**Appup Table <appup-table/>**

Appup table is a component that converts raw (JSON) data into HTML table format with ease and flexibility. And it should be able to work with any CSS framework because of the HTML table generated.

Fields definition is a unique feature in Appup table that allows you to define which fields to be used and manipulated before displaying.

You begin by defining data fields for your JSON data structure that you want to present in HTML table. Then, progressively configuring it to display the data the way you want.

## 

## **Features**

* Work with data from API endpoint or existing data array/object.
* Define fields to map your JSON data structure for display.
* Customize your field data display with formatter if needed.
* Advanced field customization can be done via scoped slot and field component.
* Single sort or multi-sort if your API is supported.
* Pagination components included, swappable and extensible, or write your own.
* Optional detail row to display additional data for each row.
* Adaptable to any CSS framework if it supports HTML table.

You'll need to use at least the following prop to initially configure Appup Table

* **api-url** -- the source of the JSON data
* **fields** -- the fields definition of the data you want to display
* **data-path** -- the path of the data inside your JSON
* **pagination-path** -- the path of the pagination metadata inside your JSON

## **API Mode**

In API mode, which is the *default* mode, Appup Table will interact with an API endpoint to request data from it via AJAX request.

Usually, the API will support sorting, filtering, and pagination of the data. But in order to do so, the client needs to send additional query with the request indicating that it wants one or all of those functionalities. This usually is done by appending additional key-value pair to the query string of the request.

So, all of the features specific to API mode usually relating to manipulating this API related payload, including sorting, filtering, and pagination.

Here are the list of properties that will only work in API mode

* **[load-on-start](https://www.vuetable.com/api/vuetable/properties.html#load-on-start)**
* **[api-url](https://www.vuetable.com/api/vuetable/properties.html#api-url)**
* **[api-mode](https://www.vuetable.com/api/vuetable/properties.html#api-mode)**
* **[http-method](https://www.vuetable.com/api/vuetable/properties.html#http-method)**
* **[reactive-api-url](https://www.vuetable.com/api/vuetable/properties.html#reactive-api-url)**
* **[query-params](https://www.vuetable.com/api/vuetable/properties.html#query-params)**
* **[append-params](https://www.vuetable.com/api/vuetable/properties.html#append-params)**
* **[http-options](https://www.vuetable.com/api/vuetable/properties.html#http-options)**
* **[http-fetch](https://www.vuetable.com/api/vuetable/properties.html#http-fetch)**
* **[sort-params](https://www.vuetable.com/api/vuetable/properties.html#sort-params)**

## **Data Mode**

In Data mode, you tell Appup table that you don't want it to handle data request, but you will supply that data to it instead. It will only format and display the data in each column as you specified in the fields definition.

To turn on Data mode, you set **api-mode** prop to **false** and set your data via **data** prop or **setData** function.

|  |
| --- |
| **<appup-table ref="table"  :api-mode="false"  :data="localData" ></appup-table>** |

Data manipulation features like sorting, filtering, and pagination will not work by default as Appup table relies on server side to provide such functionality.That means you have to manually manipulate the data (sorting and filtering) before you assign the manipulated data to Appup table.

## **The Data Manager**

In Data mode, whenever the user interact with Appup table that will indicate a request for a new data, e.g. changing sort order, filtering the data, move to another page, etc., Appup Table will call the function defined in data-manager prop with two parameters:

* **sortOrder** -- the current sort order array
* **pagination** -- an empty pagination data object

You can use these parameters to do necessary processing to the data and return it back from the function. Appup table will use this returned data to do its job.

You can either return it as

* an array of data, or
* an object containing an array of data and a pagination metadata.

Basically, the data manager function will act as a API endpoint that Appup table tries to request the data from.

# **Appup Table Properties:**

## **api-mode**

* type:**Boolean**
* default:**true**
* Usage: By default, Appup Table will request the data from an API endpoint specified in **api-url.**
* However, you can instruct Appup Table to use your existing data instead by setting api-mode to false (also known as "Data mode"). Then, Appup Table will look for the data in the data prop.
* The data you supply to data prop must be of the JSON structure that Appup Table can work with. Please see [JSON Data Structure] for more detail.
* Please note that disabling api-mode (by setting it to false) will also disable all API related functionalities, like sorting, filtering, etc. But you can use data-manager prop to define the data manager function that can perform those functionalities to the data before Appup Table is using it.
* Note : Setting api-mode to false is not recommended. It will not scale well when you have to handle a large dataset yourself instead of letting database manager at the backend handles it for you.

## **api-url**

* **works in *API mode* only**
* **type:String**
* **default**:**''*(empty string)***
* **Usage**: The URL of the api endpoint that Appup Table will interact with.
* If the API supports sorting, filtering, pagination of the data, Appup Table can automatically append appropriate information to the server via query string.
* Note
* Appup Table does not do sort or know how to sort your data. It is just a presentation layer of your data returned from the server side (API endpoint).
* In API mode, your API endpoint must be able to accept sort options; otherwise, the sorting will not work. Appup Table can send appropriate request with sort option when the user interacts with it. See [Sorting] for more detail.
* In Data mode, you, as a developer, are responsible for sorting the data using any mechanism of your choice and then supply that sorted data to Appup Table to display.

## **append-params**

* **works in *API mode* only**
* **type:Object**
* **default**:**{}*(empty object)***
* **Usage**: Additional parameters that Appup Table should append to the query string when requesting data from the server.

## **css**

* type:Object
* default:{} (empty object)
* usage:
* This is where you should override the CSS classes that Appup Table uses to render HTML table that should help you style the table to your needs.

## **data**

* works in *Data mode* only
* type:Array | Object
* default:null
* Usage: The data that Appup Table will be used to render the table when api-mode is set to false.

## **data-path**

* works in *API mode* only
* type:String
* default:data
* usage:
* The path inside the data structure that contains actual the data.
* TIP
* If the data is at the root of the structure, set this prop to empty string, e.g. data-path="".

## **detail-row-component**

* type:String
* default:''*(empty string)*
* usage:
* A component name to be used as the content of detail row to be displayed underneath the current row.

## **detail-row-transition**

* type:String
* default:''*(empty string)*
* usage:
* The CSS class to apply to detail row during transition.

## **detail-row-class**

* type:String
* default:appup-table-detail-row
* usage:
* The CSS class to apply to the detail row.

## **detail-row-options**

* type:Object
* default:{}*(empty object)*
* Usage: Object to be passed to the detail row component as options prop.

## **event-prefix**

## **fields**

* type:Array
* default:*none*
* required: *true*
* Usage: Array of field definition that Appup Table will be used to map to the data structure in order to render the table for presentation.

## **first-page**

* works in *API mode* only
* type:Number
* default:*1*
* usage:
* First page number. Set this prop to 0 for zero based pagination.
* If the first page of your API endpoint is 0, you can use this prop to set it.

## **header-rows**

* type:Array
* default:*['Appup TableRowHeader']*
* usage:
* Array of row header components to be rendered as table header.

