

Softsquare

Lightning DataTable Dev



Soft
Square

Apps as a Service

User Guide
October 18, 2018

CONTENTS

Overview.....	3
Features.....	3
Attributes	3
header	3
dataRows.....	6
config (optional).....	6
title (optional).....	8
titleIcon (optional)	8
titleIconURL (optional)	8
selectedRows (optional)	8
noRecordsMessage (optional).....	8
Events.....	8
Methods	10
initialize(object:initConfig)	10
addRow(object:rowData)	11
updateRow(number:rowIndex, object:rowData)	11
updateRowByKey(string:fieldName, string:fieldValue, object:rowData)	11
deleteRow(number:rowIndex)	11
deleteRowByKey(string:fieldName, string:fieldValue, object:rowData)	11
rerenderRows()	11
exportAs(object:fileProperty)	11
Customizing Labels	12
Modifying Labels / Adding Translation support	12
task management app	15
Data Model	16
Showing Data in Lightning DataTable	16
Adding and Editing a row	20

Deleting a Row.....	28
Mass Updating Rows	30
Running the component:	32
Change Logs:.....	36

OVERVIEW

Lightning DataTable Dev is a Lightning Component intended for Lightning developers to let them display data in a table/grid with searching, sorting and pagination features. In order to leverage this table, the developer has to prepare the JSON data and provide that as input for this component. Please see detailed example below to understand how this works.

Features

1. Searching
2. Sorting
3. Pagination
4. Limiting records per Page
5. Selectable rows
6. Table Level Actions
7. Row Level Actions

Attributes

`header`

Holds information of the columns to be displayed in the table. It accepts array of objects. Each object contains information like column name, type of the data and what property to look into. Columns are rendered based on the order of the objects.

Structure of the Object

- `label`: Label to be displayed in the column header.
- `name`: The property name in the row object to be displayed for the particular column.
- `width`: A fixed width to be used by the column. The unit is in pixels.
- `type`: Type of the data displayed in the column.
 - `string` (default value)
 - `reference`
 - `url`
 - `date`
 - `datetime`
 - `currency`
 - `checkbox`
 - `phone`
 - `number`
 - `email`
 - `percent`
 - `image`

- o richtext

- value: This is used only if the type is reference or url.
- displayLabel: This is used for displaying label for URL .
- format: Format in which number, currency, percent, date and datetime value to be displayed.

NOTE : Format is based on the formatter pattern available in <https://docs.oracle.com/javase/7/docs/api/java/text/DecimalFormat.html>. Please check the grouping limitation before using the format option for number, currency and percent field.

- class: CSS class to be applied to the column.
- target: target value to be applied to type="url" or "reference" column. Default value is _blank for "url" Column.
- resizable: Setting to true enables column to be resizable. Default value is false.
- truncate: Allows you to truncate text content initially with show more / less capability.
Accepts object which requires passing below properties:

characterLength : No.of characters to be displayed initially .

showMoreLabel : Link text to be shown which on click displays the full text content

showLessLabel : Link Text to be shown which on click shows the truncated text content.

- searchable: To make a column searchable in the table. Acceptable values: true, false. Default value is true.
- visible: To control the column visibility in the table. Acceptable values: true, false. Default value is true.
- sortable: To make a column sortable in table. Acceptable values: true, false. Default value is true.
- exportable: To make a column exportable in table. Acceptable values: true, false. Default value is true.
- enableCellClickEvt: To fire event on a column's cell click set it to true. Acceptable values: true, false. Default value is false.

NOTE : If enableCellClickEvt is set to true. Redirection will not happen for URL columns.

Sample data for header attribute

```
[
  {
    'label': 'Contact Name',
    'name': 'Name',
    'type': 'reference',
    'value': 'Id',
    'width': 200,
    'class': 'boldCls',
    'resizeable': true
  },
  {
    'label': 'Birthdate',
    'name': 'Birthdate',
    'type': 'date',
    'sortable': false
  },
  {
    'label': 'Personal Phone',
    'name': 'Personal_Phone__c',
    'type': 'phone',
    'visible': false
  },
  {
    'label': 'Mail',
    'name': 'Email',
    'type': 'string',
    'searchable': false,
    'enableCellClickEvt': true
  },
  {
    'label': 'Description',
    'name': 'Description__c',
    'type': 'richtext',
    'type': {
      'characterLength': 50,
      'showMoreLabel': 'more..',
      'showLessLabel': 'less..'
    }
  },
  {
    'label': 'CreatedDate',
    'name': 'CreatedDate',
    'type': 'datetime',
    'format': 'DD/MM/YYYY'
  }
];
```

dataRows

Holds the data to be displayed in the table. It accepts array of objects. The properties of the object should match with respective header attribute's name property of each column.

Sample data for header attribute

```
[
  {
    Name: 'Andy Young',
    BirthDate: '',
    HomePhone: '(785) 241-6200',
    Email: 'a_young@dickenson.com',
    Id: '00336000003vZntAAE'
  },
  {
    Name: 'Arthur Song',
    BirthDate: '1947-09-13',
    HomePhone: '',
    Email: 'asong@uog.com',
    Id: '00336000003vZo1AAE'
  },
  {
    Name: 'Lauren Boyle',
    BirthDate: '1956-06-30',
    HomePhone: '(212) 843-2200',
    Email: 'lboyle@uog.com',
    Id: '00336000003vZnxAAE'
  }
  ... More Data
]
```

config (optional)

Advanced features can be enabled by config attribute like row actions, global actions and mass selecting rows.

