



appian SPEAR Team

# Dataless Design

*freedom to reuse*

By Dave Hillier Jul 20, 2023

## Concept

Dataless Design is the principle of building out a working building block of functionality, that includes UX and all logic required for it to be a fully functioning component, BUT does not have direct awareness or commitment in advance to what the data model is like.

It's a BYOData approach, but the component handles all the work for both querying and writing data back to the designers record for them even though it doesn't know where the data resides. This is a great story for why Appian's Data Fabric can be so powerful.

The idea is to circumvent the challenges that come with reuse when you want to use a building block in two totally discreet applications in the same environment that should not share data.

More simply perhaps it's a "Custom Record Backed Component".

## What qualifies a Design Library Component as “Dataless Design”

- No hard-coded data model assumptions or record references, so free from the precedence tree with your app. (all links to data are via field reference using rule inputs)
- Operate like components with SaveInto and Value inputs

### Two Core Subtypes:

- **True Dataless:** lets you provide the query via the data rule input, and you specify the field names in other rule inputs so it knows how to access your data.
- **Record Backed Dataless:** these are like the above, but you provide a record type and record field references as inputs, the component queries the data and even writes the

data back for you in some cases. These are easier to use and are often a wrap layer on top of true dataless, but require you to use Sync'd records for full functionality.

Additionally Dataless Components should:

- Be NULL safe, can be easily dragged from the design library on to prototype UX layouts fast.
- Dataless components have common naming conventions and all take UX customisation via a standardized Theme rule.

Watch the Bite Size Video Series

<https://www.youtube.com/playlist?list=PLbOe6ai2qLBpsenhWhjVjrdceLTQ1LdZV>

Can You use the blocks shown in the videos?

Yes, get the latest assets here: <appMarket Listing>

## How does it work?

This design principle exploits the fact that Appian Record queries can be configured using references to record type and record field types that can be passed in through rule inputs.

In this example a Dataless Comments building block understands the 4 fields it cares about, but doesn't know which record they are coming from or even what field the values will be in until its run.

## Querying the Data

```
a!queryRecordType(  
  recordType: ri!recordType,  
  fields: {  
    ri!field_author,  
    ri!field_caseId,  
    ri!field_dateTime,  
    ri!field_comment,  
  },  
  filters: a!queryFilter(  
    field: ri!field_caseId,  
    operator: "=",  
    value: ri!caseId  
  ),  
  pagingInfo: a!pagingInfo(  
    startIndex: 1,  
    batchSize: 5000,  
    sort: a!sortInfo(  
      field: ri!field_dateTime,  
      ascending: false  
    )  
  )  
)
```

## Using the Query Results

Assuming the above query was saved into local!caseNotes, you can then iterate over the returned record rows and use index or square brackets to select the field. When we are building this we do not know the record type or record field names, so just like the query we can use the rule input for the field to index into our query results.

```
a!forEach(  
  items: local!caseNotes,  
  expression: fv!item[ri!field_author]  
)
```

## Additional Filtering

If your dataless component is doing the query work for the user as above, it can be helpful to give them extra control over filtering. This can be done by letting the user pass in a query logical expression. I suggest the field naming convention:

**additionalQueryLogicalExpressions (Anytype)**

## To Query or Not To Query: *Data-backed vs Record-backed*

A small alternative to this pattern is also possible and equally as compelling. In some cases perhaps it's desirable for the component NOT to query and write on behalf of the user. Rather than being a *RECORD BACKED* it is more just *DATA BACKED*. Allowing an additional DATA rule input for the user to provide the query result or rule or any other data source.

In this scenario, the FIELD definition inputs must be ANYTYPE rather than RECORD FIELD, because they can be simple STRING field indexes OR RECORD FIELD. "dueDate" or recordtype!checklist.dueDate.

This gives more flexibility to the user, and does not require Sync'd records as an assumption (required for writes).

The great news is: THE COMPONENT DESIGN IS EXACTLY THE SAME. EXCEPT the data comes from the ri!data rather than a queryRecordType(), and the rule inputs are different.

So I suggest for each Dataless component designed, we publish both a RECORD-BACKED version and a truly "DATALESS" component.

Example of the alternative, where the user provides the Query.

```

rule!Dataless_checklist(
  checklisttitle: "Onboarding Checklist",
  itemnamefield: SC Checklist.name ,
  itemduedatefield: SC Checklist.dueDate ,
  itemstylefield: SC Checklist.style ,
  itemstatusfield: SC Checklist.status.value ,
  itemassigneeffield: SC Checklist.completedByUser.firstAndLastName ,
  itemidfield: SC Checklist.id ,
  data: a!queryRecordType(
    recordType: recordType!SC Checklist ,
    filters: {↔},
    fields: {↔},
    pagingInfo: a!pagingInfo(↔) )
  ).data,
  pagingbatchsize: 5,
  itemlinkfield: SC Checklist.ssrRequestId ,
  linktype: recordType!SC Supplier Company ,
  recordactions: { recordType!SC Checklist.actions.dataStewardReview , recordType!SC Checklist.actions.requestNdaSi
  showtitlebar: a!isNativeMobile()=false(),
  theme: rule!Dataless_ThemeConfig(
  headingColor: "#000",
  showCardShadow:true
  ),
  customstyleconfigs: { 'type!{urn:com:appian:types:DLCL}DL_checklistStyle'(↔)
)

```

## Rule inputs

Let's discuss the rule input data types that make this possible.

There are a few core types that belong to synced records:

- **RecordType** - the recordType! domain pointer to the record that will hold the data, it must be synced to use this build style fully, in that writeRecord() is used to create and update data in many cases. The type must be passed in separately, at this time there is no way to resolve the record type from a record field reference which would be preferable.
- **RecordField** - This is the reference to the record field dot notated into from the recordType
- **RecordAction** - You can also allow users to pass in record actions that can be displayed in your components.
- **RecordRelationship** - This can be used if you have a 1.M concept within the block you are building and expect the designers to give you details of how to access sub-items, maybe like documents attached to comments as an example.
- **RecordIdentifier** - When you are asking for a ID Field Value you might be tempted to use an Number Integer type, but don't forget in some cases, like when working with Salesforce, the IDs can be alphanumeric. Record Identifier allows this flexibility, but I recommend sticking to AnyType rather than this identifier to better cover the two key styles: Records Backed or Data Backed.
- **AnyType** - This is your flexible friend, any of the above could be replaced with AnyType, and in the example below the caseId is shown as an AnyType to give the flexibility

described above in RecordIdentifier. With the exception of RecordIdentifier, it is better to use the specific types to aid your users in providing the correct input values to your components for the Record Backed Components, but field names need to be AnyType for Data Backed Components. If in doubt use AnyTime, but always give good descriptions to your rule inputs.

### Test Inputs

Enter initial input values to test interface

commentsRecordType (RecordType)	Provide the RecordType! of your record where you store comments.	1	<input type="text" value="recordType!BFS Comment"/>
commentsRecordField_caseld (Record Field)	Provide the record field in your comments record that contains the case id. eg recordType!comments[caseid] <a href="#">Less</a>	1	<input type="text" value="BFS Comment.requestId"/>
commentsRecord_caseld (Any Type)	The Case ID for the Comments block to query from the Comments Record to get all ... <a href="#">More</a>	1	<input type="text" value="1"/>
commentsRecordField_author (Record Field)	Provide the FIELD that contains author user type of the comment from your comments record. <a href="#">Less</a>	1	<input type="text" value="BFS Comment.createdBy"/>
commentsRecordField_dateTime (Record Field)	Provide the record FIELD that contains the DateTime stamp for your Comments	1	<input type="text" value="BFS Comment.createdOn"/>
commentsRecordField_messageBody (Record Field)	Provide the Record FIELD from your Comments Record that contains the Comment Message Text <a href="#">Less</a>	1	<input type="text" value="BFS Comment.comment"/>

CANCEL

## Component Thinking

This is a continuation of the rule inputs theme to talk to two important concepts for custom component building that is not common practice with normal interface or subinterface creation. For components you should design to include these two rule inputs:

- **value** - this should often be an **AnyType** when working with record data, but for some components this might need to be an out of the box Appian Type
- **saveInto** - this should be of type **Save** and be an multiple array.