## **http-fetch**

* works in *API mode* only
* type:Function
* default:null
* usage:
* Allow specifying external HTTP request function to fetch the data via AJAX.
* If null, Appup Table will fallback to using axios internally.
* If specified, Appup Table will call the given function passing apiUrl and already constructed httpOptions to the function.

Here's an example using vue-resource

|  |
| --- |
| //...  myFetch(apiUrl, httpOptions) {  return Vue.$http.get(apiUrl, httpOptions)  },  //... |

## **http-method**

* works in *API mode* only
* type:String
* default:get
* Usage: Only support get or post method. Please note that it must be the lowercase string.

## **http-options**

* works in *API mode* only
* type:Object
* default:{}*(empty object)*
* Usage: Allow passing additional options to the server during AJAX call. Internally, Appup Table uses [axios](https://github.com/mzabriskie/axios)to handle AJAX request.

## **initial-page**

* works in *API mode* only
* type:Number
* default:1
* Usage: The initial page number of data to be requested the first time Appup Table is loaded.

## **load-on-start**

* works in *API mode* only
* type:Boolean
* default:true
* required: *true*
* Usage: Whether Appup Table should immediately load the data from the server after finish initialization.

## **min-rows**

* type:Number
* default:0
* Usage: The minimum number of rows that should be displayed when rendering the table.
* If the number of row available is less than the number specified in min-rows prop, Appup Table will render empty table rows to satisfy that minimum rows.

## **multi-sort**

* works in *API mode* only
* type:Boolean
* default:false
* Usage: Enable multiple sort columns interaction.
* When this is enabled, user can press a modifier key (specified in multi-sort-key prop) to do subsequent sorting of the sort result.
* Note
* In multi-sort mode, the sort icon in subsequent sort column will appear dimmer to indicate the level of sorting.

## **multi-sort-key**

* works in *API mode* only
* type:String
* default:alt
* possible values: alt, ctrl, shift, meta
* usage:
* The key to trigger column adding/subtracting when multi-sort is enabled.

## **per-page**

* works in *API mode* only
* type:Number
* default:10
* usage:
* The number of data to be requested per page.

## **query-params**

* works in *API mode* only
* type: [Object, \_Function`]
* default:

|  |
| --- |
| {  sort: 'sort',  page: 'page',  perPage: 'per\_page' } |

* Usage: The text to be used as keys in query string that will be sent to the server. If your API endpoint uses different keys, you can specified them via this prop.
* When you set query-params prop as a \_Function, you can completely override the behevior ofquery-prams` prop on how the query string is constructed.
* Appup Table will pass the following parameters to the given function:
  + sortOrder -- the current sort order
  + currentPage -- the current display page
  + perPage -- the current page size
* The function must return an object containing key-value pair(s) to be constructed as query string; otherwise, and empty object{} will be returned instead.

## **reactive-api-url**

* works in *API mode* only
* type:Boolean
* default:true
* Usage: This tells Appup Table whether it should watch for the change in api-url prop and refresh the data automatically.

## **row-class**

* type:String, Function
* default:''*(empty string)*
* usage:
* The CSS class name that will be applied to each table row.
* If row-class prop is a \_Function`, Appup Table will automatically call the given function on each row, passing the row data and row index to it. Appup Table will then use the returned string from the given method as CSS class for that row.
* Here is the example on using a method to return the row class for styling.

|  |
| --- |
| <template>  <appup-table>  :row-class="onRowClass"  ></appup-table> </template> <script>  //..  methods: {  onRowClass (dataItem, index) {  return (dataItem.isOverdue) ? 'color-red' : 'color-white'  }  } </script> |

**show-sort-icons**

* type:Boolean
* default:true
* Usage: This tells Appup Table whether or not icons should be added to th elements whenever a given column is sortable.

## **sort-order**

* works in *API mode* only
* type:Array
* default:[]*(empty array)*
* usage:
* The default sort order that Appup Table should use when requesting the data from server.

## **sort-params**

* works in *API mode* only
* type:Function that returns String
* default:null
* usage:
* If assigned, this function will be called by Appup Table passing the current sort orders array as a parameter. You can use it to override how the sort parameters are constructed as part of the query string that will be sent to the API endpoint.
* The function must return a string to be included in the query string of API request.

## **no-data-template**

* type:String
* default:''*(empty string)*
* usage:
* Text to be displayed when there are no records in the table. It also support HTML.
* The text will be inserted into a td which is spaned the whole row.

## **pagination-path**

* works in *API mode* only
* type:String
* default:links.pagination
* usage:
* The pagination path inside the data structure that contains the pagination information.
* If the your data from the server does not have pagination metadata, you should set the prop to empty string, e.g. pagination-path="", to suppress warning from Appup Table.

## **table-height**

* type:String
* default:null
* usage:
* Fix the height of table body.
* When set, the height of the body of Appup Table will be set at the given value and vertical scrollbar will appear automatically when the content of table body is longer than the given value.

## **track-by**

* type:String
* default:id
* Usage: The key that uses to unqiuely identified each row in the data array to help track the state of detail row and checkbox features of Appup Table. This is necessary for the detail row and checkbox features to function correctly.
* For detail row, whenever the user clicks to expand the detail row, Appup Table will insert the id of that row into its internal array (visibleDetailRows). And when that detail row is hidden, the idof that detail row is removed from the array.
* For checkbox, when the user selects (checked) a row, Appup Table will insert the id of the row into its internal array (selectedTo). And when that row is unselected (unchecked), the id of that row is removed from the array.

## **transform**

* works in *API mode* only
* type:Function
* default:null
* Usage: A function that allows the user to programmatically transform the receiving data into the one can Appup Table can works with.
* When defined, the given function will be called whenever Appup Table recieves the requested data from the server, before the data are displayed. This provides a hook to manipulate the data that developer might have control over its format, transforming the data into the format that Appup Table can be used.
* The function will receive the raw data returned from the server as its parameter. It must returns the already transformed data; otherwise, the table might appear as empty.

# **Appup Table Data**

## **eventPrefix**

* type:String
* default:appup-table:
* Usage: The event prefix that Appup Table is going to use when emitting the event.

## **tableFields**

* type:Array
* default:[]*(empty array)*
* Usage: The normalized version of fields definition. This is done during the created hook.

## **tableData**

* type:Array
* default:null
* Usage: In api-mode, this stores the data that returned from the server after the sucessful AJAX request. Otherwise, it stores the data assigned to via data prop or setData method. Appup Table always use tableData for table rendering.

## **tablePagination**

* type:Object
* default:null
* Usage: If the data returned from the server contains pagination information specified in the pagination-path, this is where it gets stored.

## **currentPage**

* type:Number
* default:1
* usage:
* Appup Table use this to keep track of the current page being diplayed.

## **selectedTo**

* type:Array
* default:[]*(empty array)*
* Usage: When \_\_checkbox field option is used and the user selected/unselected any checkbox, its row indentifier is either stored in or remove from here. The row identifier can be specified using track-by option.

## **visibleDetailRows**

* type:Array
* default:[]*(empty array)*
* Usage: This stores the row identifier of any row where its detail row is visible.

