Rate this page:

Creating a form

ON THIS PAGE

Before you begin
Create the app
Add the READ and WRITE scopes
Add a dynamic content macro
Create the form view
Create a route handler
...

Having trouble?
Ask for help on our Developer Community
Get help

This tutorial teaches you how to build a simple form on a Confluence page, storing responses in content properties. A content property is a key-value pair associated with a page. Each page can have an unlimited number of content properties, but each one is limited to 32KB.

In this tutorial, you'll learn about:

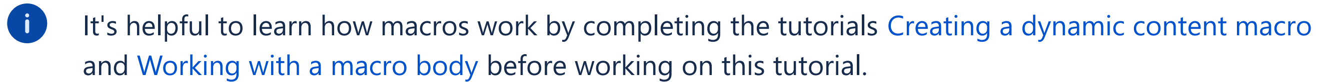
- Using a macro to display a form on a Confluence page
- Getting data from the form
- Storing data in content properties
- Retrieving and displaying data from content properties

Don't store private data such as credentials or authorization keys in content properties. A content property on a Confluence page can be edited by anyone with the ability to edit the Confluence page and can be retrieved by anyone with read access to the Confluence page.

Before you begin

Ensure you have installed all the tools you need for Confluence Connect app development by Getting set up with Atlassian Connect Express (ACE):

1. Get a cloud development site.
2. Enable development mode in your site.
3. Install ACE.



It's helpful to learn how macros work by completing the tutorials Creating a dynamic content macro and Working with a macro body before working on this tutorial.

Create the app

The first step is to use `atlas-connect` to create the app framework. This step creates a directory called `form-tutorial` that contains all the basic components of a Connect app along with a `generalPages` module, route handler, and view.

1. From the command line, go into a directory where you'd like to work and type:

```
atlas-connect new form-tutorial
```

2. Select **Confluence** in the menu.
3. When the command finishes, go into the `form-tutorial` directory and run `npm install` to install any dependencies for the app.
4. Add a **`credentials.json`** file in your app directory with your information:
    - **your-confluence-domain**: Use the domain of your cloud development site (for example, *https://your-domain.atlassian.net* ↗).
    - **username**: Use the email address of your Atlassian account ↗ .
    - **password**: Specify the API token ↗ .

````
``` json
{
   "hosts" : {
      "<your-confluence-domain>": {
         "product" : "confluence",
         "username" : "<user@example.com>",
         "password" : "<api_token>"
      }
   }
}
```
````

If you've completed the Getting started tutorial, you can copy the `credentials.json` file you already created.

Add the READ and WRITE scopes

In the app descriptor `atlassian-connect.json`, make sure your app has the `READ` and `WRITE` scopes. Example:

```
1  "scopes": [
2          "READ",
3          "WRITE"
4      ],
```

Add a dynamic content macro

You'll use a dynamic content macro to hold the form.

Paste the following lines over the existing `modules` element in the app descriptor:

```
1      "modules": {
2          "dynamicContentMacros": [
3              {
4                  "url": "/form?pageId={page.id}",
5                  "description": {
6                      "value": "Form for Confluence"
7                  },
8                  "name": {
9                      "value": "Form macro"
10                 },
11                 "key": "simple-form"
12             }
13         ]
14     }
```

You'll pass the `page.id` to the handler, to specify the page to store the content properties on. This is similar to the technique from Working with a macro body.

Also notice the `name.value` element, which specifies the name of the macro: *Form macro*. You'll need to know this name when you add the macro to a Confluence page.

Create the form view

Create the view for the form: in the `views` directory, create a file called `form.hbs` with the following content:

```
{{!< layout}}

<input class="text long-field" type="text" id="response-field" name="response-field"/>
<button id="submit-button" class="aui-button aui-button-primary">Submit</button>
<div id="form-response-list"></div>
```

The form is very simple, containing one text field and a button. Below the form is an area where we'll display form responses retrieved from content properties.

Create a route handler

The route handler displays the form view whenever the `form` URL is called. To create the route handler, add the following lines to `index.js` in the `routes` directory, under the line `// Add additional route handlers here...`

```
    app.get('/form', addon.authenticate(), async (req, res) => {
        const title = 'Simple form';
        const pageId = req.query['pageId'];
        res.render('form', {title, pageId});
    });
```

Now you've got the basics: a form with a field and a button, a route handler to get there, and a macro to hold everything. Let's make it all work!

Make the form functional

You'll create all of the logic that makes the form work by adding JavaScript to the layout. First, create an empty script by adding the following lines to the bottom of `views/form.hbs` :

```
<script>
$(function(){

})
</script>
```

The `$(function(){})` notation makes the script run when the page is loaded.