One thing you commonly run into when trying to use a reusable object is that the object owns the data and the events, you can pass in configuration and often get out data, but you can't define that other things happen when a user makes a data change or interfection in the component. This can be avoided and give the user a more natural interaction with the object if you build around the **value-saveInto** data handling model used by out of the box components.

This does make designing the component harder, because it doesn't work properly when testing, this is because you are saving data into the `ri!saveInto`, and that doesn't automatically jump over to the `ri!value`. You need to test in steps, manually moving the `saveInto` to the value OR you wire up your component with test values in another interface just like someone using your component would, this is of course the recommended approach.

Getting your head around this takes a little time, but the benefits to people using the components are high in the long term.

Use case where an add button in a component is appending a selection to a list, using these in `value-SaveIntos` your components looks like this:

Typical way (no flexibility for the user):

```
a!buttonWidget(  
  label: "Add",  
  icon: "plus",  
  
  saveInto: {  
    /*save the data to a rule input the user can use*/  
    a!save(ri!list, append(ri!list, local!selection) ),  
  
    /* Clear selection now its added to the list */  
    a!save(local!selection, {}),  
  },  
  width: "FILL",  
  style: "SECONDARY",  
)
```

Component way `value-SaveIntos` (lots of flexibility) :

```

a!buttonWidget(
  label: "Add",
  icon: "plus",
  value: append(ri!value, local!selection),
  saveInto: {
    /*run the saveInto saves, value above will be passed as save!value */
    ri!saveInto,|

    /* Clear selection now its added to the list */
    a!save(local!selection, {}),
  },
  width: "FILL",
  style: "SECONDARY",
),

```

Now as a user of the component I am controlling where the data saves to and can save it to multiple locations or update other variables when this button is pressed in the component.

A different example of using this concept in the *Dataless Documents* Component is shown below, here the designer using the component is choosing to check if the selected file returned by the component is a PDF or not before choosing to show the document viewer component when a document is selected. In this case the NDA is a PDF and the view is shown. This is simple to do when the component is designed as described above with Value and SaveInto fields.

The image displays a code editor on the left and a user interface on the right. The code editor shows a component definition for a document viewer. A red box highlights the following code block:

```

saveInto: {
  local!selectedDoc,
  a!save(
    local!showDocumentPreview,
    if(document(save!value, "extension") = "pdf", true, false)
  ),
  value: local!selectedDoc
}

```

The user interface on the right shows a document viewer for an NDA contract. The document title is "NDA Contract Radio Chips Limit..." and the size is 8.76KB. The document content is an NDA template with the following text:

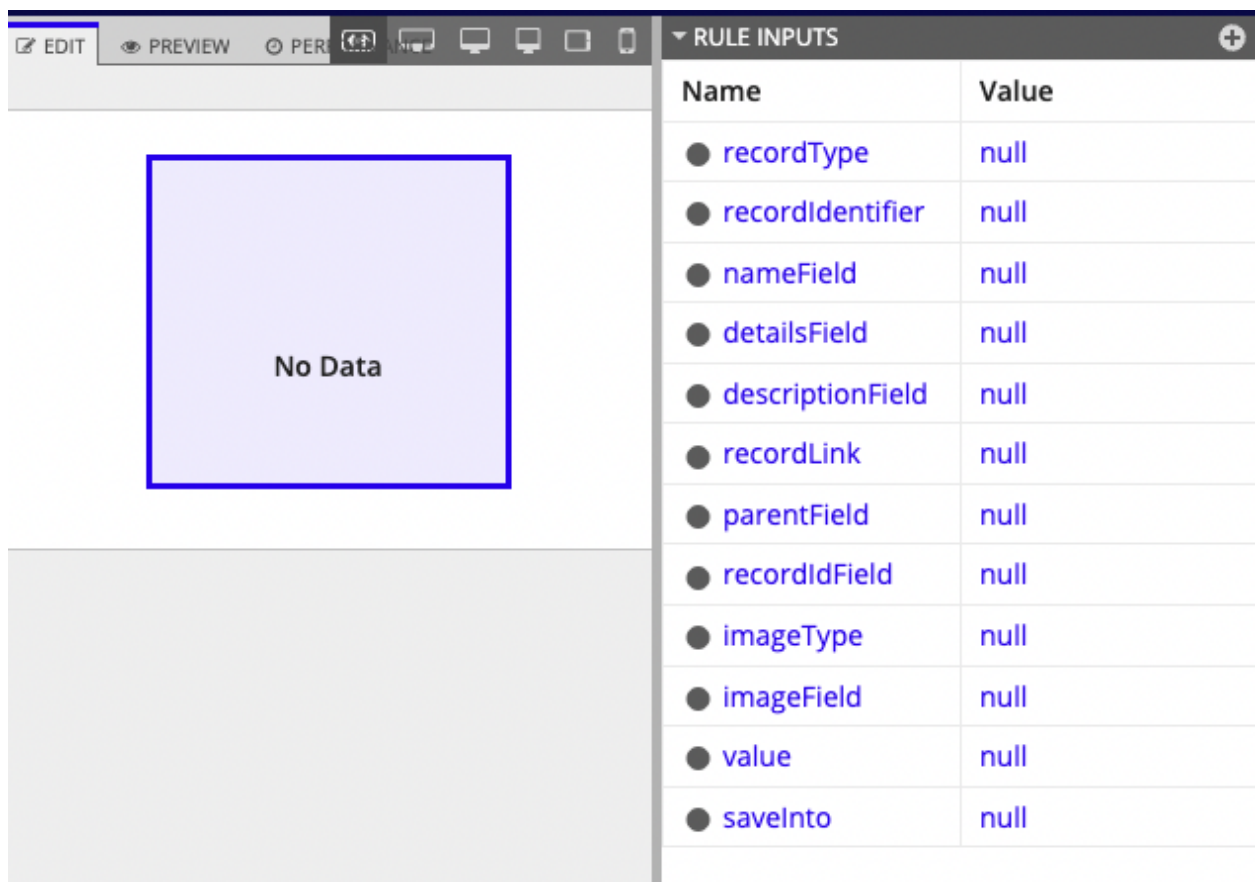
This is the Contract for: NDA Contract Radio Chips Limited  
 Please do not tell anyone what we tell you or vice  
 versa. Please sign on the line.



## Null safe

By ensuring that your component will still render, even if the required data fields have not been provided will help users make use of your component more easily in DESIGN mode. This makes them great for UX Prototyping. ([Watch this Video](#) if you want to see what I mean)

Taking the hierarchy component as an example, now record backed, if you call the rule with no data or incomplete data, it will still render, rather than show a red error.



The screenshot shows a design tool interface with a 'No Data' component and a 'RULE INPUTS' table. The component is a light purple square with a blue border and the text 'No Data' in the center. The table lists various input fields and their values, all of which are 'null'.

Name	Value
● recordType	null
● recordIdentifier	null
● nameField	null
● detailsField	null
● descriptionField	null
● recordLink	null
● parentField	null
● recordIdField	null
● imageType	null
● imageField	null
● value	null
● saveInto	null

## Themes and Pixel Perfect Design

When creating blocks of functionality that are designed to be re-usable across many customers it is easy to fall into normal build practices of statically setting styles in your interfaces but this actually takes us further away from the level of control that a designer or customer might expect.

It is simply not practical to put every style configuration from every component into the rule inputs so that a control can be customized.

Equally it is not great to encourage people to edit the building blocks to style them in their own specific ways.

So I have created the concept of a design Theme for Dataless designed components, a single optional rule input called **theme**, that takes the output from a standardized rule called `Dataless_ThemeConfig()`, that looks like this:

---

**rule!Dataless\_ThemeConfig**(headingColor, headingSize, subheadingColor, subheadingSize, textColor, textSize, iconColor, iconSize, infoColor, buttonStyle, cardShape, showCardBorder, showCardShadow, backgroundColor, contrastBackgroundColor, cardDecorativeBarPosition, cardDecorativeBarColor, padding, marginAbove, marginBelow)

Dataless Components can all take a DL\_ThemeConfig that sets a default color pallet for the components, font colors, icon colors, background colors, and other styling options.

**headingColor** (Text): Determines the text color. Valid values: Any valid hex color or "STANDARD" (default), "ACCENT", "POSITIVE", "NEGATIVE", "SECONDARY".

**headingSize** (Text): Determines the text size. Valid values: "SMALL", "STANDARD" (default), "MEDIUM", "MEDIUM\_PLUS", "LARGE", "LARGE\_PLUS", "EXTRA\_LARGE".

**subheadingColor** (Text): Determines the text color. Valid values: Any valid hex color or "STANDARD" (default), "ACCENT", "POSITIVE", "NEGATIVE", "SECONDARY".

**subheadingSize** (Text): Determines the text size. Valid values: "SMALL", "STANDARD" (default), "MEDIUM", "MEDIUM\_PLUS", "LARGE", "LARGE\_PLUS", "EXTRA\_LARGE".

**textColor** (Text): Determines the text color. Valid values: Any valid hex color or "STANDARD" (default), "ACCENT", "POSITIVE", "NEGATIVE", "SECONDARY".

**textSize** (Text): Determines the text size. Valid values: "SMALL", "STANDARD" (default), "MEDIUM", "MEDIUM\_PLUS", "LARGE", "LARGE\_PLUS", "EXTRA\_LARGE".

**iconColor** (Text): Determines the icon color. Valid values: Any valid hex color or "STANDARD" (default), "ACCENT", "POSITIVE", "NEGATIVE", "SECONDARY".

**iconSize** (Text): Determines the icon size. Valid values: "SMALL", "STANDARD" (default), "MEDIUM", "MEDIUM\_PLUS", "LARGE", "LARGE\_PLUS", "EXTRA\_LARGE".

**infoColor** (Text): Hex Color or "NONE", "STANDARD", "ACCENT", "SUCCESS", "INFO" (default), "ERROR", "ACCENT", "POSITIVE", "WARN", "NEGATIVE".

**buttonStyle** (Text): Determines the intent of the button. Valid values: "NORMAL" (default), "PRIMARY", "SECONDARY", "DESTRUCTIVE", and "LINK".

**cardShape** (Text): Determines the card shape. Valid values: "SQUARED" (default), "SEMI\_ROUNDED", "ROUNDED".

**showCardBorder** (Boolean): Determines whether the layout has an outer border. Default: true.

**showCardShadow** (Boolean): Determines whether the layout has an outer shadow. Default: false.

**backgroundColor** (Text): Determines the card main background color. Valid values: Any valid hex color or "NONE" (default), "STANDARD", "ACCENT", "SUCCESS", "INFO", "WARN", "ERROR", "CHARCOAL\_SCHEME", "NAVY\_SCHEME", "PLUM\_SCHEME". See documentation for more information about how to use color schemes.

**contrastBackgroundColor** (Text): Determines a contrasting card background color. Valid values: Any valid hex color or "NONE" (default), "STANDARD", "ACCENT", "SUCCESS", "INFO", "WARN", "ERROR", "CHARCOAL\_SCHEME", "NAVY\_SCHEME", "PLUM\_SCHEME".

**cardDecorativeBarPosition** (Text): Determines where the decorative bar displays. Valid values: "TOP", "BOTTOM", "START", "END", "NONE" (default).

**cardDecorativeBarColor** (Text): Determines the decorative bar color. Valid values: Any valid hex color or "ACCENT" (default), "POSITIVE", "WARN", "NEGATIVE".

**padding** (Text): Determines the space between the card edges and its contents. Valid values: "NONE", "EVEN\_LESS", "LESS" (default), "STANDARD", "MORE", "EVEN\_MORE".

**marginAbove** (Text): Determines how much space is added above the layout. Valid values: "NONE", "EVEN\_LESS", "LESS", "STANDARD" (default), "MORE", "EVEN\_MORE".

**marginBelow** (Text): Determines how much space is added below the layout. Valid values: "NONE", "EVEN\_LESS", "LESS", "STANDARD" (default), "MORE", "EVEN\_MORE".

And is used like this:

```

/*styles can be overridden with Theme:*/
theme: rule!Dataless_ThemeConfig(
    headingColor: "ACCENT",
    subheadingColor: "#000",
    backgroundColor: "STANDARD",
    textColor: "STANDARD",
    iconColor: "ACCENT",
    cardShape: "SEMI_ROUNDED",
    buttonStyle: "NORMAL"
)

```

This works like a simple style sheet that groups configurations into a few parameters that allow the look and feel of a component to be adjusted and a single Theme be used for all styleable Dataless components in an App so that they end up with a consistent branded feel with a lot less effort.

A single item can be overridden or all of them. The rule will return the defaults, duplicating this rule is the best approach to create a branded theme rule like Dataless\_ThemeConfig\_HSBC(), here you might set all the colors for the brand, but you can still override individual parameters at the component level if you need something slightly different.

## Creating or Updating Record Data.

If you are building Record Backed Components that both read and Write, then this section is for you or the same principles apply if you are creating objects with your component and passing them out of the SaveInto.

Updating a queried record is easy. Creating a new one requires an initialisation step using cast, as you can't directly create a new record type instance when you don't know what the record type is. Normally you might get your comment ready in this record pill style but this **can't** be done:

```

local!newRecordToSave: recordType!BFS Comment (
    BFS Comment.requestId : 1,
    BFS Comment.createdBy : loggedInUser(),
    BFS Comment.createdOn : now()
)

```

To initialize a local variable as a record type that is being passed into your building block via a rule input you need to do the following: (where `ri!commentsRecordType` is of type `RecordType`)

```
local!newRecordToSave: cast(ri!commentsRecordType, a!map()),
```

Note that it is not possible to prime it with values at this stage within the map as we can't know the field names.

Now that you have a variable of the correct type you can write to it in three ways:

- 1) If you have fields that are saving values like in a form, they can write to this record type directly:

```
a!textField(  
    ...  
    value: local!newRecordToSave[ri!commentsRecordField_messageBody],  
    saveInto: local!newRecordToSave[ri!commentsRecordField_messageBody],  
    ...  
)
```

- 2) You can use the square brackets to reference is as normal in a `a!save`

```
a!buttonArrayLayout(  
    buttons: {  
        a!buttonWidget(  
            label: "Save Case Note",  
            saveInto: {  
  
                /*build record*/  
                a!save(local!newRecordToSave[ri!commentsRecordField_caseId], ri!commentsRecord_caseId ),  
                a!save(local!newRecordToSave[ri!commentsRecordField_author], loggedInUser()),  
                a!save(local!newRecordToSave[ri!commentsRecordField_dateTime], now()),  
                a!save(local!newRecordToSave[ri!commentsRecordField_messageBody], local!newNote)  
            },  
  
            /*save record */  
            a!writeRecords(  
                records: local!newRecordToSave,  
                onSuccess: {  
                    a!save(local!newNote, local!newRecordToSave)  
                },  
                onError: a!save(local!newNote, fv!error)  
            )  
        },  
    },  
)
```

- 3) Or you can also use `a!update`.

```

a!buttonArrayLayout(
  buttons: {
    a!buttonWidget(
      label: "Save Case Note",
      saveInto: {
        /*build record*/
        a!save(local!newRecordToSave,
          a!update(local!newRecordToSave, ri!commentsRecordField_caseId, ri!commentsRecord_caseId)
        ),
        a!save(local!newRecordToSave,
          a!update(local!newRecordToSave, ri!commentsRecordField_author, loggedInUser())
        ),
        a!save(local!newRecordToSave,
          a!update(local!newRecordToSave, ri!commentsRecordField_dateTime, now())
        ),
        a!save(local!newRecordToSave,
          a!update(local!newRecordToSave, ri!commentsRecordField_messageBody, local!newNote)
        ),
      }
    )
  }
)

/*save record */
a!writeRecords(
  records: local!newRecordToSave,
  onSuccess: {
    a!save(local!newNote, local!newRecordToSave)
  },
  onError: a!save(local!newNote, fv!error)
)

```

## Versioning

Sometimes breaking changes are needed, you can copy Appians technique for versioning functions.

1. Rename Rule from Dataless\_myComponent to Dataless\_myComponent\_23v1
2. Appian will change this in all objects
3. Duplicate your renamed rule and give it the original name.
4. You can take a list of Dependencies from the renamed component \_23v1 to get a todo list of upgrades and notify owners if you want to encourage upgrades.

## Default Test Cases for Dataless Components that will Drag and Drop

Dataless Components are special by the fact that they are not tied to or dependent on a single record and don't dictate all the fields or the names that should be in that record, but they are as easy to use as an out of the box Record Backed Component.

However, to make configuration easy, there is a button called Load Default Test Values, and if you include a record in here by way of example then it becomes a dependent record that you need to ship with the component and for the record to be valid you will also need to provide a SQL statement for them to run. This turns the using of the building block into a pain, as no one will want to put a test example table into their database to use a building block.

Solutions:

- 1) No example test case. This prevents the dependency but results in a harder to use building block but if it's designed null safe then you can drag it on and click okay to see its null configuration. Note that Design library components have some odd null wrapping behavior seen in expression mode that looks strange in 23.1, this should be resolved in 23.2.
- 2) Example test case that sets some values, but does not provide a record. This is probably the most preferable, it is easy enough to check if the required record type fields are null and if so render something else in the component like help text, or placeholder data; and it doesn't create a dependency on shipping a record with the block.

## Dataless Components Backlog:

- 1) Record Backed Comments **Completed** ▾
- 2) Record Backed Editable Drilldown Grid **Completed** ▾
- 3) Record Backed Hierarchy Diagram **Completed** ▾ (credit [Alexandru Boerescu](#) )
- 4) Record Backed Dual Pick List **Completed** ▾ (credit [Alexandru Boerescu](#) )
- 5) Record Backed Dropdown **Completed** ▾
- 6) Record Backed MultipleDropdown **Completed** ▾
- 7) Record Backed Radio Buttons **Completed** ▾
- 8) Record Backed Check Boxes **Completed** ▾
- 9) Record Backed Hybrid Database/Process Task Checklist **Completed** ▾
- 10) Record Backed Documents **Completed** ▾ (credit [Mark Ansink](#) )
- 11) Record Backed Milestone List **Completed** ▾
- 12) Record Backed Message Inbox **Completed** ▾ (credit [Alexandru Boerescu](#) )
- 13) Process Task List **Completed** ▾ (credit [Kelsey Sartorius](#) )

## PALETTE

COMPONENTS PATTERNS DESIGN LIBRARY

🔍 Search design library...

### ▼ DATALESS LIST COMPONENTS

- 📄 AUDIT FIELD HISTORY DISPLAY
- 📄 AUDIT FIELD SUBMIT BUTTON (RECORD)
- 📄 AUDIT FIELD SUBMIT BUTTON LITE (REC...
- 📄 COMMENTS (RECORD)
- 📄 DOCUMENTS LIST (RECORD)
- 📄 DUAL PICK LIST
- 📄 DUAL PICK LIST (RECORD)
- 📄 EDITABLE DRILL-DOWN GRID (RECORD)
- 📄 EMAIL MESSAGE FOLDERS
- 📄 EMAIL MESSAGE INBOX (RECORD)
- 📄 EMAIL MESSAGE LIST
- 📄 EMAIL MESSAGE READING PREVIEW
- 📄 PAGING CONTROL BAR
- 📄 TASK LIST (DATABASE)
- 📄 TASK LIST (HYBRID)
- 📄 TASK LIST (PROCESS)
- 📄 VERTICAL TIMELINE (RECORD)

### ▼ DATALESS SELECTION COMPONENTS

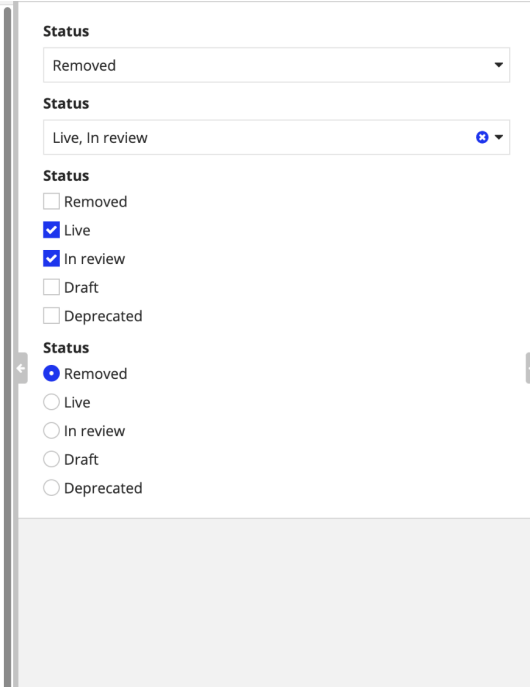
- 📄 CHECKBOX
- 📄 CHECKBOX (RECORD)
- 📄 DROPDOWN
- 📄 DROPDOWN (RECORD)
- 📄 HIERARCHY DIAGRAM (PARENT-CHILD R...
- 📄 HIERARCHY DIAGRAM (RECORD)
- 📄 MULTIPLE DROPDOWN
- 📄 MULTIPLE DROPDOWN (RECORD)
- 📄 RADIO BUTTONS
- 📄 RADIO BUTTONS (RECORD)

### ▼ DATALESS UX SECTIONS

- 📄 'THE COLIN' HEADER
- 📄 INFORMATION BOX
- 📄 MORE LESS TEXT DISPLAY
- 📄 RECORD ACTIONS
- 📄 STEP VERTICAL DISPLAY (STAMPS)
- 📄 THEME CONTAINER
- 📄 VERTICAL TIMELINE

Example of Selection Components now record backed.

```
{
  rule!Dataless_dropdown(
    label: "Status",
    recordType: recordType!NA Status ,
    choiceLabels: NA Status.value ,
    choiceValues: NA Status.value ,
    value: ri!selectSingle,
    saveInto: ri!selectSingle,
    readOnly: false
  ),
  rule!Dataless_multipleDropdown(
    label: "Status",
    recordType: recordType!NA Status ,
    choiceLabels: NA Status.value ,
    choiceValues: NA Status.value ,
    value: ri!selectMultiple,
    saveInto: ri!selectMultiple,
    readOnly: false
  ),
  rule!Dataless_checkbox(
    label: "Status",
    recordType: recordType!NA Status ,
    choiceLabels: NA Status.value ,
    choiceValues: NA Status.value ,
    value: ri!selectMultiple,
    saveInto: ri!selectMultiple,
    readOnly: false
  ),
  rule!Dataless_radioButton(
    label: "Status",
    recordType: recordType!NA Status ,
    choiceLabels: NA Status.value ,
    choiceValues: NA Status.value ,
    value: ri!selectSingle,
    saveInto: ri!selectSingle,
    readOnly: false
  )
}
```



The image displays four different selection components for a 'Status' field, each with its own 'Status' label. The first is a standard dropdown menu with 'Removed' selected. The second is a multi-select dropdown menu with 'Live, In review' selected. The third is a checkbox group with 'Live' and 'In review' checked. The fourth is a radio button group with 'Removed' selected.

- Dataless design gives you the freedom to reuse.
- Let's halve the time again
- Discover delivery economies of scale with reuse
- When you invest in reusability, you naturally invest in better UX.