# **Appup Table - Computed Properties**

## **countTableData**

* return:Number
* Usage: Return the number of rows for the current data or zero (0) if the tableData is null.

## **countVisibleFields**

* return:Number
* Usage: Return the number of visible fields in the table by checking the field's visible option.

## **displayEmptyDataRow**

* return:Boolean
* Usage: Determine if Appup Table should display empty data row message.

## **lessThanMinRows**

* return:Boolean
* Usage: Determine if the number of data rows available is less than the number specified in min-rowsprop.

## **blankRows**

* return:Number
* Usage: Return the number of blank rows that Appup Table needs to render. If the number of row data is greater than or equal to min-rows, it returns 0.

## **useDetailRow**

* return:Boolean
* Usage: Determine if detail row should be rendered by inspecting the availability of the data and various properties.

## **isApiMode**

* return:Boolean
* Usage: Determine if Appup Table is currently in API mode.

## **isDataMode**

* return:Boolean
* Usage: Determine if Appup Table is currently in Data mode.

# **Appup Table - Methods**

## **normalizeFields**

* params:*none*
* Usage: Parse fields definition to field objects usable by Appup Table. This method is called automatically "once" during the created life cycle hook.
* If you dynamically change the fields prop, you will need to manually call normalizeFieldsmethod to properly parse the fields definition as Appup Table will not be able to pickup the change and will not work as expected.

## **setData**

* params:
  + data: Array | Object
* Usage: You can use this method to manually set the data that Appup Table will be used for table rendering instead of requesting data from the server.
* If the data parameter is of type Array, Appup Table will use those array as the data to render the table.
* If the data parameter is of type Object, it must be conform to the [Data Structure](https://www.vuetable.com/api/vuetable/methods.html#) that Appup Table expects (e.g. contains both data and pagination information).
* Note The method will automatically set api-mode to false.

## **reload**

* params:*none*
* Usage: Force Appup Table to reload the data from the server using the current value of parameters. However, the page number will not be reset.

## **refresh**

* params:*none*
* Usage: Force Appup Table to reload the data from the server and the page number will be reset to 1. It's the same as using goto-page page event to load page 1.

## **resetData**

* params:*none*
* Usage: This will set tableData and tablePagination to null resulting in not displaying any data in the table (as there is no data to display). This method will also fire appup-table:data-reset event which can be captured to force update pagination component accordingly.

## **transform**

* params:
  + data: Array
* must return: Array
* Usage: In case, the data returned from the server is not in the format that Appup Table expected, you can define transform method on the main Vue instance and Appup Table will automatically called it with the data from the server.
* The transform method must return the array of data that Appup Table expected.
* See also: Data Format (JSON)

## **gotoPreviousPage**

* params:*none*
* Usage: \*\* API Mode Only \*\*
* This method will automatically request the previous page of data from the server.

## **gotoNextPage**

* params:*none*
* Usage: \*\* API Mode Only \*\*
* This method will automatically request the next page of data from the server.

## **gotoPage**

* params:
  + page: Number
* Usage: \*\* API Mode Only \*\*
* This method will automatically request the specified page of data from the server.

## **changePage**

* params:
  + page: String, Number
* Usage: \*\* API Mode Only \*\*
* This method will automatically request the specified page of data from the server. You can either pass in the page number, or 'prev' string for previous page, or 'next' string for next page.

## **isVisibleDetailRow**

* params:
  + rowId: RowIdentifier
* Usage: Determine if the detail row of the given row identifier is marked as visible.

## **showDetailRow**

* params:
  + rowId: RowIdentifier
* Usage: Force displaying the detail row of the given row.

## **hideDetailRow**

* params:
  + rowId: RowIdentifier
* Usage: Force hiding the detail row of the given row.

## **toggleDetailRow**

* params:
  + rowId: RowIdentifier
* Usage: Toggle the display of the detail row of the given row.

## **showField**

* params:
  + index: Number
* Usage: Force displaying the specified field.

## **hideField**

* params:
  + index: Number
* Usage: Force hiding the specified field.

## **toggleField**

* params:
  + index: Number
* Usage: Toggle display of the specified field.

# **Appup Table Events**

## **appup-table:initialized**

* params:
  + tableFields: Array -- the normalized fields definition
* Usage: This event will be fired after the fields definition have been normalized during the createdlifecycle hook.

## **appup-table:header-event**

* params:
  + type: String
  + payload: Object
* Usage: This event will be fired when a table header part of Field Component wants to send any information back to its parent Appup Table.

## **appup-table:field-event**

* params:
  + type: String
  + payload: Object
* Usage: This event will be fired when a table data part of Field Component wants to send any information back to its parent Appup Table.

## **appup-table:pagination-data**

* params:
  + tablePagination: Object -- pagination information
* Usage: This event will be fired when the data has sucessfully been retrieved from the server and there is pagination information available.

## **appup-table:loading**

* params:*none*
* Usage: This event will be fired before Appup Table starts to request the data from the server. This is useful for triggering the display of loading image.

## **appup-table:load-success**

* params:
  + response: Object -- the response returned from the server
* Usage: This event will be fired when the data was successfully loaded from the server.

## **appup-table:load-error**

* params:
  + response: Object -- the response returned from the server
* Usage: This event will be fired when up the data cannot be retrieved from the server or the server responses with an error.

## **appup-table:loaded**

* params:*none*
* Usage: This event will be fired after Appup Table got response back from the server. This event does not indicate whether the request was successful or failed. It just indicates that the request is finished and it got the response back.
* This is useful for ending/hiding the loading image.

## **appup-table:row-changed**

* params:
  + dataItem: Object -- the current data item
* Usage: This event will be fired when Appup Table loops through the data during table row rendering. This will be useful if you want to do some processing with the data, e.g. summing up the values.

**Important!**

**Please note that you MUST NOT change the pass-in data item or try to update any instance data during this event, or it will cause "indefinite update loop". The only way to work inside this event is to use the variable define outside of Vue.js instance.**

## **appup-table:row-clicked**

* params:
  + dataItem: Object -- the current data item
  + event: Event -- the MouseObject event
* Usage: This event will be fired when the user clicked on any column in the row. You can use the pass-in event object to target the DOM element that you want to manipulate. The pass-in data item argument is the actual data that Appup Table received from the server and it is reactived. Which means if you changed its value, the data displayed in the table will also be changed.
* Changing the pass-in data in this event will not cause "indefinite update loop" However, the change only affects the current displaying data. It does not change anything on the server side.

## **appup-table:row-dblclicked**

* params:
  + dataItem: Object -- the current data item
  + event: Event -- the MouseObject event
* Usage: This event will be fired when the user "double-clicked" on any column in the row. You can use the pass-in event object to target the DOM element that you want to manipulate. The pass-in data item argument is the actual data that Appup Table received from the server and it is reactived. Which means if you changed its value, the data displayed in the table will also be changed.
* Changing the pass-in data in this event will not cause "indefinite update loop" However, the change only affects the current displaying data. It does not change anything on the server side.

## **appup-table:cell-rightclicked**

* params:
  + dataItem: Object -- the current data item
  + field: Object -- the field object that causes this event
  + event: Event -- the MouseObject event
* Usage: This event will be fired when a cell in the table body has been right-clicked.

## **appup-table:cell-clicked**

* params:
  + dataItem: Object -- the current data item
  + field: Object -- the field object that causes this event
  + event: Event -- the MouseObject event
* Usage: This event will be fired when a cell in the tabel body has been clicked.

## **appup-table:cell-dblclicked**

* params:
  + dataItem: Object -- the current data item
  + field: Object -- the field object that causes this event
  + event: Event -- the MouseObject event
* usage:This event will be fired when a cell in the table body has been double-clicked.

## **appup-table:detail-row-clicked**

* params:
  + dataItem: Object -- the current data item
  + event: Event -- the MouseObject event
* usage:This event will be fired when an area of detail row has been clicked.

## **appup-table:checkbox-toggled**

* params:
  + isChecked: Boolean -- the state of the checkbox
  + dataItem: Object -- the current data item
* usage:This event will be fired whenever the checkbox has been toggled.

## **appup-table:checkbox-toggled-all**

* params:
  + isChecked: Boolean -- the state of the checkbox
* usage:This event will be fired when the select-all checkbox has been toggled.

## **appup-table:data-reset**

* params:*none*
* Usage: This event will be fired as a result from calling resetData method to clear in tableData and tablePagination.

# **Appup Table IN BRIEF**

## 

**Fields**

* **type**: **Array**
* **default**: **none**
* **required**: **true**
* **Description**: Array of field definition that Appup table will be used to map to the data structure in order to render the table for presentation.

Fields definition is an array of field definition object describing which field of the data you want Appup Table to generate to HTML table. It also has various options to let you take full control of how the field data would be displayed.Fields can be defined as simple array of string, array of object, or the mix.

* Fields defined as simple array

|  |
| --- |
| let columns = ['name', 'email', 'birthdate', 'gender'] |

* Fields defined as array of object

|  |
| --- |
| let columns = [  {  name: 'name',  sortField: 'name'  },  {  name: 'email',  title: 'Email Address'  },  {  name: 'birthdate',  sortField: 'birthdate',  titleClass: 'center aligned',  dataClass: 'center aligned',  callback: 'formatDate|D/MM/Y'  },  {  name: 'gender',  sortField: 'gender',  titleClass: 'center aligned',  dataClass: 'center aligned',  callback: 'gender'  }, ] |

* Fields defined as array of the mix

|  |
| --- |
| let columns = [  'name',  'email',  {  name: 'birthdate',  sortField: 'birthdate',  titleClass: 'center aligned',  dataClass: 'center aligned',  callback: 'formatDate|D/MM/Y'  },  {  name: 'gender',  sortField: 'gender',  titleClass: 'center aligned',  dataClass: 'center aligned',  callback: 'gender'  },  //... ] |

During the initialization, Appup table will convert each field to field definition object containing a number of options with their default values.

Appup table normally works with data from an API endpoint and fields definition. If you prefer to handle the data retrieval yourself, you can switch to use Appup table in Data mode, by turning the API mode off with :api-mode="false" and supply your data through data prop or setData function.

|  |
| --- |
| <template>  <appup-table ref="appup-table"  :fields="['name', 'nickname', 'email', 'gender']"  :api-mode="false"  :data="localData"  ></appup-table> </template> |

If it is working with an API endpoint to retrieve the data, it is called **API mode**. If it is working with an existing data supplied manually, it is called **Data mode**. However, most of the features work regardless of the source of the data.

## **Field options**

### **name**

* **Type**: String | ComponentDefinition
* **Usage**:

Name of the data field to be displayed. The name can also be the name of a *globally* registered [Field Component](https://www.vuetable.com/guide/special-field.html#field-component), a Field Component definition, or a slot name defined inside <appup-table> tag.

* If the name is of type String, Appup table checks in the following order:
  + if the name is begins with \_\_ (double underscore) or appup-table-fields- ([field Prefix](https://www.vuetable.com/guide/fields-definition.html)), then it is a Field Component.
  + if there is a scoped slot of that name defined inside <appup-table> tag, then it is a Field Slot.
  + otherwise, it defaults to column name in the data.

### **sortField**

* **Type**: String
* **Usage**: Usually, it will be the same as name option. But you can specify different value if you use different field name when querying data on the server side, e.g. firstname.
* If specified, the field will be marked as sortable. Appup Table will display appropriate clickable icon after the field title. Appup Table will also make a new request to the server with the sort=<sortField> query string appended.

### **title**

* **Type**: String | Function
* **Usage**: If you would like to specify the title yourself, you can put it in here. Otherwise, appup-table will try to guess it from the name option instead.
* You can even put the icon class in the title, see example below

|  |
| --- |
| {  name: 'birthdate',  title: '<i class="orange birthday icon"></i> Birthdate' } |

### **titleClass**

* **Type**: String
* **Usage**: The css class you would like to apply for the title of this field.

### **dataClass**

* **Type**: String
* **Usage**: The css class you would like to apply for the data of this field.

### **formatter**

* **Type**: Function
* **Usage**: The formatter function to be called by Appup Table to format the value of the column to be displayed.
* The formatter function will receive the following parameters from Appup Table, and it must return the value that Appup Table will be used to display in that column.
  + value -- value of the field
  + appup-table -- a reference to Appup Table itself  
    This makes it possible to separate the fields definition to a separate file while still allowing it to have access to Appup Table functionality when needed.
* Here is the example of a field defining a formatter function.

|  |
| --- |
| // define as a function reference. // in this case, function `gender` must be defined in the `methods` section. {  name: 'gender',  formatter: gender } // or, a reference to a function defined in other place {  name: 'gender',  formatter: Util.formatGender }  // define as inline function {  name: 'gender',  formatter: (value) => (value === 'M') ? 'Male' : 'Female' } |

* In this case, if the value of the gender field gets passed to the formatter function is M, it will return Male. Appup Table will display Male for this field instead of M.

### **visible**

* **Type**: Boolean
* **Usage**: Whether this field should be visible or hidden when rendering the table.

### **width**

* Type: String
* Usage:
* The width of the column, e.g. '200px', '10%', '2rem'.
* The given value will be applied to the specified field on both table header and data.

### **$\_index**

* Type: Number
* Usage: The $\_index option gets inserted automatically by Appup Table to indicate the position of the field in Fields defintion array. This can be useful when using Appup Table's function to hide/show specific fields as it requires the index position of the field.

## **Referencing Nested JSON Data**

If the JSON data structure contains nested objects, eg:

|  |
| --- |
| [  {  "id": 1,  "name": "xxxxxxxxx",  "email": "xxx@xxx.xxx",  "group\_id": 1,  "address" {  "street\_address":"123 abc place",  "city":"townsville",  "province":"Harbor",  "country":"Antegria"  }  }  .  .  .  {  "id": 50,  "name": "xxxxxxxxx",  "email": "xxx@xxx.xxx",  "group\_id": 3,  "address" {  "street\_address":"456 xyz street",  "city":"cityville",  "province":"Portia",  "country":"Norland"  }  } ] |

In this case, the column names of nested objects can be specified in the following format:

|  |
| --- |
| let columns = ['name', 'email', 'address.city', 'address.country'] |

# **Sorting**

If your API endpoint supports sorting, Appup Table can also automatically interact with it when you specify which column in the data is sortable.

To specify that a particular column is sortable, you add [sortField](https://www.vuetable.com/guide/fields-definition.html#sortfield) option to the field definition object of that column.

|  |
| --- |
| {  name: 'email',  sortField: 'email'  } |

The sortable column will be rendered with .sortable CSS class in the <th> and it will respond to the click to toggle between

* setting the column as sort order if it was not the current sort order.
* toggle between ascending and descending if it is the current sort order.

What it does is Appup Table will send a new request to the server endpoint with sort parameter specifying which sortOrder.field and which sortOrder.direction it expects from the server.

Here is the sample sort parameter in the query string.

|  |
| --- |
| http://api.example.com/users?sort=email|asc |

where, **sort=email|asc** is the **sort** parameter constructed from the default sort order field **email** and the default sort direction **asc**. They are concatenated by a pipe character (|).

The "sort" parameter key and the way its value constructs can be customized.

* Use query-params prop to change the "key" name from sort.
* Use sort-params to change the sort value format.

If your API endpoint uses different construct, you can override this behavior by supplying your own function to construct the sort parameter. See below for more info.

## **Initial Sorting Order**

To provide initial sorting order, use [sort-order](https://www.vuetable.com/api/vuetable/properties.html#sort-order) prop to specify the default sorting column.

|  |
| --- |
| <appup-table  //...  :sort-order="sortOrder" ></appup-table> new Vue({  //...  data: {  //...  sortOrder: [  {  field: 'email',  direction: 'asc'  }  ]  } }) |

## **Multi-column Sorting**

Multi-column sorting can be enabled by setting multi-sort prop value to true.

Once enabled, the user can use Alt+Click (Option+Click on mac) to work with multi-column sorting feature. If you would like to use other key than the alt, use [multi-sort-key](https://www.vuetable.com/api/vuetable/properties.html#multi-sort-key) prop to specify one of the following value

* alt -- Alt / Option
* ctrl -- Ctrl / Control
* meta -- Window / Command
* shift -- Shift

## **Overriding the Sort Query String**

You can override how the sort parameter in the query string is constructed by assigning a function to [sort-params](https://www.vuetable.com/api/vuetable/properties.html#sort-params) prop. The function will receive the current array of [sortOrder](https://www.vuetable.com/api/vuetable/properties.html#sort-order) as a parameter, and it must return a string value for the sort query string.

|  |
| --- |
| <template>  <appup-table   api-url="..."  :fields="fields"  :sort-params="getSortParam"  ></appup-table> </template> // // If // sortOrder = [ // { field: "gender", direction: "desc"}, // { field: "name", direction: "asc"}, // ] // This will return // '-gender,name' // <script>  //..  methods: {  getSortParam(sortOrder) {  return sortOrder.map(function(sort) {  return (sort.direction === 'desc' ? '-' : '') + sort.field  }).join(',')  }  } </script> |

## **Styling the Sortable Columns**

By default, if you specify the [sortField](https://www.vuetable.com/guide/fields-definition.html#sortfield) in field definition object, the field will have a "sortable" CSS class in its <th> tag.

You can style this "sortable" class the way you want. For example, when the user hover the mouse over a "sortable" field, it changes the foreground color.

|  |
| --- |
| .appup-table th.sortable:hover {  color: #2185d0;  cursor: pointer; } |

## **Displaying Sort Icon on Sortable Columns**

To display sort icon, you have to

* enable it by :show-sort-icons="true"
* set the CSS related properties in the [css](https://www.vuetable.com/api/vuetable/properties.html#css) prop
  + sortableIcon -- sort icon
  + ascendingIcon -- ascending sort icon when the field is currently sorted in ascending order
  + descendingIcon -- descending sort icon when the field is currently sorted in descending order
  + ascendingClass  
    class name to be inserted to <th> when the field is currently sorted in ascending order, otherwise, the class name is removed.
  + descendingClass class name to be inserted to <th> when the field is currently sorted in descending order, otherwise, the class name is removed.

## **Overriding Default Query String**

By default, Appup Table uses the following key in the query string that will be sent to the server.

* sort -- its value will contain the requested sort order
* page -- its value will contain the requested page number
* per\_page -- its value will contain the requested records per page

If you're making your own API backend, you can just use this default.

But if you're using someone's else API or you already have your own existing API that do not correspond to this. You can tell Appup Table what key it should be using instead, by supplying it via [query-params](https://www.vuetable.com/api/vuetable/properties.html#query-params) property.

Here is the default value of query-params property.

|  |
| --- |
| {  sort: 'sort',  page: 'page',  perPage: 'per\_page' } |

Suppose your API uses sort\_order, page\_no, and page\_size, you can easily tell Appup Table to use them like so,

|  |
| --- |
| <appup-table  //...  :query-params="{ sort: 'sort\_order', page: 'page\_no', perPage: 'page\_size' }" ></appup-table> |

You can also completely override how the query string is constructed by assigning a function to query-params instead.

The function will receive 3 parameters which are

* array of current sort order
* current page number
* current page size

The function must return an object of key-value pairs. Here is an example.

|  |
| --- |
| <template>  <appup-table  //...  :query-params="makeQueryParams"  ></appup-table> </template>  <script> export default {  //...  methods: {  makeQueryParams (sortOrder, currentPage, perPage) {  return {  sortBy: sortOrder[0].field,  sortOrder: sortOrder[0].direction,  pageNo: currentPage,  pageSize: perPage  }  }  } } </script> |

# **Special Field**

Special field is an extension to normal fields. It is not part of your data structure, but instead is a slot or component that you defined to handle your data in a special way.

The following are two types of special field in Appup Table.

* Field Slot
* Field Component

## **Field Slot**

Field Slot makes it very easy to customize field rendering in Appup Table.

You can use Field Slot by defining a named [scoped slot](https://vuejs.org/v2/guide/components-slots.html#Scoped-Slots) inside Appup Table tag (as a child element), then, use that slot name in the [name](https://www.vuetable.com/guide/fields-definition.html#name) option of the field definition object. Appup Table will automatically looking for the scoped slot of that name and use it as a column.

Here is an example,

|  |
| --- |
| <appup-table ref="appup-table"  api-url="..."  :fields="fields" >  <div slot="gender-slot" slot-scope="props">  <span v-if="props.rowData.gender === 'M'" class="ui teal label"><i class="large man icon"></i>Male</span>  <span v-else class="ui pink label"><i class="large woman icon"></i>Female</span>  </div> </appup-table> new Vue({  data: {  fields: [  //...  {  name: "gender-slot",  title: '<i class="grey heterosexual icon"></i>Gender',  titleClass: "center aligned",  dataClass: "center aligned",  width: "15%",  },  ]  //...  } }) |

## **Field Component**

Field Component is unique to Appup Table and it is a crucial part in making it extensible and sharable.

It is responsible for rendering both the table header <th> and table data <td> column.

The component will either be called by Appup TableRowHeader component to render the table header of each field (by passing :is-header="true"), or Appup Table itself to render the table column data of the field.

Here is the structure of the single file component (SFC) to help you getting started in creating Field Component for Appup Table.

|  |
| --- |
| <template>  <th v-if="isHeader"  //...  ></th>  <td v-else  //...  ></td> </template>  <script> import Appup TableFieldMixin from 'appup-table-2/src/components/Appup TableFieldMixin'  export default {  mixins: [Appup TableFieldMixin],  //... } </script> |

To summarize,

* Use [formatter](https://www.vuetable.com/guide/fields-definition.html#formatter) function in field definition object first.
* Then, use Field Slot if you need more customization.
* Then, use Field Component if you need to do the same thing more than once or in more than one place.

# **Creating Field Component**

## **Implementation Detail**

Here is the based structure of Field Component.

|  |
| --- |
| <template>  <th v-if="isHeader"  //...  ></th>  <td v-else  //...  ></td> </template>  <script> import Appup TableFieldMixin from 'appup-table-2/src/components/Appup TableFieldMixin'  export default {  mixins: [Appup TableFieldMixin],  //... } </script> |

* Use the provided [Appup TableFieldMixin](https://www.vuetable.com/api/field/mixin.html).
* Importing [Appup TableFieldMixin](https://www.vuetable.com/api/field/mixin.html) mixin and define it in the mixins section. This mixin will declare all necessary props for you. These props will get passed to your component by Appup Table.
* Use <th> tag to render table header and use <td> tag to render table data.
* Use is-header prop together with v-if and v-else to differentiate between table header and table data template. This is still consider as single root as only one tag gets rendered.
* TIP
* Appup Table uses [Appup TableRowHeader](https://www.vuetable.com/api/row/header.html) component to render all table header. If the field definition object is a component, it will activate the component with the :is-header="true".
* Use appup-table:header-event and appup-table:field-event events
  + To emit event in the code that handles table header section, use appup-table:header-event.
  + To emit event in the code that handles table data section, use appup-table:field-event.
* Appup Table has v-on directive to capture appup-table:header-event and appup-table:field-eventevents. [Appup Table](https://www.vuetable.com/api/vuetable/properties.html) and [Appup TableRowHeader](https://www.vuetable.com/api/row/header.html) will relay the event out to Appup Table.

Here are the props declared in [Appup TableFieldMixin](https://www.vuetable.com/api/field/mixin.html)

* is-header
  + type:Boolean
  + default:false
* Indicate whether Field Component should render the "header" or "data" template section.
* title
  + type:String
  + default:""
* The title option from field definition object is passed via title prop. You can simply use titleprop to render column title if there is no special need other than display the column title.
* row-data
  + type:Object
  + default:null
* row-index
  + type:Number
* row-field
  + type:Object
* The field definition object of this field.
* TIP
* Remember that you can use field definition object to hold additional data, parameters, or results.
* appup-table
  + type:Object
  + default:null
* If you need to access the parent Appup Table, use appup-table prop.

## **Using Your Field Component Locally**

Normally, you don't need to register Field Component. Just import it and assign it to the [name](https://www.vuetable.com/guide/fields-definition.html#name)option of the field definition object.

|  |
| --- |
| import MyActionComponent from './components/MyActionComponent.vue'  export default [  {  name: MyActionComponent,  title: 'Action`  },  //...  ] |

## **Registering Field Component Globally**

You can also register your Field Component globally via Vue.component(..) and refer to it in the field definition object by its name.

|  |
| --- |
| // main.js import Vue from 'vue' import MyActionComponent from './components/MyActionComponent.vue'  Vue.component('appup-table-field-action', MyActionComponent) // FieldsDef.js export default [  'code',  'description',  {   name: 'appup-table-field-action'  } ] |

If you name your Field Component with the prefix appup-table-field-, you can even shorten its name in field definition object like this.

|  |
| --- |
| // FieldsDef.js export default [  'code',  'description',  `\_\_action` ] |

# **CSS Styling**

Appup Table is trying to be CSS framework agnostic. As it generates its output as normal HTML table, you should be able to style it with any CSS framework table classes.

To make it easy to adapt to any CSS framework, the CSS classes that would affect Appup Table appearance are store in one file which could easily be assigned to css prop.

## Here is the example of CSS classes

|  |
| --- |
| **export default {  table: {  tableWrapper: '',  tableHeaderClass: 'fixed',  tableBodyClass: 'appup-table-semantic-no-top fixed',  tableClass: 'ui blue selectable unstackable celled table',  loadingClass: 'loading',  ascendingIcon: 'blue chevron up icon',  descendingIcon: 'blue chevron down icon',  ascendingClass: 'sorted-asc',  descendingClass: 'sorted-desc',  sortableIcon: 'grey sort icon',  handleIcon: 'grey sidebar icon',  },   pagination: {  wrapperClass: 'ui right floated pagination menu',  activeClass: 'active large',  disabledClass: 'disabled',  pageClass: 'item',  linkClass: 'icon item',  paginationClass: 'ui bottom attached segment grid',  paginationInfoClass: 'left floated left aligned six wide column',  dropdownClass: 'ui search dropdown',  icons: {  first: 'angle double left icon',  prev: 'left chevron icon',  next: 'right chevron icon',  last: 'angle double right icon',  }  },   paginationInfo: {  infoClass: 'left floated left aligned six wide column',  } }** |

# This makes it easy to create your own if you're using other CSS framework.

## **Usage**

Just import and assign it to css prop of Appup Table.

|  |
| --- |
| <template>  <div>  <appup-table ref="appup-table"  //...  :css="css.table"  ></appup-table>  <appup-table-pagination ref="pagination"  :css="css.pagination"  ></appup-table-pagination>  </div> </template> <script> import CssForBootstrap4 from './Appup TableCssBootstrap4.js' export default {  data () {  return {  //...  css: CssForBootstrap4,  }  } } </script> |

## **Customization for Other CSS Frameworks**

# Normally, you will have replace the CSS class of each option with the one specified in the HTML table section of your CSS framework of choice.

# Here is an example for Bootstrap4 with FontAwesome Icon.

|  |
| --- |
| **// Bootstrap4 + FontAwesome Icon export default {  table: {  tableWrapper: '',  tableHeaderClass: 'mb-0',  tableBodyClass: 'mb-0',  tableClass: 'table table-bordered table-hover',  loadingClass: 'loading',  ascendingIcon: 'fa fa-chevron-up',  descendingIcon: 'fa fa-chevron-down',  ascendingClass: 'sorted-asc',  descendingClass: 'sorted-desc',  sortableIcon: 'fa fa-sort',  detailRowClass: 'appup-table-detail-row',  handleIcon: 'fa fa-bars text-secondary',  renderIcon(classes, options) {  return `<i class="${classes.join(' ')}"></span>`  }  },  pagination: {  wrapperClass: 'pagination float-right',  activeClass: 'active',  disabledClass: 'disabled',  pageClass: 'page-item',  linkClass: 'page-link',  paginationClass: 'pagination',  paginationInfoClass: 'float-left',  dropdownClass: 'form-control',  icons: {  first: 'fa fa-chevron-left',  prev: 'fa fa-chevron-left',  next: 'fa fa-chevron-right',  last: 'fa fa-chevron-right',  }  } }** |

## Rendering Icon

# Each CSS framework seems to use different way to constructed icon. Appup Table provides a way to help you on this.

# Inside the object you assigned to the css prop, if you define a function named renderIcon, Appup Table will call it whenever it needs to render icon.

# **You should define the renderIcon function to accept 2 parameters**

# **classes -- contains array of CSS class string, e.g. ['class-a', 'class-b']**

# **options -- contains other HTML attributes, e.g. style="margin:0px"**

# **Here is the one for Bootstrap 4 with FontAwesome**

|  |
| --- |
| **{  tableWrapper: "",  tableHeaderClass: "mb-0",  tableBodyClass: "mb-0",  tableClass: "table table-bordered table-hover",  loadingClass: "loading",  ascendingIcon: "fa fa-chevron-up",  descendingIcon: "fa fa-chevron-down",  ascendingClass: "sorted-asc",  descendingClass: "sorted-desc",  sortableIcon: "",  detailRowClass: "appup-table-detail-row",  handleIcon: "fa-bars text-secondary",  renderIcon(classes, options) {  return `<i class="${classes.join(" ")}" ${options}></span>`;  } }** |

## **Partial CSS Options**

# If you only want to change one or two options in the CSS and do not want to put together the whole CSS classes as in the above example, you can specify only the option(s) that you want in the css prop.

|  |
| --- |
| **<appup-table ref="appup-table"  :css="{tableClass: 'my-table', loadingClass: 'Please wait...'}" ></appup-table>** |

# **TIP**: When you assign value to css prop, Appup Table will merge it to the default value. The merged copy is stored in $\_css data property.

## **Appup Table Specific CSS**

# There are some specific CSS classes generated by Appup Table during the rendering of HTML table. You can use your browser DevTool or Inspector to see it if you need to target various part of the table.

## **Generated HTML Table**

# Here is the structure of the HTML Table generated by Appup Table

|  |
| --- |
| **<div class="{css.tableWrapper}">  <div class="appup-table-head-wrapper">  <table class="appup-table {css.tableClass} {css.tableHeaderClass}">  // content generated by column group component VeutableColGroup  <thead>  // content generaged by table header component Appup TableRowHeader  </thead>  </table>  </div>   <div class="appup-table-body-wrapper">  <table class="appup-table {css.tableClass} {css.tableBodyClass}">  // content generated by column group component VeutableColGroup  <tfoot>  // content generated by "tableFooter" slot  </tfoot>  <tbody class="appup-table-body">  // content of <tr> and <td> generated by v-for,   // will probably be another component in the future  </tbody>  </table>  </div> </div>** |

# Please note that in the above structure, there are actually two HTML table generated. One for the table header and another for table body and footer. This is necessary to support fixed table header and scrollable table body (by setting table-height prop).

# However, if fixed header is not used, the first table will not be generated. And the table inside <div class="appup-table-body-wrapper"> will contain the thead section as well.

# The <colgroup> tags are also generated for both table to make sure the columns in both table have the same width.

### WARNING

# If using fixed header feature, you should always set the width for each field in fields definition, otherwise, the column width in the table header and body may not be the same.

# 

# **JSON Data Format**

In order for Appup Table to do its task, it needs to work with a certain JSON data structure. Here is the default data format that Appup Table expects:

|  |
| --- |
| {  "links": {  "pagination": {  "total": 50,  "per\_page": 15,  "current\_page": 1,  "last\_page": 4,  "next\_page\_url": "...",  "prev\_page\_url": "...",  "from": 1,  "to": 15,  }  },  "data": [  {  "id": 1,  "name": "xxxxxxxxx",  "nickname": "xxxxxxx",  "email": "xxx@xxx.xxx",  "birthdate": "xxxx-xx-xx",  "gender": "X",  "group\_id": 1,  },  .  .  .  {  "id": 50,  "name": "xxxxxxxxx",  "nickname": "xxxxxxx",  "email": "xxx@xxx.xxx",  "birthdate": "xxxx-xx-xx",  "gender": "X",  "group\_id": 3,  }  ] } |

Mostly, it looks for two pieces of information; the array of data, the pagination information.

It uses two keys (data-path and pagination-path) to look into the returned object to extract the information it is needed.

* data-path points to the array of data
* pagination-path points to the pagination information

In the example above, the data-path is data and the pagination-path is links.pagination (a nested object), which is the default value that Appup Table uses if none is provided.

If you're familiar with [Laravel](https://laravel.com/), you would know that Laravel automatically convert the query data to JSON format when Eloquent objects are returned from application's routes or controllers. And if you use paginate() function, the result would look something like this.

|  |
| --- |
| {  "total": 50,  "per\_page": 15,  "current\_page": 1,  "last\_page": 4,  "next\_page\_url": "...",  "prev\_page\_url": "...",  "from": 1,  "to": 15,  "data": [  {  "id": 1,  "name": "xxxxxxxxx",  "nickname": "xxxxxxx",  "email": "xxx@xxx.xxx",  "birthdate": "xxxx-xx-xx",  "gender": "X",  "group\_id": 1,  },  .  .  .  {  "id": 50,  "name": "xxxxxxxxx",  "nickname": "xxxxxxx",  "email": "xxx@xxx.xxx",  "birthdate": "xxxx-xx-xx",  "gender": "X",  "group\_id": 3,  }  ] } |

In this case, you just specify values for data-path and pagination-path like so

|  |
| --- |
| <div id="app">  <appup-table  api-url="/api/users"  :fields="columns"  data-path="data"  pagination-path=""  ></appup-table>  </div> |

This tells Appup Table that the data is in the path named data inside the JSON structure returned from the server, and the pagination information is in the root of the JSON structure.

# **Data Transformation**

If the data you're working with is not in the format that Appup Table uses, you can setup a function that accepts response data as an argument, to transform it to the format that Appup Table can work with.

By creating a data transformation function, you will be able to pre-process the data you received back from the API endpoint and "transform" them before passing into Appup Table by using transform prop to specify the data transformation function to be used.

|  |
| --- |
| new Vue({  el: '#app',  data: {  //...  },  methods: {  transformData (data) {  var transformed = {}   transformed.pagination = {  total: data.total,  per\_page: data.per\_page,  current\_page: data.current\_page,  last\_page: data.last\_page,  next\_page\_url: data.next\_page\_url,  prev\_page\_url: data.prev\_page\_url,  from: data.from,  to: data.to  }   transformed.mydata = []   for (var i=0; i < data.length; i++) {  transformed.mydata.push({  id: data[i].id,  fullname: data[i].name,  email: data[i].email  })  }  return transformed  }  }  }) <appup-table  api-url="..."  :fields="fields"  :transform="transformData"  data-path="mydata"  pagination-path="pagination" ></appup-table> |

# **Detail Row**

Detail row is the additional row hidden underneath each data row in the table. When you want to only display relevant data for each row, but also would like to display additional information when needed, detail row would be a great help.

In order to use detail row, you will need to create a component and register it globally using Vue.component helper.

|  |
| --- |
| import Vue from 'vue' import MyDetailRow from './MyDetailRow.vue'  Vue.component('my-detail-row', MyDetailRow) <template>  <appup-table ref="appup-table"  //..  detail-row-component="my-detail-row"  ></appup-table> </template> |

Or, you can import and assign it to a variable in data section.

|  |
| --- |
| <template>  <appup-table ref="appup-table"  //..  :detail-row-component="detailRow"  ></appup-table> </template> <script> import MyDetailRow from './MyDetailRow.vue'  export default {  //..  data() {  //..  detailRow: MyDetailRow,  },  //... } </script> |

Then, you can use detail-row-component property to specify the name of your detail row component.

Your detail row component should define the following two properties:

* row-data -- the current row data will be passed to
* row-index -- the current row index will be passed to
* options -- extra object in case you need to pass additional options to your detail row component

# **Pagination**

Appup Table comes ready with two pagination components and a pagination information component which you can extend to easily build your own.

* [Appup TablePagination](https://www.vuetable.com/api/pagination/pagination.html) It is implemented as a sliding window which displays only a certain number of pages with buttons to jump to the first page, previous page, next page, and the last page.

|  |
| --- |
| [First][Prev][1][2][3][4][5][Next][Last] |

* [Appup TablePaginationDropdown](https://www.vuetable.com/api/pagination/dropdown.html) It is implemented as a dropdown button showing the current page. User can click the dropdown to select the page. It also has previous page button and next page button on its side. This pagination component is designed to take less space.

|  |
| --- |
| [Prev][ Page 1 ▼ ][Next] |

This is possible because they are all built on top of the Appup TablePaginationMixin, which contains most of the pagination logic. So, you just provide the template that embedded pagination functionality from this mixin.

And this is exactly what happens with Appup TablePagination component. If you look at the [source code](https://www.vuetable.com/guide/pagination.html) of the component, you'll see that it only contains the template. All the methods and props are in fact inside the mixin.

For Appup TablePaginationDropdown, it only uses the mixin, but also adds its own prop and event.

## **Zero Based Pagination**

In some system, the first page starts at 0 (zero) and the second page starts at 1, and so on. In that case, you can use [first-page](https://www.vuetable.com/guide/pagination.html) prop to set it to 0. So when Appup Table requests the first page from the API endpoint, it will request page 0 instead.

|  |
| --- |
| <appup-table  api-url="..."  :first-page="0"  ></appup-table> |

## **Binding Pagination Component to Appup Table**

In order for the Appup Table's compatible pagination component to work, you'll need to bind it with Appup Table first. This could be done in 3 easy steps:

* On Appup Table, use v-on to listen to appup-table:pagination-data event and specify the binding handler function.

|  |
| --- |
| <template>  <div>  <appup-table ref="appup-table"  // This tells Appup Table that when the paginaiton data is available,  // call `onPaginationData` method.  @appup-table:pagination-data="onPaginationData"  ></appup-table>  <appup-table-pagination ref="pagination"></appup-table-pagination>  </div> </tempalte> |

* On Appup TablePagination, use v-on to listen to appup-table-pagination:change-page event and specify the binding handler function.

|  |
| --- |
| <template>  <div>  <appup-table ref="appup-table"  //...  ></appup-table>  <appup-table-pagination ref="pagination"  // This tells Appup TablePagination component that when there is  // a request to change the page, call `onChangePage` method.  @appup-table-pagination:change-page="onChangePage"  ></appup-table-pagination>  </div> </tempalte> |

* Define both of the binding handler functions in the component.

|  |
| --- |
| //... methods: {  //...  // when the pagination data is available, set it to pagination component  onPaginationData (paginationData) {  this.$refs.pagination.setPaginationData(paginationData)  },  // when the user click something that causes the page to change,  // call "changePage" method in Appup Table, so that that page will be  // requested from the API endpoint.  onChangePage (page) {  this.$refs.appup-table.changePage(page)  } } |

# **Multi-Row Header**

You can customize Appup Table to display more than one table header row via

* tableHeader scoped slot, or
* header-rows prop using header component

By default, Appup Table uses Appup TableRowHeader component to render table header. It will iterate through each field to display the provided title.

## **Using tableHeader scoped slot**

When you define a scoped slot named tableHeader, this will completely replace the header auto-generated by Appup Table.

Appup Table will also pass the normalized fields definition to the tableHeader slot. You will have complete control on how the table header row is rendered, you can even write a component to render your own table header. It's your choice.

|  |
| --- |
| <appup-table>  <template slot="tableHeader" slot-scope="{ fields }">  <!-- define your own table row -->  <tr>  <th>...</th>  <th>...</th>  </tr>  </template> </appup-table> |

**However, please note that you will also lose the ability to interact with column header (e.g. click for sorting).**

Luckily, Appup Table uses Appup TableRowHeader component to generated the default table header row that also handle user interaction. That means you can also use it in your tableHeader slot as well.

|  |
| --- |
| <template>  <appup-table ref="appup-table"  api-url="..."  :fields="..."  >  <template slot="tableHeader">  <appup-table-row-header></appup-table-row-header>  </template>  </appup-table> </template>  <script> import Appup Table from 'appup-table-2/src/components/Appup Table.vue' import Appup TableRowHeader from 'appup-table-2/src/components/Appup TableRowHeader.vue'  export default {  components: {  Appup Table,  Appup TableRowHeader  },  //... } </script> |

## **Using header-rows prop**

Another way to render multi-row header is to use header-rows prop.

By default, it is an array containing the only value Appup TableRowHeader. That means it will delegate the rendering of table header row to Appup TableRowHeader component.

But since it is an array, you can write your own component and add it to the array of header-rowsprop. Appup Table will use the component in the array to render the table header row one by one in sequence specified.

The component can also be as simple as this.

|  |
| --- |
| <template>  <tr>  <th colspan="2">Group AAA</th>  <th colspan="4">Group BBB</th>  </tr> </template> |

## 

## **Communicating with Appup Table**

If you need to communicate with the parent Appup Table, you can either create a computed property and return this.$parent, which is what Appup TableRowHeader does.

|  |
| --- |
| export default {  //...  computed: {  appup-table() {  return this.$parent  }  } } |

Or, you can emit an appup-table:header-event event containing two parameters as followed:

* type String
* A named string used to distinguish the origin of the event, e.g. the name of your header row component.
* payload Object
* An object containing anything that is necessary or relevant to the outside component.

Appup Table will relay this event out from itself, so that the outside component can capture and used it.

|  |
| --- |
| <template>  <appup-table  //..  @appup-table:header-event="onHeaderEvent"  ></appup-table> </template>  <script> export default {  //...  methods: {  //...  onHeaderEvent (type, payload) {  // if (type === 'my-header-row') {  // .. do something ..  // }  }  } } </script> |