You'll modify this script to add three parts:

- An on-click function that creates JSON from the submitted text
- A REST API call that stores the JSON in a content property
- Another REST API call that displays all the data submitted so far

After completing these sections, you'll know how to grab data from a form, how to store data in a content property, and how to retrieve and display data from content properties.

Add the on-click function

Modify the `$(function(){})` to look like this:

```
$(function(){
  // submit button on-click
  $('#submit-button').on('click', function(){
    var newKey = "appkey_form_" + Date.now();
    var newEntry = $("#response-field").val();
    var jsonData = {
      "key":newKey,
      "version": {
        "number": 1
      },
      "value": {
        "response_values": [newEntry]
      }
    }
  })

})
```

When someone clicks the submit button, the function creates a new key and value for the content property using JSON:

- Key: a unique ID
- Value: the submitted text

Notice that the code adds `appkey_form_` to the beginning of the content property key. Adding an identifying string makes it easy to identify the content properties created by the form. One good approach is to use the app key (the `key` from your app descriptor) to make the identifying string unique, so that you can distinguish your content properties from other apps that use this technique to create forms.

We're also using `Date.now()` to add a timestamp, because it's an easy way to ensure the key is unique for each content property. This makes it possible to retrieve and address each content property individually.

Store the submitted text in a content property

Inside the on-click function, below the code you added in the previous step, add the following function:

```
// Store the form response
AP.request({
  url: "/rest/api/content/" + {{pageId}} + "/property/" + newKey,
  type: 'PUT',
  data: JSON.stringify(jsonData),
  contentType: 'application/json',
  headers: {
    Accept: 'application/json'
  },
  success: function(response) {
    console.log("Stored the form submission!");
    location.reload();
  },
  error: function(err){
    console.log("Error storing the form submission!");
      }
    });
```

This code uses the content properties REST API to store the key and value as a content property.

Notice that we have added success and failure handlers, which are always good practice.

At this point, the form is submitting text as content properties. In the next step, you'll retrieve content properties from storage and display them.

Retrieve and display the content properties

To retrieve the content property, you'll use the content properties REST API again.

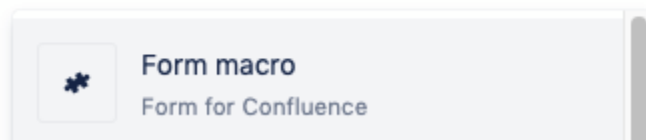
Before the submit button on-click function, add the following lines:

```
// Display previous responses
AP.request({
    url: "/rest/api/content/" + {{pageId}} + "/property",
    success: function(response) {
      var macro = JSON.parse(response);
      console.log("Retrieved form submissions!");
      var searchVal = "appkey_form_";
      for(var i=0;i<macro.results.length;i++){
        if(macro.results[i]["key"].startsWith(searchVal)){
          var response = macro.results[i].value.response_values[0];
          $("#form-response-list").append(response + '<br />')
        }
      }
    },
    error: function(err){
      console.log("Error getting form submissions!");
    }
  });
```

This code runs every time the macro is loaded, making a request to the content properties REST API to get all the content properties stored on the page. It then loops through the returned result looking for content properties that start with our key. Each time it finds one, it adds the text to a list of responses to display.

Try it out

1. From the command line, make sure you're in the `form-tutorial` directory, then type `npm start` to start the app.
2. Check that the app is running by going to `http://127.0.0.1:4040` in a browser and looking for `POST /installed`.
3. Edit a page in your Confluence Cloud developer site.
4. Type **/Form** or select **Form macro (Form for Confluence)** from the Macro browser.



5. Publish the page.

Text I'm typing now Submit
Text I typed earlier

You should see a simple form on your page. When you type text in the field and click **Submit**, the text is added to the displayed responses.

Conclusion

Now that you know how to create a simple form, grab submissions, and store them in content properties, you can create polls, registrations, and other simple apps that let users interact with your pages in Confluence.

Reference

This tutorial was a little more complex than the others and involved a bit more code. In case you'd like to check your work, here's the whole `views/form.hbs` file for reference:

ATLASSIAN

```
{{!< layout}}
<input class="text long-field" type="text" id="response-field" name="response-field"/>
<button id="submit-button" class="aui-button aui-button-primary">Submit</button>
<div id="form-response-list"></div>
<script>
$(function(){
  // Display previous responses
  AP.request({
      url: "/rest/api/content/" + {{pageId}} + "/property",
      success: function(response) {
        var macro = JSON.parse(response);
        console.log("Retrieved form submissions!");
        var searchVal = "appkey_form_";
        for(var i=0;i<macro.results.length;i++){
          if(macro.results[i]["key"].startsWith(searchVal)){
            var response = macro.results[i].value.response_values[0];
            $("#form-response-list").append(response + '<br />')
          }
        }
      },
      error: function(err){
        console.log("Error getting form submissions!");
      }
    });
  // submit button on-click
```

Rate this page: