Headless Adaptive Form Specification

Table of Contents

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 RFC2119 RFC8174 when, and only when, they appear in all capitals, as shown here.

Motivation

There exists two Form Implementations to define Forms in JSON with well defined vocabularies within Adobe : DFL 1.1 and AEM Adaptive Form. DFL 1.1 uses Adobe Sign Expressions, while Adaptive Form specification uses JCR terminology and embeds JavaScript inside its JSON representation.

There has been a parallel work going on to define a specification for a new version of PDF Forms aka PDF Forms.Next specification which deals with PDF currently.

With this specification we are trying to align the current Adaptive Form Specification with the PDF Next Specification and add some extra constructs which can be translated as custom properties or custom events which are supported in PDF Forms.Next specification when rendering the same Form inside of Acrobat.

There are two motivations behind the creation of this specification

- To create a headless representation of Adaptive Forms, so that it can be rendered on multiple channels

- having a single specification for JSON based Forms in the future if all the use cases of the AEM Forms and PDF Forms align.

Introduction

Headless Adaptive Form specifies a mechanism to create a Form or Data Capture Experience using a JSON representation, which allows rendering that experience across multiple channels.

The Specification can be used by implementors to generate a visual appearance for the Data Capture Experience in the choice of their language.

A JavaScript and React based implementation of this specification can be obtained at our internal Git Repository with a live playground to try out the specification. The status of the implementation is provided in [Appendix B: Implementation and Examples]

There is a section in the end which compares the current Form Specification with the PDF Forms Specification : [Appendix A: Comparison with Proposed PDF Forms Specification].

Versioning

The spec is versioned using semantic versioning and follows the semver versioning. The *major.minor* version of the specification SHALL designate the features that the specification provides. The *.patch* version address errors in the document and do not add any new features. The implementation that supports version 1.0 SHOULD be compatible with all the 1.0.* versions

Every document MUST provide the version of the specification which it complies to in addition to its own version.

Definitions

Form Document

A document (or a set of document) that defines the Data Capture Experience to be presented to the user. The Form document uses and conforms to this Form Specification.

The Form Document is a JSON file having a single Form Object. The structure of the document MUST conform to the JSON Schema of the Form specification which is hosted at git

A very simple one field Form can be represented in JSON as

```
{
  "adaptiveform" : "<latest_version>",
  "data" : {},
  "items" : [{
        "name" : "firstName",
        "type" : "string",
        "fieldType" : "text-input",
        "label" : {
          "value" : "Enter your first name"
        }
   }],
   "metadata" : {
        "version" : "1.0.0"
   }
}
```

Form Object

Form Object is the root object in any Form Document. It specifies the version to which the form conforms to. It contains a collection of Fields optionally grouped in Panels to capture data from the end user

Form Field Object

A Field is the unit entity in the Form Document and is represented as a JSON Object. A field can specify one or more of,

- constraints on the data they capture,
- dynamic rules to change their state they have including visibility, enabled, etc.,
- label, help text, etc.,
- handling of any events that can be performed by the end user, like click, focus etc.

A Field that can capture a First Name of a person would look like this

```
{
  "name" : "firstName",
  "label" : {
    "value" : "Enter your First Name"
  },
  "fieldType" : "text-input",
  "description" : "Your given/first name as it appears on your Driving License",
  "type" : "string"
}
```

The terms Field, Form Field and Adaptive Form Field map to the same term in this specification

Form Panel Object

A Form Panel is an entity to group a set of fields logically (to create a complex data type) or visually. The example below shows a Form Panel to capture address of a person

```
{
  "label" : {
    "value" :"Home Address"
  },
  "name" : "homeAddress",
  "fieldType" : "panel",
  "items" : [
    {
      "name" : "houseNumber",
      "type" : "string",
      "fieldType" : "text-input",
      "label" : {"value": "House Number"}
    },
    {
      "name": "street",
      "type" : "string",
      "fieldType" : "text-input",
      "label" : {"value": "Street Address"}
    },
    {
      "name": "city",
      "type" : "string",
      "fieldType" : "text-input",
      "label" : {"value": "City"}
    },
    {
      "name": "state",
      "type" : "string",
      "fieldType" : "text-input",
      "label" : {"value": "State"}
    },
   {
      "name": "zipCode",
      "type" : "string",
      "fieldType" : "text-input",
      "label" : {"value": "Zip Code"},
      "pattern" : "^[0-9]{6}$"
    }
  ]
}
```

Form Object

The root object in any Form Document is a Form Object. The object contains the properties mentioned below. Fields marked as * are required

Property Name	Value
adaptiveform*	The key identifies the version of the Form Specification the document complies to.
metadata	Form Metadata object
data	Data Model of the Form.
items*	a JSON array containing Fields or Panels.
lang	The language in which the Form was authored as per the BCP 47 tag
title	A user friendly title of the Form
description	A user friendly description of the Form
action	an HTTP/HTTPS endpoint where the data would be submitted when submit event is triggered
events	Events is a JSON Object where keys are the name of eventName and value is an array of JSON Formula Expression (or a single JSON Formula Expression) that determine the action to perform. For simplicity a single action can be specified using a string instead of an array of single string. See Events section for more details

Form Metadata

Property Name	Value
version	the version of the document (which is different from the version of the Form Specification) to be interpreted by the application
grammar	the grammar along with its version which the expressions in the Document support. The value MUST be `json-formula-1.0.0`.
formattingLanguage	The grammar to use for formatting the value in different contexts. Current value is `xfa-picture-clause`

Form Label

To make Form Fields and Panels accessible, every Field and Panel SHALL have a label. The implementations MUST ensure that labels and Fields are associated with each other

- In Android they should be associated either using the `android: hint` or `android:labelFor` attributes
- or `aria-labelled-by` or `for` attribute in HTML

The label has the following properties

Property Name	Description	Type	Defaults	ReadOnly
richText	Whether the Label is rich text or not	boolean	false	true
visible	whether the label should be visible to author or not. In case the labels are hidden, the Field should be able to specify a mechanism to provide the cue to the user filling the form. • For HTML, the spec RECOMMENDS using the `aria-labelledBy` attribute on the input elements. • For android, the spec RECOMMENDS using the `android:hint` property on the input element.	boolean	true	false
value*	The content of the label. If the label type is rich text, the contents will be considered as Rich Text	string		false

```
{
    "name" : "firstName",
    "fieldType" : "text-input",
    "label" : {
      "value" : "First Name"
    },
    "rules" : {
      "label" : "{value : if($required, 'First Name *', 'First Name')}"
    }
}
```

File Attachment Field

File attachment field enables capturing file inputs from user. There are two ways to use a file

attachment field

- By declaring a `string` type along a `data-url` or `binary` format

```
{
  "fieldType" : "file-input",
  "type" : "string",
  "format" : "binary"
}
```

- or by specifying type as `file`

```
{
  "fieldType" : "file-input",
  "type" : "file"
}
```

Multiple files selectors are supported by defining a `string[]` or `file[]`

Representation of files

Data URL

Data URLs, URLs prefixed with the data: scheme, allow content creators to embed small files inline in documents. While using `string` type along with `data-url` or `binary` format to use file attachment, its value would be stored as per the following regex pattern, `/^data:([a-z]+\/[a-z0-9-+.]+)?;(?:name=(.);)?base64,(.)$/`

Example of data url, `data:text/plain;name=file1.txt;base64,SGVsbG8sIFdvcmxkIQ==`

Storing large dataURIs into form state might slow rendering, hence we need an optimized way to store files in form state. This implementation of optimization can change as per the channel (for example) in web this could be the native file object

File Object

Value of file attachment field is always stored as a file object containing the following properties,

- `name`: name of the file stored as `string`
- `mediaType`: media type of the file as per the IANA media type stored as `string`
- `size` : size of the file in bytes
- `data` : non-serialized file data, this could be native file object or a URI referring the data.

The contents of file are only serialized during submit (for example) during submission file contents are sent as multipart/form-data over HTTP / HTTPS

Form Field Object

A Form Field is the entity with which the user interacts. It is represented as a JSON Object in the Form Document. The Form Field supports the following properties

Form Field Properties

Property Name	Description	Type	ReadOnly
name	Name of the Field. The name is used to map the value of the field to the data model using default binding rules. If name is empty string and parent panel's data type is not array, the value is not submitted or exported out of the Form. (This use case generally happens for buttons or plain-text fields) If the parent panel's data type is array, the name property is ignored	string	true

Property Name	Description	Type	ReadOnly
fieldType	Type of widget to show to the user for capturing the data. It can be either be one of the types from the list below or a custom type defined by the user. • text-input • number-input • date-input • file-input • drop-down • radio-group • checkbox-group • plain-text • checkbox • button • multiline-input • panel **Default** : is calculated using the [Default Field Types] Algorithm which is also used when transforming a JSON Schema document to an Form Document	string	true
:type	Custom widget type for a particular field In case runtime doesn't support that custom type, it needs to fallback to the fieldType property	string	true
label	Label to be used for the field	Form Label	false
description	Extra description to be shown to the user to aid in form filling experience. It can be rich text. Can be used as help text for a field or a top level description for a Panel	string	false
dataRef	To map the field's value to a property in the data model.	string	true

Property Name	Description	Type	ReadOnly
visible	whether the field should be visible to the user or not **Default**: true	boolean	false
enabled	whether the field is enabled and takes part in rules, events etc. A disabled field can have calculations and custom events can be dispatched to it **Default**: true	boolean	false
default	The value of the field when no value is provided by the end user or data model. The type of this property should match the value of the `type` property defined in the Field. If not, then a type coercion will be tried and if that fails, the value will be set to null.	oneOf: • string • boolean • number • string[] • boolean[] • number[]	true
emptyValue	The value when user has not entered any value in the field. Determines what value should be saved when user has not entered any value in the field. Can be one of • "null" • "undefined" • "" (empty string) (only valid for dataType = string) **Default**: "undefined"	enum: ["null", "undefined", ""]	true
readOnly	whether the field should be readOnly to end user or not **Default**: false	boolean	false

Property Name	Description	Type	ReadOnly
displayFormat	The format in which the value will be displayed to the user on screen in the field. For example when using a currency field, the currency sign should be shown to the user.	string	true
editFormat	The format in which the value will be edited by the user. For instance users in Germany would want to interchange decimal (.) and comma (,) when entering numerical values.	string	true
dataFormat	The format in which the value will be exported or submitted.	string	true
placeholder	The placeholder to show on the widget.	string	true
screenReaderText	a string to indicate the text to be read by screen readers	string	false
rules	Rules that modify the property of the object dynamically. An author can dynamically change any property that is defined in the spec as modifiable, i.e. readOnly is false. The rules are evaluated whenever the dependency changes. Apart from properties defined on the field, rules can have an extra key `value` which auto computes the value of the field The Rules section defines how rules will be processed in detail	object	true
events	Events is a JSON Object where keys are the name of eventName and value is an array of JSON Formula Expression (or a single JSON Formula Expression) that determine the action to perform. For simplicity a single action can be specified using a string instead of an array of single string. See Events section for more details	object	true

Property Name	Description	Type	ReadOnly
enum	A list of options to put restrictions on the possible values of the field The type of values in the enum array must match the value of the `type` property defined in the field. In case the `type` property is not defined, then the type of elements in the enum becomes the value of the `type` property. If the value of the `type` property doesn't match with the type of values in the `enum` array, then a type coercion will be made to match the `type` property. If the coercion is not possible, then the value will be set to null	array	false
enumNames	A user friendly text to display for the possible options to be shown to the end user. The length of enum and enumNames array must match. In case the length of enum is greater, then those will be used as display text for the user. If the length of enumNames is greater, those will be discarded.	array	false

Property Name	Description	Type	ReadOnly
constraintMessages	An object containing the custom error messages to be shown to the end user on different constraint validation. The object MAY contain any of the following keys, • type • required • minimum • maximum • minLength • maxLength • step • format • pattern • minItems • maxItems • uniqueItems • enforceEnum • validationExpression	object	false

Form Field Constraints

Since Form Field capture data from the users, they can also specify the constraints on that data. When creating a Form Document using Schema, the constraints can be derived from the schema itself and translated into Field Constraints.

The vocabulary of the constraints is derived from JSON Schema with subtle differences which are mentioned in the table below

Constraint Name	Description	Type	Defaults	ReadOnly
type*	The Data Types defined in this specification	string	string	true
required	Indicates whether the value is required or not. It means that the value SHALL be non empty string	boolean	false	false

Constraint Name	Description	Type	Defaults	ReadOnly
pattern	As specified in the JSON Schema specification, the regex against which the value of the field should be tested with. The constraint is applicable only for fields with type string	string		true
format	formats as specified in JSON Schema specification. The constraint is applicable only for fields with type string	string		true
maxLength	Maximum Length (inclusive) of the data. The constraint is applicable only for field with type string	number		false
minLength	Minimum Length (inclusive) of the data. The constraint is applicable only for field with type string	number		false
enforceEnum	Whether a user can enter a value that is not present in the enum array. If set to true, a user will not be able to enter any other value that is not in the list of enum. That generally means that enum is used a aid for users to enter the value but is not a validation constraint. The constraint is applicable only if the enum property is defined on the Field	boolean	false	false
minimum	Minimum value (inclusive) that a user can enter in the Field. The constraint is applicable for Fields that have type number or type string and format date (i.e. Date Fields)	Date serialized in ISO 8601 format or a number		true
maximum	Maximum value (inclusive) that a user can enter in the Field. The constraint is applicable for Fields that have type number or type string and format date (i.e. Date Fields)	Date serialized in ISO 8601 format or a number		true
exclusiveMinimum	Applicable for Date and Number Types. Minimum value (exclusive) , either date or number, that can be entered by the user	Date serialized in ISO 8601 format or a number		true

Constraint Name	Description	Type	Defaults	ReadOnly
exclusiveMaximum	Applicable for Date and Number Types. Maximum value (exclusive) , either date or number, that can be entered by the user	Date serialized in ISO 8601 format or a number		true
step	The step attribute is a number that specifies the granularity that the value must adhere to. The step sets the stepping interval when changing the value of a Field. If not explicitly included, step defaults to 1 for number, and 1 day) for the date input types. For example, if we have `{"type" : "number", "min" : 10, "step": 2}` that means any even integer, 10 or great, is valid	number		
uniqueItems	For Fields or Panel mapped to array type of properties	boolean	false	false
minItems	For Fields or Panel mapped to array type of properties	number	0	false
maxItems	For Fields or Panel mapped to array type of properties	number	unlimited (represented as -1)	false
validationExpression	An expression returning boolean value indicating whether the value in the field is valid or not	expression		true
maxFileSize	Maximum file size (in IEC specification) that a field can accept. The constraint is applicable for file attachment field	string	2MB	false
accept	List of standard IANA media types which field can accept. The constraint is applicable for file attachment field	array	[audio/*, video/*, image/*, text/*, application/pdf]	false

Form Field Runtime Properties

Form Field would define some runtime properties that can be accessed in the rules but are not serialized

Property	Value
value	The current value of the Field. The property is serialized in the Data Model
valid	The current validation state of the Field. Whenever the value of the field is changed, and any of the constraint fails, the value of `valid` property is set to false otherwise it remains `true`. The value SHALL be undefined untill the value is changed either by the end user or by means of importData.
errorMessage	The current errorMessage being shown on the Field. Depending upon the constraint that fails, the message specified for that constraint using constraintMessages object is set as the errorMessage
index	The index of the Field within its parent. For the items inside transparent panels, the index value is computed after moving the items to its parent.
parent	The Parent Panel of the Field/Panel. For the items inside the transparent panels, the ancestor of the transparent panel that is not transparent is returned.
panel	The actual parent Panel of the Field/Panel, whether it is transparent or not
indexInPanel	The index within its panel.

Form Panel Object

A Form Panel groups a set of fields logically (to create a complex data type) or visually. It can have a Data Model, so as to capture a hierarchical structure of data and when it is just used for visual grouping, it doesn't modify the structure of data generated from the Form.

Form Panel Properties

The Panel supports the following Field Properties

- label
- fieldType where value is a **panel**
- description
- dataRef
- visible

- screenReaderText
- rules
- events
- index
- parent
- name
- panel
- indexInPanel

Property Name	Description	Type	Defaults	ReadOnly
items*	a JSON array containing Fields or Panels.	Array	[]	depends on the type property. if type is object, then items is fixed but if type is array, then fields in the items array can be added as defined in Repeatable Structures in Form
itemsOrder	The property is an array indicate the order of items to be rendered in the items object.	array		true

Form Panel Constraints

- type
  - The type property in panel accepts these two values : object or array. The type property defines the shape of the data model it is bound to or generates.
- uniqueItems
- minItems
- maxItems

Visual and Logical Grouping of Fields

There are two use cases of a Panel : to group the Fields or Panels logically or visually.

Logical Grouping

Logical Grouping is done to create hierarchical structure of data. For instance capturing a person's information as per the following hierarchy :

```
{
   "basicDetails": { "name" :  "", "age" :  10 },
   "educationQualifications" : {"degree" : "", "year" :  1987, "marks" :  80},
   "addresses" : [{"city" :  "", "state" : ""}, {"city" :  "", "state" :  ""}]
}
```

The type property in the Panel defines the data model of the Panel. If the `type` property is an object, its data model is object where the keys are the names of fields/panels inside it and the values/data of those fields/panels are the values of the keys in the data model.

If the `type` property is an array then the values of the fields/panel inside it are inserted in an array as per the index of the element in the array and that array becomes the data model of the Panel

A form that creates the data as per the above data hierarchy (using the Default Data Binding Rules) has the following structure

```
{
  "items": [
    {
      "name": "basicDetails",
      "type": "object",
      "fieldType": "panel",
      "label": {
        "value": "Basic Information"
      },
      "items": [
        {
          "name": "name",
          "label": {
            "value" : "Name"
          },
          "type": "string",
          "fieldType": "text-input"
        },
        {
          "name": "age",
          "type": "number",
          "label": {
            "value" : "Age"
          },
```

```
        "fieldType": "text-input"
      }
    ]
  },
  {
    "name": "educationQualifications",
    "type": "object",
    "fieldType": "panel",
    "label": {
      "value": "Education Details"
    },
    "items": [
      {
        "name": "degree",
        "type": "string",
        "label": {
          "value": "Degree"
        },
        "fieldType": "text-input"
      },
      {
        "name": "year",
        "type": "number",
        "label" : {
          "value": "Year"
        },
        "fieldType": "text-input"
      },
      {
        "name": "marks",
        "type": "number",
        "label": {
          "value": "Marks"
        },
        "fieldType": "text-input"
      }
    ]
  },
  {
    "name": "addresses",
    "type": "array",
    "fieldType": "panel",
    "label": {
      "value": "Provide all your Address"
    },
    "items": [
      {
        "type": "object",
        "fieldType": "panel",
        "label": {
          "value": "Address Details"
```

```
          },
          "items": [
            {
              "name": "city",
              "type": "string",
              "label": {
                "value": "City"
              },
              "fieldType": "text-input"
            },
            {
              "name": "state",
              "type": "string",
              "label": {
                "value": "State"
              },
              "fieldType": "text-input"
            }
          ]
        }
      ],
      "minItems": 1,
      "maxItems": 5
    }
  ]
}
```

If the hierarchy was not to be created for basicDetails and educationalQualifications, i.e. generating the following structure

```
{
    "name" :  "",
    "age" :  10,
    "degree" : "",
    "year" :  1987,
    "marks" :  80
}
```

the form would be

```
{
  "items": [
    {
      "name": "name",
      "label": {
        "value" : "Name"
      },
      "type": "string",
      "fieldType": "text-input"
    },
    {
      "name": "age",
      "type": "number",
      "label": {
        "value" : "Age"
      },
      "fieldType": "text-input"
    },
    {
      "name": "degree",
      "type": "string",
      "label": {
        "value": "Degree"
      },
      "fieldType": "text-input"
    },
    {
      "name": "year",
      "type": "number",
      "label" : {
        "value": "Year"
      },
      "fieldType": "text-input"
    },
    {
      "name": "marks",
      "type": "number",
      "label": {
        "value": "Marks"
      },
      "fieldType": "text-input"
    }
  ]
}
```

Visual Grouping

If the Panel doesn't specify a type property, then it cannot have any data and is transparent in the Form Runtime Model. Though it can still define rules but other Fields in the form cannot access it.

The fields inside such a Panel would be accessible as if they are directly under its ancestor which is not transparent.

A Panel which doesn't have a type property is termed as Visual Panel in other parts of this document.

A common use case of visual grouping is an author wants to capture the data without hierarchy but they want to capture them in different screens. For instance in the second example above for person's information, they want to capture personal details in one screen the educational details in second screen and so on, the form would look like

```
{
  "items": [
    {
      "name": "Screen1",
      "fieldType": "panel",
      "items": [
        {
          "name": "name",
          "type": "string",
          "fieldType": "text-input"
        },
        {
          "name": "age",
          "type": "number",
          "fieldType": "text-input"
        }
      ]
    },
    {
      "name": "Screen2",
      "fieldType": "panel",
      "items": [
        {
          "name": "degree",
          "type": "string",
          "fieldType": "text-input"
        },
        {
          "name": "year",
          "type": "number",
          "fieldType": "text-input"
        },
        {
          "name": "marks",
          "type": "number",
          "fieldType": "text-input"
        }
      ]
    },
```

```
    {
      "title": "Screen3",
      "fieldType": "panel",
      "items": [
        {
          "name": "addresses",
          "type": "array",
          "fieldType": "panel",
          "items": [
            {
              "type": "object",
              "fieldType": "panel",
              "items": [
                {
                  "name": "city",
                  "type": "string",
                  "fieldType": "text-input"
                },
                {
                  "name": "state",
                  "type": "string",
                  "fieldType": "text-input"
                }
              ]
            }
          ],
          "minItems": 1,
          "maxItems": 5
        }
      ]
    }
  ]
}
```

Repeatable Structures in Form

Another common use case in Forms is to capture the repeatable data ,i.e. multiple addresses, dependent names, etc. To support that use case, the repeating entity SHALL be wrapped in a Panel that SHALL have its `type` property's value as array.

The last `item` of the items array of the Panel SHALL repeat based on the `minItems` and `maxItems` property on the Panel. The Panel SHALL receive addItem and removeItem events to add or remove an instance of the `item`.

For instance a Field capturing multiple Additional Names can be represented as

```
{
  "title": "Additional Names",
  "type": "array",
  "name": "additionalNames",
  "fieldType": "panel",
  "items": [
    {
      "type": "string",
      "title": "Name",
      "fieldType": "text-input"
    }
  ],
  "minItems": 1,
  "maxItems": 10
}
```

Same Named Elements

The items in a Panel need to have unique names. If any two items (either Panel or Field) have the same name within a given Panel,

- they would bind to the same Data Model, leading to same values
- Only the first element (sorting them by their index property) would be accessed in the rules.

Events

Form Specification defines an event model to make the form work consistent across devices. The Form SHALL trigger events in a certain order as defined in this specification. Each event is categorized by its name, payload or a target object which is the Field/Panel or Form object which is the source of that event

In the Form Definition, one can define operations that are to be performed whenever an event is triggered. These operations are termed as Event Handlers.

The event handlers are defined using the events property of the Form or Field object.

The events object is a collection of `{key: value}` pairs, where key is the name of the event as defined below and value is a JSON Formula Expression (or an array of JSON Formula Expression) that are executed whenever that event is triggered.

There will be two types of events, * one which are triggered on all the Fields in depth-first order starting from the top level Form object and others * which are triggered on a specific Field/Panel object.

Event processing is asynchronous. Events should be pushed to an event queue and from there they should be picked in order when the current event processing is over. The dispatchEvent Expression Grammar Function puts the event handlers in the queue and is also asynchronous

Event Object

Each event also exposes an object which can be accessed in the expression as $event. The event object will have the basic properties

Property	value
type	the name of the event
target	the field element on which the event is triggered.
originalTarget	the original element on which the event was triggered. This is needed when the event propogates down
payload	custom payload as defined by the event.

Default Events

load

The event is triggered on the form object when the form load is complete. The event executes on the form only.

initialize

The event is triggered on all the fields when the form initialisation is complete, which includes

- execute any importdata operations made in load event
- data is merged on to the client and all the fields have the value as per the data imported into the form

The event starts from the top level form element and triggers down in depth first order.

change

The change event is triggered on the field whenever the value of the field is changed. It will be triggered on a Panel whenever a child inside that is added (if the child was repeatable). The payload for that would be the entire element added.

prevValue	Previous value of the field
newValue	The new value as entered by the user

focus

The event is dispatched on the field that gets focus. The focus is applicable on panels as well. Whenever the First field in the panel gets focus, the focus event of the parent is triggered and then the focus event of the field is invoked

blur

The event is dispatched on the field that loses focus. This event is applicable for panel as well. Whenever the last field in the panel loses focus, the blur event of the field is triggered and then the blur event of the parent is invoked

click

The event is triggered when user clicks on an element.

invalid

The invalid event is triggered when a Field's value becomes invalid after a change event or whenever its value property changes (as part of dependency tracking). After the value changes, or change event is executed, all the constraints are evaluated and in case a constraint fails, invalid event is fired and may be processed ahead of any other event in the queue. The payload must contain the status of all the failed constraints that are evaluated.

value	Invalid value of the Field
constraints	an array of constraints that failed.
messages	Message defined by the authors for each constraint
conditions	the value of the constraint defined by the author that failed.

valid

The valid event is triggered whenever the field's valid state is changed from invalid to valid.

submit

The submit event is triggered on the Form and submits the form data to configured action. To trigger the submit event, submitForm function needs to be invoked or one can invoke dispatchEvent API.

save

The save event is triggered on the Form and saves the form data to AEM. To trigger the save event, save event is dispatched on Form

reset

The reset event is triggered on the Form and dispatches to all the Fields. The reset event is triggered at the form level and triggers down in DFS order to each field. The default behaviour of the event is to reset the value to default Value or pre-filled value which can be cancelled by the author. Since this event changes the value of the form, all the change events and rules will get triggered.

addItem

The event is triggered on a panel to add a new instance of items inside it. The Form Processor adds a new instance of the field if the maxItems constraint is greater than the count property and executes the expression specified on the addInstance event. If the processor cannot add a new instance, the addInstance event is ignored. Authors are required to dispatch this event on the panel to add a new instance in its item array.

The payload of the addItem is a number indicating the index at which to add the item

removeItem

The event is triggered on a panel to remove an instance of items inside it. The Form Processor removes a new instance of the field if the minItems constraint is greater than the 0 and less than the count property and executes the expression specified on the removeInstance event. If the processor cannot remove an instance, the event is ignored. Authors are required to dispatch this event on the Panel to remove an instance in its item array.

The payload of the removeItem is a number indicating the index at which to remove the item

error

The error event is triggered at the form level whenever there is any error in Form Processor or script. Similar to invalid event it may be processed ahead of all the other events in the queue.

Custom Events

Event dictionary can have custom events that are dispatched using dispatchEvent function dispatchEvent. The event on the Form and Panel will be triggered down

Response Events

Response Events are specific type of Custom Events that are dispatched on the Form after the Submit or Web Service Request Succeeds or Fails. The Payload of the Response Event is defined as

initiator	Initiator of the Submit or Web Service Request
response	the response of the Web Service Request as JSON

Event Handler

The Event Handler(s) for any event can be defined in the JSON using the events property of the Field/Panel object.

The keys in the events object determine the name of the event and value against that key determines the actions to be performed.

The example below demonstrates how to perform submit on a click of a button

```
{
  "name" : "submit_btn",
  "fieldType": "button",
  "title" : {
    "value" : "Submit"
  },
  "events": {
    "click" : "submitForm()"
  }
}
```

Whenever the event is dispatched the expressions defined against that event are evaluated. The return value of that expression is applied to the field. The return value of the expression can be either a

- null, array, number, string or boolean: The `value` property of the field is set to the return value
- JSON Object: The field json is merged with the returned dictionary
- empty Object (`{}`): the field should not be modified.

Multiple Event handlers are evaluated sequentially and the return value of one is applied to the field object before evaluating the next expression

```
{
  "fieldType": "button",
  "label" : {
    "value" : "Submit"
  },
  "events": {
    "click" : ["dispatchEvent('custom:submitStart')", "submitForm()"]
  }
}
```

Global Event Handler

Proposal

Similar to Global Event Handler in HTML an author can define common functionality on events Globally. For instance if they want to show error only on Focus, they can have a global event handler for that event which would be applicable for all items of that panel or form.

Rules

The properties of the fields are dynamic and can change on user actions. The Rules property helps the form author specify how the properties change dynamically. The structure of the rules is pretty straight forward. It is an object with keys specifying the property name and their value is an expression as defined by the Expression Grammar

```
{
  "name": "SpouseName",
  "fieldType" : "text-input",
  "label": {
    "value": "Spouse Name"
  },
  "rules" : {
    "required" : "maritalStatus == 'married'",
    "readOnly" : "someField == 'inactive'"
  }
}
```

The rules are evaluated whenever the dependency changes, for example in the case above, whenever maritalStatus field changes the required rules is evaluated. Rules on any field that is dependent on spouseName will also get evaluated after spouseName gets modified

Form Runtime Model

The Form Runtime Model is a memory representation of the Form stored in a tree structure with Form at the root of the tree. Form Field are the leaves and Form Panels are the internal nodes in the tree.

The expressions written in rules or events are evaluated against this Runtime Model.

There are two special types of nodes in the Runtime Model :

- The Panels that have no type property defined on them.
- The Fields or Panels that have no name property, except when they are present in a Panel that has `type` property as `array`

This Form Specification terms these nodes as transparent nodes, i.e. they do not take part in Expressions but are only required for the View layer to render them on screen. Though they can define rules or events that are dispatched by the View layer (or when event propagates down via the dispatchEvent API), they cannot be accessed by other Fields. The fields inside such a Panel would be accessible as if they are directly under its ancestor which is not transparent.

Accessing an element in the Runtime Model

Since the Form Runtime Model is stored as a Tree Structure with internal nodes are Panels while the leaf nodes are Fields. The items of a Panel are stored as its child nodes in the structure.

Each node in the leaf has a name property (except for Panels). Every node has an Expression by which it can be accessed in the grammar which is defined as the names of all the nodes in its path from the rootNode joined by the period character '.'. The unnamed elements except when they are in an `array` type are ignored from this construction. If the names contain characters not in the character set [a-zA-Z0-9_$], then it has to be put inside "" (double quotes) as required by the JSON Formula Grammar.

The name of the root node is `$form`

If in the path there exists a panel with `type` property as `array` then its items are referred to using the index notation.

```
{
    "fieldType": "panel",
    "type" : "object",
    "name": "accidentData",
    "items" : [{
        "name" : "fullName",
        "fieldType" : "panel",
        "type" : "object",
        "items" : [{
            "fieldType" :  "panel",
            "items" : [{
              "name" : "firstName"
            }]
        }]
    }, {
      "name" : "accidentCoordinates",
      "fieldType" : "panel",
      "type" : "array",
      "items" : [{
        "label" : {
          "value": "latitude"
        },
        "fieldType" : "number-input"
      },
      {
        "label" : {"value": "longitude"},
        "fieldType" : "number-input"
      }]
    },
  {
    "name" : "sp3cial'Field",
    "fieldType" : "number-input"
  }
]
}
```

In the example above (only relevant properties are shown for brevity), the Field firstName can be accessed using its path as `$form.accidentData.fullName.firstName` in the JSON Formula expressions while the latitude and longitude fields will be accesed as `$form.accidentData.accidentCoordinates[0]` and `$form.accidentData.accidentCoordinates[1]`. While the field with special characters need to be accessed as `$form."sp3cial'Field"`

Apart from the absolute path, a Form Field can access the following elements directly by their name

- direct children and the children of their transparent child,

- siblings

Expression Evaluation

Expression Queue

The Implementations must manage an Expression Queue, and expressions should be picked from the top of the queue for processing. Whenever an expression has to be evaluated, it must be placed at the end of the queue.

Dependent Rules

A Field can define rules to dynamically update its property depending on the value of some other Field. The Expression that defines how that property is updated is termed as Depdendent Rule in this section.

Expression Processing

The Expressions in the Expression Queue are processed after a user event is generated by the UI and completes until there are no more expressions in the queue. When a user event, like change or click, is captured : the expression to be evaluated must be put in an Event Queue and then evaluated. If a Field's value changes either using expression or any event, then all its dependent rules are put at the end of the event queue.

Execution Order

The Execution order is maintained by how elements are being put in the Queue. Consider the Form

```
{
    "items" : {
        "A" : {},
        "B" : {
          "rules" : { "value"  : "C + A"}
        },
        "C" : {
          "rules" : { "value"  : "'1' + A"}
        }
    }
}
```

When the Field A changes, a simple queue implementation may lead to the following evaluation order : B, C, B.

To avoid that, when a rule expression needs to be evaluated, it checks whether any of its dependency is in the Queue or not, and if it is then it puts itself back in the Queue. This can be optimized by doing it at the time of putting the expressions in the Queue as well.

Cycling Dependency

The count of an element being put inside the Queue must be managed (except when it was done to avoid unnecessary calculations) from the time the Queue Processing starts till the Queue becomes empty. If the count reaches a threshold, say 10, it should not be put inside the Queue again, until the

Queue becomes empty.

Expression Grammar

The Form Specification uses JSON Formula for its expressions. JSON Formula is an expression language used by the PDF Forms.Next specification. Using the same language in this Form Specification ensures that the expressions in Form aligns and can be used when the forms are rendered in PDF.

JSON Formula is developed as an combination of the JSON query language JMESPath. and the spreadsheet formula specification OpenFormula

JMESPath allows you to perform different operations on JSON data. The language is expressive enough to support most required operations on JSON data without exposing the engine processing the expressions to security attacks.

The OpenFormula functions added to JSON Formula enable additional operations which were not possible with JMESPath.

JSON Formula comprises of the following additional features from JMESPath:

- OpenFormula Functions
- Additional Operators

JMESPath Features

This section provides a very quick overview of the features of JMESPath relevant for Forms.

A detailed tutorial on JMESPath can be found here

For simplicity of understanding, the examples in this section are example JSONs and do not represent valid Forms.

Query

The `.` operator is used to access properties within an input JSON.

JSON:

```
{
    "A" : {
        "B": {
            "C": 10
        }
    }
}
```

Expression: `A.B.C`

Result: `10`

Slicing

Arrays can be sliced using the common syntax `[start:stop:step]`.

JSON:

```
{
    "A" : [0, 1, 2, 3, 4]
}
```

Expression: `A[0:3]`

Result: `[0, 1, 2]`

Expression: `A[::-1]`

Result: `[4, 3, 2, 1, 0]`

List/Slice Projections

Projections are used to collect nested data by iterating sub-elements.

JSON:

```
{
    "A" : [
        { "ID": 1 },
        { "ID": 2 },
        { "ID": 3 },
        { "ID": 4 }
    ]
}
```

expression: `A[*].ID`

Result: `[1, 2, 3, 4]`

Expression: `A[0:2].ID`

Result: `[1, 2]`

Object Projections

Object projections are similar to list projections but they work on objects.

JSON:

```
{
    "A" : {
        "B" : { "ID": 1 },
        "C" : { "ID": 2 },
        "D" : { "ID": 3 },
        "E" : { "ID": 4 }
    }
}
```

Expression: `A.*.id`

Result: `[1, 2, 3, 4]`

Nested Projections

Both list/slice projections and object projections can be nested to multiple levels.

Flatten Projection

The `[]` syntax can be used to flatten the results provided by nested projections.

JSON:

```
{
    "A": [
        {
            "B": [
                {"ID": "1"},
                {"ID": "2"}
            ]
        },
        {
            "B": [
                {"ID": "3"},
                {"ID": "4"}
            ]
        }
    ]
}
```

Expression: `A[*].B[*].ID`

Result:

```
[
    ["1", "2"],
    ["3", "4"]
]
```

Expression: `A[*].B[].ID`

Result: `["1", "2", "3", "4"]`

Filter Projection

Use the `?` operator inside projections to filter them based on a provided condition.

JSON:

```
{
    "A": [
        {
            "B": [
                {"ID": 1},
                {"ID": 2}
            ]
        },
        {
            "B": [
                {"ID": 3},
                {"ID": 4}
            ]
        }
    ]
}
```

Expression: `A[*].B[?ID==\`3\`].ID`

Result: `[[], [3]]`

Expression: `A[*].B[?ID==\`3\`][].ID`

Result: `[3]`

Pipes

The `|` operator can be used to stop execution of a projection and send the output as in input to another sub-expression.

JSON:

```
{
    "A": [
        {
            "B": [
                [1, 2, 3],
                [4, 5, 6]
            ]
        },
        {
            "B": [
                [7, 8, 9]
            ]
        }
    ]
}
```

The following Expression: `A[*].B[][0]`

Result: `[1, 4, 7]`

Note, how the projection continues in the above expression. Pipes can be used to stop the projection.

Expression: `A[*].B[] | [0]`

Result: `[1, 2, 3]`

Functions

The following functions are supported by JMESPath. Each function call enforces the types of the parameters passed to the function.

- abs
- avg
- contains
- ceil
- ends_with
- floor
- join
- keys
- length
- map
- max
- max_by

- merge
- min
- min_by
- not_null
- reverse
- sort
- sort_by
- starts_with
- sum
- to_array
- to_string
- to_number
- type
- values

Others

In addition to the above, JMESPath supports other features like multi-selects, expression references, etc. Refer to JMESPath documentation for more details.

Additional Features over JMESPath

This section describes the additional functionalities in JSON Formula which are not part of JMESPath.

Operators

The following operators are supported:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `^` power
- `<` less than
- `>` greater than
- ⇐ less than or equal to
- `>=` greater than or equal to
- `!=` not equals

- `<>` not equals
- `==` equals
- `%` remainder
- `!` unary not
- `&` string concatenation

Some examples are provided below:

JSON:

```
{
    "A": 9,
    "B": 2,
    "C": "first",
    "D": " second"
}
```

Expressions:

```
exp> A + B
res> 11

exp> A - B
res> 7

exp> A * B
res> 18

exp> A / B
res> 4

exp> A ^ B
res> 81

exp> A % B
res> 1

exp> A < B
res> false

exp> A > B
res> true

exp> A == B
res> false

exp> !(A == B)
res> true

exp> C & D
res> "first second"
```

Functions Extensions

In addition to the functions specified in JMESPath, the following extra functions are added to JSON Formula

Function	Signature	Description
casefold	`string casefold(string $input)`	Return a lower-case string using locale specific mappings.
toMap	`object toMap(string $key, any $value)`	Create an object with the given key and value.
and	`boolean and(…any)`	Returns the logical AND result of all parameters.
or	`boolean or(…any)`	Returns the logical OR result of all parameters.

not	`boolean not(any $input)`	The logical NOT applied to the input parameter.
true	`boolean true()`	Returns boolean `true`.
false	`boolean false()`	Returns boolean `false`.
if	`any if(boolean $condition, any $true, any $false)`	Return one of two values, depending on a condition.
substitute	`string substitute(string $text, string $old, string $new)`	Returns input `text`, with text `old` replaced by text `new` (when searching from the left).
value	`any value(object|array input, string|integer index)`	Perform an indexed lookup on a map or array.
lower	`string lower(string $value)`	Convert string to lowercase.
upper	`string upper(string $upper)`	Convert string to uppercase.
exp	`number exp(number $power)`	Return the result of constant `e` raised to `power`.
power	`number power(number $base, number $power)`	Return the result of `base` raise to `power`.
find	`number find(string $query, string $text [, number $start = 0])`	Return the starting position of `query` in `text` starting from position `start`. Returns `null` if no matches are found.
left	`string left(string $text [, number $length = 1])`	Returns `length` number of characters of `text` from the left.
right	`string right(string $text [, number $length = 1])`	Returns `length` number of characters of `text` from the right.
mid	`string mid(string $text, $number start, number $length)`	Returns `length` number of characters of `text` starting from `start`.
proper	`string proper(string $text)`	Returns `text` with first character of every word converted to uppercase and the rest converted to lowercase.
rept	`string rept(string $text, number $count)`	Returns `text` repeated `count` number of times.
replace	`string replace(string $text, number $start, number $length, string $new)`	Returns `text` replacing `length` characters at starting at postion `start` with `new`.
round	`number round(number $num, number $precision)`	Rounds `num` to precision specified by `precision`.

sqrt	`number sqrt(number $num)`	Returns the square root of `num`.
stdevp	`number stdevp(array $values)`	Returns the standard deviation of `values`.
stdev	`number stdev(array $values)`	Returns the sample standard deviation of `values`.
trim	`string trim(string $text)`	Remove leading and trailing spaces, and replace all internal multiple spaces with a single space.
trunc	`number trunc(number $num, number $digits)`	Truncates `num` to the specified `digits`.
charCode	`string charCode(number $value)`	Returns the character represented by numeric `value`.
codePoint	`number codePoint(string $text)`	Returns the code point of the first character of `text`.
date	`number date(number $year, number $month, number $day)`	Returns the date representation.
day	`number day(number $datetime)`	Extracts the day from `datetime`.
month	`number month(number $datetime)`	Extracts the month from `datetime`.
year	`number year(number $date)`	Extracts the year from `datetime`.
time	`number time(number $hour, number $minute, number $second)`	Returns the time representation.
hour	`number hour(number $datetime)`	Extracts the hours from `datetime` .
minute	`number minute(number $datetime)`	Extracts the minutes from `datetime`.
second	`number second(number $datetime)`	Extracts the seconds from `datetime`.
now	`number now()`	Returns the current local date time representation.
today	`number today()`	Returns the current local date representation.
weekday	`number weekday(number $datetime, number $type)`	Returns the weekday corresponding to `datetime`.

The below functions are not part of JSON Formula default implementation but have to be added as an extension to the available functions by the implementation.

Function	Signature	Description

validate	`validate([any $element = undefined])`	Run validation on `$element`. Run's validation on the entire form if element is not provided.
getData	`any getData()`	Return the Form's data
submitForm	`submitForm(string $success, string $failure [, string $submit_as = 'json', any $data = null])`	Trigger a Form submission. The `$success` event will be triggered on success. The `$failure` event will be triggered on failure. Users can optionally specify what to submit via the `$data` parameter.
request	`request(string $uri, string $httpVerb, object $payload, string $success, string $failure`	Make a HTTP request of type `$httpVerb` to `$uri` with data `$payload`. The `$success` event will be triggered if the request succeeds. The `$failure` event will be triggered in case of failure.
dispatchEvent	`dispatchEvent(any $element, string $eventName [, any $payload])`	Dispatches an event of type `$eventName` on element determined by `$element`. The optional `$payload` will be made available to the expressions handling the event.

Differences with JMESPath

Because JSON Formula is an extension to the functionality of JMESPath, there are cases where due to the functionality extension some expressions behave differently in JMESPath and JSON Formula. This section will capture any such identified differences. * The integer `0` is treated as a `false` value where a boolean is required. * Function arguments are coerced to compatible types. This leads to certain invalid expressions of JMESPath becoming valid JSON-Formula expressions. * Some invalid expressions return `null` instead of a syntax error.

References

An antlr grammar is hosted on Git

Form Data Validation

Form can be created from a schema which can be either JSON Schema, XSD or any other schema definition that may come in the future. In addition to that a Form SHALL also export a JSON Schema that define the **structure** of the data that it exports. But that schema **only** SHOULD not be used to validate the Form Data for a bunch of reasons.

If the implementation chooses to only implement schema validations it SHALL specify the same and not support the `validationExpression` construct of this Form specification.

A note about JSON Schema

There are many technologies that use JSON schema terminology to represent Forms. JSON Schema introduces itself as a JSON media type for defining the structure of JSON data. It is in fact a great choice to define the structure of a data and validate it.

It provides a lot of functionality, but there are two problems when using JSON Schema vocabulary as a Form here

- It is not a good fit for defining UI choices to take when capturing that data.
- The entire data instance is validated by the JSON Schema. It cannot validate, in isolation, a single value in the document. This limitation requires generating entire data for throwing validations errors, leading to performance issues for Forms having 100s of Fields.
- It cannot perform validations based on the value of some other key, for instance an Insurance Form where total share of all the nominees should be less than 100. There was a proposal for the same in v5 draft, but was never supported.

There have been open source contributions which extend JSON Schema with a vocabulary to do that, notably

- React JSON Schema Form
- JSON Forms
- Vuetify-JSONSchema Form

JSON Forms and React JSON Schema Form define a separate UI Schema whose elements point to the JSON Schema. Though the approach is good, but is hard to define a vocabulary to validate such a document with a schema validator.

Vuetify Forms provide a much richer functionality but does not have a proper grammar since it defines certain form patterns that can be achieved by the JSON Schema implementation and one needs to understand those patterns to implement a rendition in a different channel.

All of them are creating a JSON Schema to create a Form which we were not interested in, but our goal is to create a JSON to represent a User Experience Schema.

Just to complete this section, here are two examples and how they can be represented in the various technologies.

Choosing Different properties depending on a previous selection

JSON Schema Representation for dynamic properties

```
{
    "oneOf": [
        {
            "title": "US",
            "properties": {
                "country": {
                    "type": "string",
                    "const": "US"
                },
                "state": {
                    "type": "string",
                    "title": "State"
                },
                "zipCode": {
                    "type": "string",
                    "title": "Zip Code"
                }
            }
        },
        {
            "title": "Canada",
            "properties": {
                "country": {
                    "type": "string",
                    "const": "Canada"
                },
                "province": {
                    "type": "string",
                    "title": "Province"
                },
                "postalCode": {
                    "type": "string",
                    "title": "Postal Code"
                }
            }
        }
    ]
}
```

Form Document for the Same Example above

```
{
    "items": [
      {
        "name": "country",
        "label" : {"value": "Country"},
        "type" :  "string",
        "fieldType" : "text-input"
```

```
      },
      {
        "name" : "state",
        "label" : {"value":"State"},
        "fieldType" : "text-input",
        "type" :  "string",
        "rules" :  {
            "required" : "$form.country.value == 'USA'",
            "visible" : "$field.required"
        }
      },
      {
        "name" : "province",
        "label" : {"value": "Province"},
        "type" :  "string",
        "fieldType" : "text-input",
        "rules" :  {
            "required" : "$form.country.value == 'CANADA'",
            "visible" : "$field.required"
        }
      },
      {
        "name" : "zipCode",
        "label": {"value": "Zip Code"},
        "type" :  "string",
        "fieldType" : "text-input",
        "rules" :  {
            "required" : "$form.country.value == 'USA'",
            "visible" : "$field.required"
        }
      },
      {
        "name" : "postalCode",
        "type" :  "string",
        "label": {"value": "Postal Code"},
        "fieldType" : "text-input",
        "rules" : {
            "required" : "$form.country.value == 'CANADA'",
            "visible" : "$field.required"
        }
      }
    ]
}
```

Controlling the number of values in an array depending upon some other value

JSON Schema representation for dependent values

```
{
```

```
    "properties" : {
      "numberOfDependents" : {
        "type" : "number"
      },
      "dependents" : {
        "items" : {
          "type" : "string"
        }
      }
    },
    "dependentSchema" : {
      "numberOfDependents" : {
        "allOf" : [
          {
            "if" : {
                "properties" : {
                  "numberOfDependents": {
                    "const" : 1
                  }
                }
            },
            "then" : {
                "properties" : {
                  "dependents": {
                    "minItems" : 1
                  }
                }
            }
          },
          {
            "if" : {
                "properties" : {
                  "numberOfDependents": {
                    "const" : 2
                  }
                }
            },
            "then" : {
                "properties" : {
                  "dependents": {
                    "minItems" : 2
                  }
                }
            }
          },
        ]
      }
    }
}
```

Form Document for the above use case

```
{
    "items" : [
      {
        "name": "numberOfDependents",
        "type" : "number",
        "fieldType" : "number-input"
      },
      {
        "name": "dependents",
        "fieldType" : "panel",
        "items" : [{
          "type" : "string",
          "fieldType" : "text-input"
        }],
        "rules" : {
          "minItems" : "$form.numberOfDependents.value",
          "maxItems" : "$form.numberOfDependents.value"
        }
      }
    ]
}
```

Data Model

An Adaptive Form allows capturing data from the end user. It SHALL export a schema that can specify the structure of that Data Model. Additionally, the implementations MAY choose to add

Data Types

The Data Types in the Data Model SHALL conform to the following data types defined in the JSON Standard

- string
- number
- boolean
- integer which is a JSON number without the fraction part
- object
- array : only homogenous arrays of the types mentioned above are supported

Default Data Model

The structure/hierarchy of the Data Model by default is same as the structure of the Form Runtime Model, with the transparent nodes removed altogether from the Data Model. When the Data Model

is serialized it is represented as the `data` property in the Adaptive Form Object.

The top level entity in the Data model is mapped to the Form Object which is present as the `data` property in the Form Object.

The Field/Panel in the Form object MAY map to a property in the Form's Data Model. By mapping, the value entered by the user in the field will be set to the value of that property in the Data Model.

The mapping can be defined implicitly using the `name` of the field or explicitly using the `dataRef` property.

The Fields SHALL only map to properties in its Parent data model that have the following data types

- string
- boolean
- number
- integer
- array of string : string[]
- array of number : number[]
- array of boolean : boolean[]
- array of integer : integer[]

A Panel SHALL have a default Data Model whose shape can either be an Object or an Array as defined by `type` property in the Panel. The Fields inside the Panel will map in that Data Model using their name, unless they have a `dataRef` property defined on them.

A Panel SHALL only map to the properties in the data whose value is either an object or an array. Any other mapping would be considered invalid and the Fields inside that panel SHALL not be bound to any entity in the data model,unless there is explicit mapping defined using the `dataRef` property.

When a Panel SHALL define an explicit binding, its Data Model would be a merge of its default and explicit Data Models, and in case of collision, the properties of implicit Data Model will take precedence

For example, the following Form

```
{
 "items": [
    {
      "name" : "name",
      "fieldType" : "text-input",
      "label" : {"value": "Name"}
    },
    {
      "name" : "dependents",
      "fieldType" : "panel",
      "label" : {"value": "Dependent Names"},
      "minItems" : 1,
      "items" : [{
        "fieldType" : "text-input",
        "label" : {"value": "Name"},
        "type" : "string"
      }]
    }
 ]
}
```

can have the following data model

```
{
  "name" : "john doe",
  "dependents" : ["rob doe"]
}
```

Data Bindings

In majority of the use cases, the structure of Form doesn't match the structure of its Data Model. To support that, this Form specifications provides a keyword `dataRef` which specifies an explicit binding of a Form Field with the Data Model.

The Specification supports three types of Data Bindings: `None Binding`, `Name Bindings` and `Explicit Bindings`.

Name Bindings

Be default the Name of the Form Element is used to bind to the property in Data Model of its Ancestor. For a Field/Panel the Name Binding is done relative to the Data Model of its closest Parent Panel in the hierarchy which has not defined None Binding.

In the current AEM Implementation Name Bindings resolve to a separate section of the Data Model and are not merged with the explicit Data Model. The separate Section is termed as the Unbound Section of the Data Model.

None Bindings

A Form element can specify none binding by setting the value in `dataRef` as the JSON value `null`. When a Field specifies `None Bindings` then its value is not captured in the Data Model and is not submitted. When a Panel specifies `None Bindings` then all its children also have `None Bindings`.

A Form MUST not have `None Bindings`

In the current AEM Implementation, When a Panel specifies `None Bindings`, its Name Bindings are not merged with the Parent's Data Model, but resolve to the unbound section of the Data Model

Explicit Bindings

A Form element can bind to a property in the Data Model using explicit binding by setting the value of the `dataRef` property to a JSON Formula expression pointing to an element in the Data Model. For the purpose of the `dataRef` only a subset of rules from the JSON Formula grammar are used with two differences (as defined below)

```
grammar AFDataRef;

dataref : expression EOF ;

expression
  : expression '.' identifier # chainExpression
    | expression bracketSpecifier # bracketedExpression
    | bracketSpecifier # bracketExpression
    | identifier # identifierExpression;

bracketSpecifier
  : '[' INT ']' # bracketIndex
  | '[' '*' ']' # bracketStar;

identifier
  : NAME
  | STRING;

NAME : [a-zA-Z_] [a-zA-Z0-9_]* ;

fragment INT
  : '0'
  | [1-9] [0-9]*
  ;

STRING
  : '"' (ESC | ~ ["\\])* '"';
```

The bracket specifier doesn't support Negative Integers but only +ve Integers

- [x] creates a new element in the array and the last entry is mapped to the Field/Panel. If the array doesn't exist a new array is created with single value and that value is mapped to the Form

In case the element doesn't exist in the data model, it SHALL be created in the Data Model

Form Creation from Existing Data Models

PDF Forms.Next spec doesn't talk about Data Schema but it is needed in Adaptive Forms.

Form Document SHALL be created from an existing Schema.The Implementations SHOULD atleast provide the current level of support that exists in AEM mentioned below

JSON Schema

When the Form is created from a JSON Schema the following validations in the Schema SHALL be honored by the implementations as per the following guidelines

- Type construct SHALL be honored when it is a string value. i.e. `types : [string, boolean]` SHALL be ignored.
- Validations construct defined for JSON types number, string and boolean SHALL be honored.
- enum constraint SHALL be honored
- Structure of the Data Model SHALL conform to the Schema.
- minItems, maxItems, uniqueItems Validations defined in the array types SHALL be honored.

Any other construct not mentioned above SHALL be left to the discretion of the implementation.

FDM

When the Form is created from FDM, the data SHALL conform to the FDM completely

XSD

When the Form is created from a XSD, then the validations in the Schema SHALL be honored by the implementation as per the following guidelines

- The Form SHALL support the following types from the SCHEMA
  - xs:string
  - xs:boolean
  - xs:unsignedInt
  - xs:int
  - xs:decimal
  - xs:date
  - xs:enumeration

    - Any complex type element
- use=required constraint SHALL be honored for all the Types
- default constraint as defined in the XSD specification
- minOccurs and maxOccurs constraint defined for Complex Element
- The other constraints that SHALL be honored by the Form are totalDigits, maximum, minimum, exclusiveMaximum, exclusiveMinimum, minLength, maxLength, length, fractionDigits
- pattern constraint when it is a Regular Expression conforming to ECMA-262, section 21.2.1

Localization

Localization is the process of adapting an application for a particular language or region. It includes translating text content (visible or non-visible), as well as adapting date formatting, number formatting, collation and sorting, text search, and more.

Properties like *label*, *description*, *placeholder*, *enumNames*, *enum* and *screenReaderText* should be localized based on the language or region.

Data Formatting

In Adaptive Forms, we allow users to enter and see value in their choice of format, while authors can specify the format in which these values are saved in the backend. We support

Edit Format

The format in which the values are entered by the user in the Field. This is represented by the editFormat property in the Field

Display Format

The format in which the values are displayed to the user when they have exited the Field. It can be provided using the displayFormat property on the Field

Data Format

The Format in which the values are displayed to the user when it is submitted or exported as represented in the Data Model.

Language and Symbols for Formatting

This section is under development

We are currently using XFA Picture Clause to define the formats but it is going to change.

DFL 1.1 uses unicode CLDR.