogue for the associated *Submit block*. It will be based on an extensible and sortable list of handlers, where new handlers are picked from a list of all handler types installed.

Each handler will present its own edit form/options specific to the purpose of the handler. Within the list of handlers, a specific handler's edit form would normally be hidden/collapsed, then shown as an accordion/expander or as a popup.

The concept of a pipeline of form handlers is designed to provide the flexibility needed for advanced applications. Nevertheless, for many applications the pipeline will be the same across all forms. The chore of configuring a *Submit button* block can be simplified by enabling a complete set of pipeline settings to be loaded from a default, or by exporting/importing settings.

Such functionality already exists in the plugin system in use by other addons, but only for fixed sets of plugins. The mechanism would need enhancing to work for a variable length list of plugins.

3.8 AJAX Forms

Extending this mechanism to AJAX forms will require an AJAX response from the *Submit button* block and then JavaScript attached to the block view to make the AJAX request and handle the response.

3.9 Dashboard

Anticipated dashboard pages.

 Copyright John Liddiard 2022

3.9.1 Documentation

The plugin mechanism adopted from previous developments includes a built in documentation capability. Each form handler plugin will provide general user/edit documentation to be available in edit dialogues and in the plugin management dashboard page.

This approach can be extended to form field blocks, where each form field block provides built in general user/edit documentation to be shown in a dashboard page.

3.9.2 Plugin management

The plugin system already has functionality for creating a dashboard page to manage form handler plugins. Lists of installed plugins, permissions and updates.

3.9.3 Submit block defaults

The plugin system already has functionality for creating a dashboard page for managing defaults and global settings for a specific block that uses the plugin system. In this instance, the block in question is the *Submit block*.

This can be used to create a dashboard page to configure default/generic settings for the *Submit block* including default settings for the plugins configured by the *Submit block*.

As noted above, such functionality is currently only developed for fixed sets of plugins. The mechanism would need enhancing to work for a variable size lists of plugins.

3.9.4 Reports

The basic reports page will be specific to the default form table/store. It will show a list/search page that shows all submissions of all forms and will have options to filter by form identity and values. Any individual submission could then be clicked to view or edit the data as JSON.

For the future, a more directed editing system will be a more involved process because the form input elements are front end blocks. On the basis that editing a previously submitted form from within the dashboard does not need the chrome or layout of the original page and can show only the form names against input elements, the default block views may render in the dashboard without modification because core/default themes are both bootstrap. Form block views could require some conditional processing for this, in the same way as many blocks already implement conditional view processing when in edit mode.

For special cases that are not compatible with dashboard rendering, we could add a /template/dashboard.php to such form blocks and use that template preferentially for the dashboard view.

Where a form is saved into the Concrete Log, the core Logs page can be used to review submitted data.

Where a form is saved into a dedicated table, it will be up to the provider of the associated form handler/table plugin to implement reports. Where Entity Designer is used, that addon already generates dashboard pages and front end blocks for reviewing and managing records of data. https://marketplace.concretecms.com/marketplace/addons/entity-designer

Copyright John Liddiard 2022

3.10 Supporting front-end functionality

Here we discuss visitor/user facing front end functionality beyond the form blocks involved in building the actual form and submit button(s).

3.10.1 Report blocks

Basic report listing and individual item viewing blocks can be implemented in a similar way to the basic dashboard report. These blocks will not require a direct submission editing capability.

The block edit dialogues will provide options to select the form(s) reported, the submission(s) reported and the data fields shown.

It would also be feasible to add a source for Universal Content Puller and, where images are involved, for Omni Gallery.

https://marketplace.concretecms.com/marketplace/addons/universal-content-puller1/
https://marketplace.concretecms.com/marketplace/addons/omni-gallery/.

3.10.2 Revision of forms

Sometimes a visitor/user submits a form, then subsequently needs access to revise that form. Such revision can be configured by a report block providing a link to a revision form.

In many cases, revision of previously submitted forms would generally only be applicable to some fields in the form. Other fields will need to be protected and maintained. There may also be a different form handling pipeline between initial submission and update submission.

With that in mind, in most cases the revision form should not be a re-use of the entire initial form, but a dedicated form constructed to omit some of the initial fields, show some of the initial fields re-populated in a read-only form, and show the fields available for update re-populated in the fully editable form.

The not shown and view only fields can be handled by the original form blocks. We could have mode options within a form block or dedicated view templates. Both of these approaches can coexist. The associated form blocks will need to handle an input along the lines of ‘populate from submission xxx’ identifying the saved data to populate from. Such population needs to be secure from hacking the identity of what is populated.

In the form handling pipeline for update, an *Update to database* handler, or options in the *Save to database* handler, can specify the fields to update and the fields to be maintained immutable.

For circumstances where the original form is re-populated for editing and updating a previous submission, the form block views will need to automatically support different modes for hidden and protected, view only and editable items. This would be an extension of the functionality switch discussed for dashboard reports.

To control submission to a form handling pipeline specific to update, a *Submit block* could have modes to ‘show for new form’ or ‘show for update’. A pair of submit blocks can then flip-flop depending on how the form is used for initial entry or for revision and route the submission to the applicable handler pipeline.

 Copyright John Liddiard 2022

3.10.3 Notification

When a *Message* is passed back to page, that message needs to be shown somewhere. We also need to cater for a situation where a message is shown after redirecting to another page.

In both cases, adding a *Show message* block to the page will provide a location to display form response messages and to facilitate alternate block templates for the *Show message* block to be used to design how notifications are shown.

A *Show message* block could also be enhanced with a polling or push capability to show asynchronous responses from a form handling pipeline.

3.11 Garbage management

Compared to previous forms systems, garbage management for the proposed form system will be straight forward because all saved data is tangible. Clearing of old or redundant form records is simply a matter of deleting their rows in a database table.

3.11.1 Uploaded files

We will need to make specific provision for uploaded files.

Files will be uploaded first to a temporary location but not saved to the file manager. Any not removed by the respective form handler plugin or existing temporary file cleaners can be batch cleaned by a a job or task.

Files uploaded to the file manager associated may be from form submissions that subsequently become redundant or are selected for deletion.

- A file may need to be deleted with the form record.
- A file may need to be kept because it has been subsequently used.

Hence any provision in the dashboard for deleting a form record will need an option to also delete or keep associated files.

3.11.2 Extreme Clean

As previously noted, the architecture of the described form system is inherently a lot cleaner and easier to clean up after than existing form systems.

It shouldn't require any requirement for a dedicated cleaner from an addon such as Extreme Clean. Nevertheless, provision of a cleaner may be useful for the general admin workflow for some site owners, so bundling a cleaner plugin for Extreme Clean could be a nice touch.

https://marketplace.concretecms.com/marketplace/addons/extreme-clean1/

– END –

 Copyright John Liddiard 2022