Structure of Configuration Object

Configuration object has seven properties:

- `massSelect(optional)`: To display checkboxes on each row for select all purpose. It accepts either `true` or `false`. Default value is `false`.
- `showRowSelect(optional)`: Accepts a function/boolean which will be executed on checkbox rendering for each row. If it returns `true`, then the action will be displayed else it will be hidden. By default, value is `true`. Please refer the Task Management App code snippet for the example.

- `rowAction(optional)`: Holds the information of the row level actions to be in each row of the table. It accepts array of action objects.
- `globalAction(optional)`: Holds the information of the global level actions to be added to the top-right corner of the table. It accepts array of action objects.
- `searchBox(optional)`: Enable/Disable search box for searching data in the table. Default value is `true`.
- `searchByColumn(optional)`: Enable/Disable searching on a particular column. Default value is `false`.
- `paginate(optional)`: Enable/Disable the pagination controls in the component. Default value is `true`.
- `itemMenuPosition(optional)`: Position the item Menu dropdown above / below the table. Accepted values are `top`, `bottom`. Default value is `top`.
- `rowActionPosition(optional)`: To positioning the row action column (button/URL/image) to either left or right of the table. Default value is `left`.
- `rowActionWidth(optional)`: Set the width of the row level action column.
- `rowClass(optional)`: Accepts a function/string which will be evaluated against each row and adds to the `tr`.

Structure of Action Object

Action object contains key information like label of the action, action type (button or link or icon).

It has following properties.

- `label`: Text which is displayed as label of the action.
NOTE : if you wanted to display only the icon for menu action, then label should be empty.
- `id`: Unique id of the action which is used to identify a specific action from other actions.
- `class(optional)`: CSS class names to be applied.
- `type`: Type of the action. Accepted values: `button`, `url`, `menu`, `image` (available for `rowAction` only).
- `src`: Url of the icon to be displayed in the action column.
- `visible(optional)`: Accepts a function/boolean which will be executed on action rendering for each row. If it returns/set to `true` action will be displayed else it will be hidden. By default, `visible` is `true`.

- `menuIconClass (optional)`: CSS class name to be icon displayed. Only available for rowAction with type="menu".
- `menuOptions (optional)`: Accepts array of objects which should contain following properties:
 - `id`: Unique id of the menu action.
 - `label`: Text to be displayed for the action.
 - `class (optional)`: CSS class to be applied to the action.
 - `visible (optional)`: Same as above.

NOTE: `menuIconClass`, `menuOptions` is available only on row action.

`title (optional)`

Sets title for the DataTable.

`titleIcon (optional)`

Sets header icon DataTable. Only icon available in force:slds is supported.

`titleIconURL (optional)`

Sets header icon for the DataTable based on the provided url.

`selectedRows (optional)`

Contains all the selected rows of the table in an array. Works only when `massSelect` is enabled.

`noRecordsMessage (optional)`

Display the message in table when no rows are shown.

Events

1.Action Click Event:

The DataTable component contains `ldt:clickedDtAction` event that fires on both row actions as well as global actions.

Structure of Event Object

- `actionId`: contains id of the action.
- `row`: row object.
- `index`: index/position of the row.

Row-Level Action

When `ldt:clickedDtAction` event fires due to a row level action, it contains data with following attributes:

- `actionId`: unique id of the action object passed as part of row level action configuration
- `row`: row object from where the click is originated.
- `index`: index of the row object.

Global Action

When `ldt:clickedDtAction` event fires due to a global action, it contains data with following attribute:

- `actionId`: unique id of the action object passed as part of global action configuration.

Please note that event handler for the above event should be named as `dtActionClick`. If not, the action will not be handled.

2.Cell Click Event:

The DataTable component contains `ldt:enableCellClickEvt` event that fires on when the cell is clicked.

Structure of Event Object:

- `fieldApiName`: contains field API Name.
- `value`: value of cell that was clicked.
- `row`: row object.
- `index`: index/position of the row.

Please note that event handler for the above event should be named as `enableCellClickEvt`. If not, the action will not be handled.

Methods

`initialize(object:initConfig)`

To initialize the table with provided data. It accepts one argument.

Structure of `initConfig`:

`order` : It accepts an array which contains two elements to specify the column to sort and its order respectively. Pass empty array to disable the default sorting.

Format:

```
initialize({  
    "order":[number:indexOfTheColumn, string:sortOrder],  
    "itemMenu":number[],  
    "itemsPerPage":number  
})
```

`indexOfTheColumn`: Index of the column to sort initially.

`sortOrder` : Order in which column to be sorted. Accepted values are `asc` and `desc`. Default value is `asc`.

`itemMenu`: To override default items per page values pass an array of integers. Values above 50 will be discarded.

itemsPerPage: Number of records to be displayed per page.

`addRow(object:rowData)`

To add a new row in the table. New object has to be passed as the parameter.

`updateRow(number:rowIndex, object:rowData)`

Update an already existing row in the table. The index position and the updated object is passed as the parameter.

`updateRowByKey(string:fieldName, string:fieldValue, object:rowData)`

Update an already existing row in the table. Pass the field / key and its value along with update object.

`deleteRow(number:rowIndex)`

To delete a given row in the table.

`deleteRowByKey(string:fieldName, string:fieldValue, object:rowData)`

To delete a given row in the table. Pass the field / key and its value along with update object.

`rerenderRows()`

To re-render the table with the updated data. It can be used if the source data for the `dataRows` attribute is modified.

`exportAs(object:fileProperty)`

Use this method to export the table data as CSV.

Structure of `fileProperty`:

`filename`: Name of the file.

`filetype`: Type of file. As of now, it supports only `csv` value.

`type`: To define which type of records to be exported

Supported values: `All_Rows, Filtered_Rows`

`useValuePropForReference`: Set to `true` if you want to export value property data of `reference` columns. Supported values: `true, false`

Customizing Labels

You can override the labels displayed in the DataTable component by modifying the appropriate custom label which is exposed as part of the package using *Translation Workbench*.

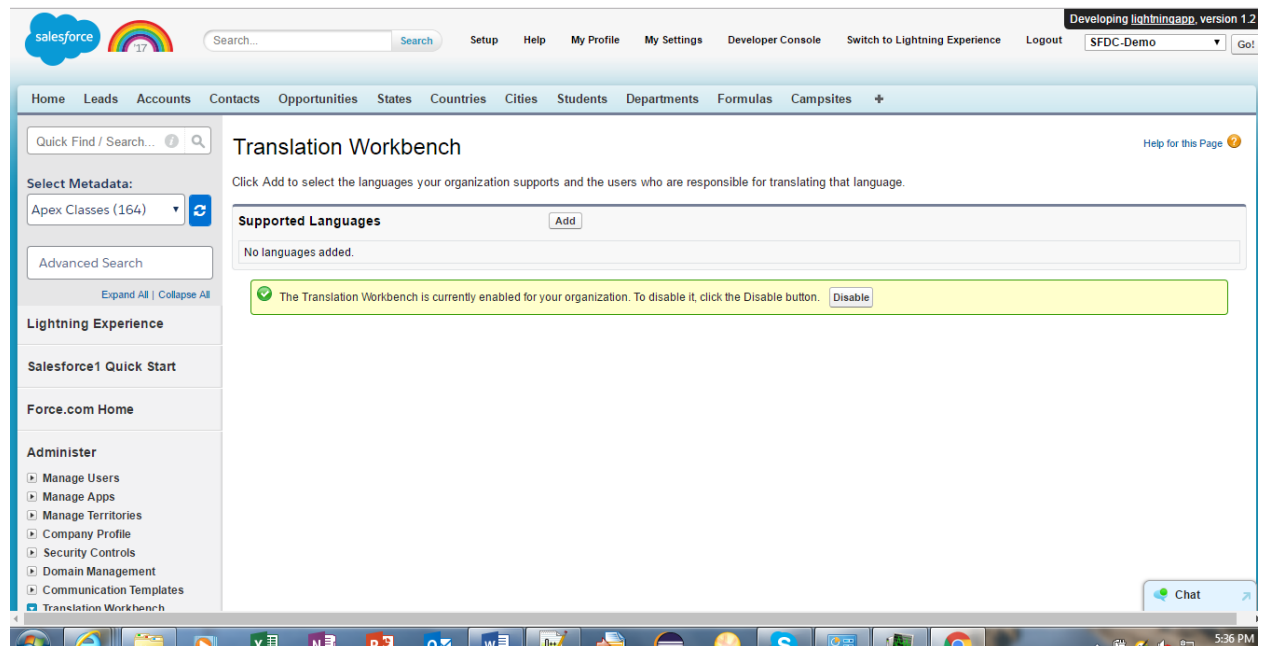
Follow are the list of custom labels which are available:

- 1) Action_Column - Action Column label
- 2) Items_Per_Page - Items Per Page menu label
- 3) Page_Next - Pagination's Next button label
- 4) Page_Prev - Pagination's Previous button label
- 5) Pagination_Info - Pagination information display Label
- 6) Search - Search box label
- 7) Search_By_Column - Search by Column menu label
- 8) Select_All - Select All Label

Modifying Labels / Adding Translation support

To customize the default labels or add translation support you need to enable the Translation Workbench in the org. Please visit the

- 1) To enable Translation go to *Setup -> Translation Workbench -> Translation Settings* and click on enable button.



- 2) Now Add languages and assign it to the User for whom the translation will be carried out. After adding the Languages it will look like below:

The screenshot shows the Salesforce Translation Workbench interface. The top navigation bar includes links for Setup, Help, My Profile, My Settings, Developer Console, Switch to Lightning Experience, Logout, and a dropdown for the current user (SFDC-Demo). The left sidebar contains a search bar and a list of metadata types, with 'Apex Classes (164)' selected. The main content area is titled 'Translation Workbench' and includes a section for 'Supported Languages'. This section has a table with columns for Action, Language, Active, and Translator(s). The table lists Spanish and English, both with 'Praveen P' as the translator. Below the table, a green message box states: 'The Translation Workbench is currently enabled for your organization. To disable it, click the Disable button.' A 'Disable' button is visible next to the message.

3) Now, go to Setup -> Custom Labels to view the Datatable custom labels.

The screenshot shows the Salesforce Custom Labels interface. The top navigation bar is the same as the previous screenshot. The left sidebar shows the 'Apex Classes (164)' metadata type selected. The main content area is titled 'All' and includes a section for 'Custom Labels'. This section has a table with columns for Action, Name, Categories, Short Description, Value, and Language. The table lists several custom labels, including 'Action Column', 'Items Per Page', 'Page Next', 'Page Prev', 'Pagination Info', 'Search', 'Search By Column', and 'Select All'. A 'New Custom Label' button is visible above the table. Below the table, a message box states: 'Custom Labels are custom text values, up to 1,000 characters, that can be accessed from Apex Classes or Visualforce Pages. If Translation Workbench has been enabled for your organization, these labels can then be translated into any of the languages salesforce.com supports. This allows developers to create true multilingual apps by presenting information to users - for example, help text or error messages - in their native language. You can create up to 5,000 custom labels.'

4) Click on the custom label(say "Search") you would want to translate and then click on New Local Translations / Overrides button.

The screenshot shows the Salesforce Lightning Experience interface. The top navigation bar includes the Salesforce logo, a search bar, and links for Setup, Help, My Profile, My Settings, Developer Console, Switch to Lightning Experience, Logout, and a dropdown for 'SFDC-Demo'. The main navigation bar lists various objects: Home, Leads, Accounts, Contacts, Opportunities, States, Countries, Cities, Students, Departments, Formulas, and Campsites. The left sidebar contains a 'Quick Find / Search...' bar, a 'Select Metadata:' dropdown set to 'Apex Classes (164)', an 'Advanced Search' button, and sections for 'Lightning Experience', 'Salesforce1 Quick Start', 'Force.com Home', and 'Administer' (with sub-links like Manage Users, Manage Apps, etc.).

The main content area is titled 'Custom Label Search (Managed)'. It includes a 'Back to List: Custom Labels' link and a yellow informational banner stating: 'This Custom Label is managed, meaning that you may only edit certain attributes. [Display More Information](#)'. Below this is the 'Custom Label Detail' section with an 'Edit' button. The details are as follows:

Short Description	Search Box Label	Name	Search
Language	English	Protected Component	☐
Namespace Prefix	Idt		
Installed Package	Lightning DataTable Dev		
Categories	Search		
Value	Search		
Created By	Praveen P. 24/4/2017 9:50 AM	Modified By	Praveen P. 24/4/2017 9:50 AM

Below the details is the 'Local Translations / Overrides' section with a 'New Local Translations / Overrides' button. A yellow banner states: 'These translations were created by administrators in your company. Local translations override packaged ones.' Below this, it says 'No records to display'. The 'Packaged Translations' section is partially visible at the bottom.

- 5) Select the Language and add the translation text for the Master Text and click on Save.

The screenshot shows the 'New Translation' page in Salesforce Lightning Experience. The top navigation bar and left sidebar are identical to the previous screenshot. The main content area is titled 'New Translation' and includes a 'Help for this Page' link. Below the title is the 'Translation Edit' section with 'Save' and 'Cancel' buttons. The 'Master Label Information' section shows 'Master Label' as 'Search' and 'Master Label Language' as 'English'. The 'Translation Information' section has a 'Language' dropdown set to 'Spanish' and a 'Translation Text' input field containing 'buscador'. At the bottom of this section are 'Save' and 'Cancel' buttons. A yellow banner at the bottom right of the page says 'Chat'.

TASK MANAGEMENT APP

We are going to build a sample Task Management app step-by-step to see how Lightning DataTable component can be leveraged for this. Here we will show how to add, edit and delete tasks under a project. Also, we will show how to Mass update status of the selected tasks.

Optionally, users can install an unmanaged package using the link below to get this sample Task Management app in their org.

Production - <http://rebrand.ly/TaskManagementApp>

Sandbox - <http://rebrand.ly/TaskManagementAppSandbox>

Follow the steps below to view the Task Management application.

- Navigate to the Lightning Datatable Demo app.
- Click Task Management App tab.

Follow the steps below to view the Task Management App in Salesforce1.

- Go to Setup.
- Find Salesforce1 Navigation under Mobile Administration.
- Move Task Management App to the selected list.
- Click Save.
- Now, go to Salesforce1 and select Task Management App in the navigation drawer.

If you would like to build this app step-by-step, please follow the instructions below.

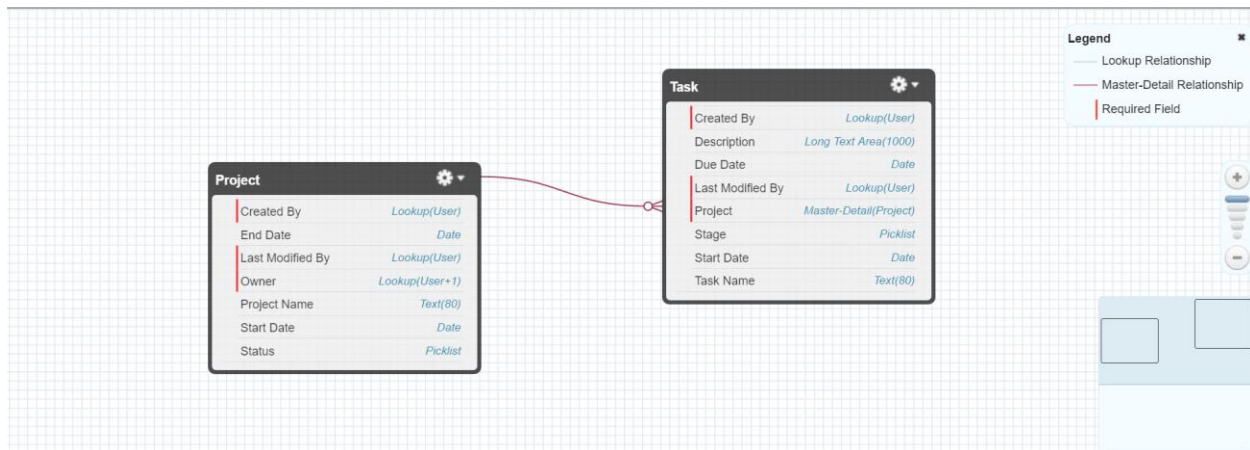
Project Tasks					Add Task	Complete Task
Items Per Page		Search by Column		Search		
10		--None--				
☐ ACTION	TASK NAME ↑	START DATE	DUE DATE	STAGE		
☐ Edit Del	Task 1	13-Apr-2017	14-Apr-2017	In Progress		
☐ Edit Del	Task 2	27-Apr-2017	22-Apr-2017	Completed		
☐ Edit Del	Task 3	20-Apr-2017	21-Apr-2017	Completed		
☐ Edit Del	Task 4	29-Apr-2017	14-Apr-2017	Started		
☐ Edit Del	Task 5	13-Apr-2017	13-Apr-2017	In Progress		
Showing 1 to 5 of 5 records					Previous	Next

COMPLETED DATATABLE COMPONENT

Note: Task and Project are custom objects where Project is parent of Task.

Data Model

Create object and fields as shown in schema builder below:



Showing Data in Lightning DataTable

1. Open *Developer Console* -> *File* -> *New* -> *Lightning Component* -> Create a component named **projectTaskList**.
2. Open *Developer Console* -> *File* -> *New* -> *Apex class* -> Create an Apex class named **ProjectTaskController**.
3. Add the following code to `projectTaskList.cmp`

```
<aura:component controller="ProjectTaskController"
    implements="flexipage:availableForRecordHome,force:hasRecordId">
    <aura:attribute name="recordId" type="String" access="public"/>
    <aura:attribute name="projectId" type="String" access="public"
        default="{!v.recordId}"/>
    <aura:attribute name="projectTasks" type="Task__c[]" access="public"/>
    <aura:attribute name="taskColumns" type="List" access="public"/>
    <aura:attribute name="taskTableConfig" type="Map" access="public"/>
    <aura:handler name="init" value="{!this}" action="{!c.fetchTasks}" />
    <section class="project-task-list slds-p-top--x-small">
        <div class="task-list slds-p-top--medium">
            <ldt:datatableDev title="Project Tasks" aura:id="taskTable"
                dataRows="{!v.projectTasks}" header="{!v.taskColumns}"
                config="{!v.taskTableConfig}"/>
        </div>
    </section>
</aura:component>
```

Explanation

- `<ldt:datatableDev/>` - Lightning DataTable component which requires `aura:id`, `dataRows` and `header` attributes to be defined for it to render the data in the table.

- `projectTasks` - Holds the task records to be displayed in the Lightning DataTable. It accepts array of objects.
- `taskColumns` - Holds the column information for the Lightning DataTable. It accepts array of objects.
- `taskTableConfig` - Holds the configuration details of the task table to be displayed.
- `init` - component's init event handler which fetches the task records.
- When the `projectTaskList.cmp` loads, its `init` handler fires and calls the `fetchTasks` method in the `projectTaskListController.js` to get the task records to be displayed

4. Add the following code to `projectTaskListController.js`.

```
({
  fetchTasks : function(cmp, event, helper) {
    var action = cmp.get("c.getTasksOfProject");

    //Column data for the table
    var taskColumns = [
      {
        'label': 'Task Name',
        'name': 'Name',
        'type': 'reference',
        'value': 'Id',
        'resizeable': true
      },
      {
        'label': 'Start Date',
        'name': 'Start_Date__c',
        'type': 'date'
      },
      {
        'label': 'Due Date',
        'name': 'Due_Date__c',
        'type': 'date'
      },
      {
        'label': 'Stage',
        'name': 'Stage__c',
        'type': 'string'
      }
    ];

    //Configuration data for the table to enable actions in the table
    var taskTableConfig = {
      "massSelect": true,
      "showRowSelect": function (row) { return row.Stage__c != 'Closed' },
      "globalAction": [
        {
```

```

        "label": "Add Task",
        "type": "button",
        "id": "addtask",
        "class": "slds-button slds-button--neutral"
    },
    {
        "label": "Complete Task",
        "type": "button",
        "id": "completetask",
        "class": "slds-button slds-button--neutral"
    }
],
"rowAction": [
    {
        "label": "Edit",
        "type": "url",
        "id": "edittask"
    },
    {
        "label": "Del",
        "type": "url",
        "id": "deltask"
    },
    {
        "type": "menu",
        "id": "actions",
        "visible": function(task){
            return task.Stage__c != "Completed"
        },
        "menuOptions": [{
            "label": "Complete",
            "id": "completeTask"
        }]
    }
],
"rowActionPosition": 'left'

};

if(cmp.get("v.projectId")){
    action.setParams({
        "projectId": cmp.get("v.projectId")
    });

    action.setCallback(this, function(resp){
        var state = resp.getState();
        if(cmp.isValid() && state === 'SUCCESS'){

            //pass the records to be displayed
            cmp.set("v.projectTasks", resp.getReturnValue());

            //pass the column information

```

```

        cmp.set("v.taskColumns",taskColumns);

        //pass the configuration of task table
        cmp.set("v.taskTableConfig",taskTableConfig);

        //initialize the datatable
        cmp.find("taskTable").initialize({
            "order":[0,"desc"]
        });
    }
    else{
        console.log(resp.getError());
    }
});

$A.enqueueAction(action);
}
})

```

Explanation

- The `fetchTasks` method calls the Apex controller method `getTasksOfProject()` to get the task records.
- After successfully fetching records, we are setting the following attributes:
 - `projectTasks`
 - `taskColumns`
 - `taskTableConfig`
 which serves as the input for the Lightning DataTable component.
- Finally call the `initialize()` method of the DataTable component to render the task records.

5. Add the following code to `projectTaskList.css`:

```

.THIS .dateFix{
    width: 95% !important;
}

.THIS .selHeight{
    height: 2.5em !important;
}

```

6. Add the following method to `ProjectTaskController` Apex Controller which queries the task records of a project using the `projectId`.

```

@AuraEnabled
public static List<Task__c> getTasksOfProject(String projectId){

```

```

return [SELECT Id,Name,Start_Date__c,Due_Date__c,Stage__c,Description__c
        FROM Task__c
        WHERE Project__c = :projectId];
}

```

Adding and Editing a row

Let's see how we can add and delete a row in the Lightning DataTable.

1. Add the following attributes to the `projectTaskList.cmp`.

```

<aura:attribute name="selectedTask" type="Task__c" access="public"
    default="{ 'subjectType':'Task__c', 'Stage__c':'', 'Project__c':'',
              'Start_Date__c':'', 'Description__c':'', 'Due_Date__c':'' }"/>

<aura:attribute name="taskOpType" type="String" access="private"/>

<aura:handler name="dtActionClick" event="ldt:clickedDtAction"
    action="{!c.tabActionHandler}"/>

```

Explanation

- `selectedTask` - contains the task that is being edited
- `rowIndex` - contains the index of the task record that is being edited
- `taskOpType` - contains either "Add/Edit" value
- `dtActionClick` - name of the component event handler fired by the Lightning Datable when a button/link is clicked.

NOTE: Please note that event handler should be named as `dtActionClick` only. If not, the action will not be handled.

2. Now create a modal component `sldsModal` to be used in the `projectTaskList.cmp`.

The screenshot shows a modal window titled "Edit Task". Inside the modal, there are five input fields arranged in two columns. The left column contains "Task Name" (a text input field with the value "Attend Marketing Cloud Builder ses"), "Description" (a text area), and "Stage" (a dropdown menu currently showing "In Progress"). The right column contains "Start Date" (a date picker showing "Sep 29, 2016") and "Due Date" (a date picker showing "Oct 8, 2016"). At the bottom right of the modal, there are two buttons: "Cancel" and "Ok".

THE MODAL TO EDIT OR ADD TASK

3. Open *Developer Console* -> *File* -> *New* -> *Lightning Component* -> Create a component named **svgIcon**.
4. Add the below markup to `svgIcon.cmp`:

```
<aura:component>
  <aura:attribute name="svgPath"          default="" type="String"
    description="the path for the icon in the static resource,
    this will be use in a SVG use tag" />
  <aura:attribute name="name"            default="" type="String"
    description="Symbol name of icon" />
  <aura:attribute name="class"           default="" type="String"
    description="the class of this SVG tag, can be use for CSS
    purpose" />
  <aura:attribute name="containerClass" default="" type="String"
    description="Container class name for span container of icon"
    />
  <aura:attribute name="category"        default="" type="String"
    description="Category of icon- action, standard, utility
    etc." />
  <aura:attribute name="size"            default="" type="String"
    description="Size of icon-- small, medium, large" />
  <aura:attribute name="assistiveText" default="" type="String"
    description="Description name of icon" />
  <span aura:id="container" class="{!v.containerClass}">
  <span aura:id="assistiveText" class="slds-assistive-
    text">{!v.assistiveText}</span>
  </span>
</aura:component>
```

5. Add the below code to `svgIconHelper.js`:

```
({
  renderIcon: function(component) {
    var prefix = "slds-";
    var svgnss = "http://www.w3.org/2000/svg";
    var xlinkns = "http://www.w3.org/1999/xlink";
    var size = component.get("v.size");
    var name = component.get("v.name");
    var classname = component.get("v.class");
    var containerclass = component.get("v.containerClass");
    var category = component.get("v.category");

    var containerClassName = [
      prefix+"icon_container",
      prefix+"icon-"+category+"-"+name,
      containerclass
    ].join(' ');
    component.set("v.containerClass", containerClassName);

    var svgroot = document.createElementNS(svgnss, "svg");
```

```

    var iconClassName = prefix+"icon "+prefix+"icon--" + size+" "+classname;
    svgroot.setAttribute("aria-hidden", "true");
    svgroot.setAttribute("class", iconClassName);
    svgroot.setAttribute("name", name);

    // Add "href" attribute (using the "xlink" namespace)
    var shape = document.createElementNS(svgns, "use");
    shape.setAttributeNS(xlinkns, "href", component.get("v.svgPath"));
    svgroot.appendChild(shape);

    var container = component.find("container").getElement();
    container.insertBefore(svgroot, container.firstChild);
  }
})

```

6. Add the below code to `svgIconRenderer.js`:

```

({
  render: function(component, helper) {
    // By default, after the component finished loading data/handling events,
    // it will call this render function this.superRender() will call the
    // render function in the parent component.
    var ret = this.superRender();

    // Calls the helper function to append the SVG icon
    helper.renderIcon(component);
    return ret;
  }
})

```

7. Open Developer Console -> File -> New -> Lightning Component -> Create a component named **sldsModal**.

8. Add the below markup to `sldsModal.cmp`:

```

<aura:component access="public">
  <aura:attribute name="header" type="Aura.Component[]" access="public"/>
  <aura:attribute name="content" type="Aura.Component[]" access="public"/>
  <aura:attribute name="footer" type="Aura.Component[]" access="public"/>
  <aura:attribute name="closeAction" type="Aura.Action" access="public"/>

  <aura:method name="open" action="{!c.toggleModal}" />
  <aura:method name="close" action="{!c.toggleModal}" />

  <div class="slds-modal" aura:id="modal" aria-hidden="false" role="dialog">
    <div class="slds-modal__container">
      <div class="slds-modal__header" onclick="{!v.closeAction}">
        <button class="slds-button slds-button--icon-inverse slds-modal__close">
          <c:svgIcon
            svgPath="/resource/ldt__SLDS202/assets/icons/utility-sprite/svg/symbols.svg#close" class="slds-button__icon slds-button__icon--medium"
            containerClass="slds-icon_container"/>
          <span class="slds-assistive-text">Close</span>
        </button>
      </div>
    </div>
  </div>

```

```

        </button>
        <h2 class="slds-text-heading--medium">{!v.header}</h2>
    </div>
    <div class="slds-modal__content slds-p-around--medium">
        {!v.content}
    </div>
    <div class="slds-modal__footer">
        {!v.footer}
    </div>
</div>
</div>

<div class="slds-backdrop" aura:id="modal-backdrop"></div>
</aura:component>

```

9. Add the below code to `sldsModalController.js`:

```

({
    toggleModal : function(cmp, event, helper) {
        $A.util.toggleClass(cmp.find("modal"), 'slds-fade-in-open');
        $A.util.toggleClass(cmp.find("modal-backdrop"), 'slds-backdrop--open');
    }
})

```

10. Add the below modal markup to `projecTaskList.cmp` to show the modal for add and edit task.

```

<!-- Edit Task Modal Begin -->
<c:sldsModal aura:id="taskEditModal" closeAction="{!c.closeTaskModal}">
    <aura:set attribute="header">
        {!v.taskOpType} Task
    </aura:set>
    <aura:set attribute="content">
        <div class="task-inputs">
            <div class="slds-grid slds-wrap">
                <div class="slds-p-horizontal--small slds-size--1-of-2">
                    <ui:inputText label="Task Name" labelClass="slds-form-
                        element__label" class="slds-input"
                        value="{!v.selectedTask.Name}" />
                </div>
                <div class="slds-p-horizontal--small slds-size--1-of-2">
                    <ui:inputDate label="Start Date" labelClass="slds-form-
                        element__label" class="slds-input dateFix"
                        value="{!v.selectedTask.Start_Date__c}"
                        displayDatePicker="true"/>
                </div>
                <div class="slds-p-horizontal--small slds-size--1-of-2">
                    <ui:inputTextArea label="Description" labelClass="slds-form-
                        element__label" class="slds-textarea"
                        value="{!v.selectedTask.Description__c}" />
                </div>
                <div class="slds-p-horizontal--small slds-size--1-of-2">
                    <ui:inputDate label="End Date" labelClass="slds-form-

```

```

        element__label" class="slds-input dateFix"
        value="{!v.selectedTask.Due_Date__c}"
        displayDatePicker="true"/>
    </div>
    <div class="slds-p-horizontal--small slds-size--1-of-2">
        <ui:inputSelect label="Stage" labelClass="slds-form-
        element__label" class="slds-select selHeight"
        value="{!v.selectedTask.Stage__c}">
            <ui:inputSelectOption text="" label="--None--"/>
            <ui:inputSelectOption text="In Progress" label="In
            Progress"/>
            <ui:inputSelectOption text="Completed" label="Completed
            "/>
            <ui:inputSelectOption text="Closed" label="Closed"/>
        </ui:inputSelect>
    </div>
</div>
</div>
</aura:set>
<aura:set attribute="footer">
    <button class="slds-button slds-button--neutral"
    onclick="{!c.closeTaskModal}">Cancel</button>
    <button class="slds-button slds-button--brand"
    onclick="{!c.saveTask}">Ok</button>
</aura:set>
</c:sldsModal>
<!-- Edit Task Modal End -->

```

Explanation

- This modal contains the necessary fields to populate/get the Task Name, Start Date, End Date, Description and Stage values.
- This modal is reused for both add task and edit task.
- There are two buttons in the modal Ok and Cancel.

11. Add the below method to `projectTaskListController.js`:

```

({
    tableActionHandler: function(cmp, event, helper){

        //get the id of the action being fired
        var actionId = event.getParam('actionId');

        if(actionId == 'edittask'){
            //get the row where click happened and its position
            var rowIdx = event.getParam("index");
            var clickedRow = event.getParam('row');

            //store the row and its position will for editing
            cmp.set("v.rowIndex",rowIdx);
            cmp.set("v.selectedTask",clickedRow);

            //set the type of task operation being done
            cmp.set("v.taskOpType",'Edit');
        }
    }
})

```

```

        //Now, Lets open the task modal
        cmp.find("taskEditModal").open();
    }
    else if(actionId == 'addtask'){
        //set the type of task operation being done
        cmp.set("v.taskOpType", 'Add');

        //Now, Lets open the task modal
        cmp.find("taskEditModal").open();
    }
    else if(actionId == 'completeTask'){
        //get the row where click happened and its position
        var rowIdx = event.getParam("index");
        var clickedRow = event.getParam('row');

        //Complete the task
        helper.completeTask(cmp, clickedRow, rowIdx);
    }
},
//Inserts/Updates Task record when `Ok` button is clicked in the task modal
saveTask : function(cmp,event,helper){
    var action = cmp.get("c.upsertTask");
    var task = cmp.get("v.selectedTask");
    var selectedProjId = cmp.get("v.projectId");

    if(selectedProjId){

        //set the Project Lookup field of the task
        task.Project__c = selectedProjId;

        action.setParams({
            "taskRec":task
        });

        action.setCallback(this,function(resp){
            var state = resp.getState();
            if(cmp.isValid() && state === 'SUCCESS'){

                //if operation is add, then add the task row to the table
                if(cmp.get("v.taskOpType") == 'Add'){
                    cmp.find("taskTable").addRow(resp.getReturnValue());
                }
                else{
                    var rowIdx = cmp.get("v.rowIndex");

                    // if operation is edit, then update the row in the table
                    cmp.find("taskTable").updateRow(rowIdx,task);
                }

                //Close the task modal
                helper.closeTaskModal(cmp);
            }
            else{
                console.log(resp.getError());
            }
        });
    }
}

```

```

        });
    }
    $A.enqueueAction(action);
}
}
})

```

12. Add the below method to `projectTaskListHelper.js`:

```

({
    completeTask : function(cmp, task, index){
        //Hide the task modal once editing is done
        var action = cmp.get("c.upsertTask");
        var selectedProjId = cmp.get("v.projectId");

        if(selectedProjId){

            //set the Project Lookup field of the task
            task.Project__c = selectedProjId;
            task.Stage__c = "Completed";

            action.setParams({
                "taskRec":task
            });

            action.setCallback(this,function(resp){
                var state = resp.getState();
                if(cmp.isValid() && state === 'SUCCESS'){

                    // if operation is edit, then update the row in the table
                    cmp.find("taskTable").updateRow(index,task);
                }
                else{
                    console.log(resp.getError());
                }
            });

            $A.enqueueAction(action);
        }
    }
})

```

Explanation

- When Add Task button is clicked, a component event `ldt:clickedDtAction` is fired which contains information of the action.
 - `actionId="addtask"`.

- The handler `dtActionClick` for above event calls the `tableActionHandler` method and add task modal is shown by calling the `open()` of the `sldsModal` component.
- Once Ok button is clicked, the task details collected from the modal is saved by calling the `upsertTask` method in the apex controller and the task record is returned back to the client.
- Now the newly created task record is added to the table, by passing the task object to the `addRow()` method of the `DataTable` component.
- Clicking Edit link on any row, would fire a component event `ldt:clickedDtAction` which contains information of the action.
 - `actionId="edittask"`,
 - `row = clicked task row`
 - `index = position of the row`.
- The handler `dtActionClick` for above event calls the `tableActionHandler` method. The `index` and the `clicked task object` are stored and edit modal is shown.
- Same steps are performed for saving the edited task. Only change is to pass the updated `task object` along with its `index` to the `updateRow()` method for rendering the row with new values.

13. Add the below method to `projectTaskListHelper.js`:

```
({
  closeTaskModal : function(cmp){
    //Hide the task modal once editing is done
    cmp.find("taskEditModal").close();

    //Reset the rowIndex
    cmp.set("v.rowIndex",-1);

    //Reset the selectedTask
    cmp.set("v.selectedTask",{ 'subjectType':'Task__c', 'Stage__c':'', 'Name':'', 'Project__c':'', 'Start_Date__c':'', 'Description__c':'', 'Due_Date__c':'' });
  },
})
```

14. Add the below method to `ProjectTaskController` Apex Controller

```
@AuraEnabled
public static Task__c upsertTask(Task__c taskRec){
  // if Id is present, then update task else insert task
  upsert taskRec;
  return taskRec;
}
```

Deleting a Row

1. Update `projectTaskListController.js` to include delete logic on the `tableActionHandler` method as follows.

```
({
  tableActionHandler : function(cmp,event,helper){

    //get the id of the action being fired
    var actionId = event.getParam('actionId');

    if(actionId == 'edittask'){
      //get the row where click happened and its position
      var rowIdx = event.getParam("index");
      var clickedRow = event.getParam('row');

      //store the row and its position will for editing
      cmp.set("v.rowIndex",rowIdx);
      cmp.set("v.selectedTask",clickedRow);

      //set the type of task operation being done
      cmp.set("v.taskOpType",'Edit');

      //Now, Lets open the task modal
      cmp.find("taskEditModal").open();
    }
    else if(actionId == 'addtask'){
      //set the type of task operation being done
      cmp.set("v.taskOpType",'Add');

      //Now, Lets open the task modal
      cmp.find("taskEditModal").open();
    }
    else if(actionId == 'deltask'){
      //get the row where click happened and its position
      var rowIdx = event.getParam("index");
      var clickedRow = event.getParam('row');

      //Call the deleteTask method in the helper
      helper.deleteTask(cmp,clickedRow.Id,rowIdx);
    },
    else if(actionId == 'completeTask'){
      //get the row where click happened and its position
      var rowIdx = event.getParam("index");
      var clickedRow = event.getParam('row');

      //Complete the task
      helper.completeTask(cmp, clickedRow, rowIdx);
    }
  }
})
```

Explanation

- When Del link is clicked on any row, a component event `ldt:clickedDtAction` is fired which contains information of the action.
 - `actionId="deltask"`.
 - `row = clicked task row`
 - `index = position of the row`.
- The event is handled by `tableActionHandler` method.
- Then `deleteTask` method in the helper is called which in turn calls the apex controller method `deleteTask` to delete the task record.
- After successfully deleting the record, the task row is removed from the table by passing the index of the row to the `deleteRow()` method.

2. Add the below code to `projectTaskListHelper.js`:

```
({
  //deletes the task record
  deleteTask : function(cmp,taskId,rowIdx) {
    var action = cmp.get("c.deleteTask");

    action.setParams({
      "taskId":taskId
    });

    action.setCallback(this,function(resp){
      var state = resp.getState();

      //if SUCCESS, delete the task row from the table
      if(cmp.isValid() && state === 'SUCCESS'){
        cmp.find("taskTable").deleteRow(rowIdx);
      }
      else{
        console.log(resp.getError());
      }
    });

    $A.enqueueAction(action);
  }
})
```

3. Add the below method to `ProjectTaskController` Apex Controller:

```
@AuraEnabled
public static void deleteTask(String taskId){
  delete new Task__c(id=taskId);
}
```

Mass Updating Rows

Now, let's mass select rows and update a field on each row. In this example, we would update the Status of selected Tasks to "Completed" using the "Complete Task" button

Project Tasks

Add TaskComplete Task

Items Per Page10

Search by Column--None--

Search

☐	ACTION	TASK NAME ↑	START DATE	DUE DATE	STAGE	
☒	Edit Del	Task 1	13-Apr-2017	14-Apr-2017	In Progress	▼
☐	Edit Del	Task 2	27-Apr-2017	22-Apr-2017	Completed	
☐	Edit Del	Task 3	20-Apr-2017	21-Apr-2017	Completed	
☒	Edit Del	Task 4	29-Apr-2017	14-Apr-2017	Started	▼
☐	Edit Del	Task 5	13-Apr-2017	13-Apr-2017	In Progress	▼

Showing 1 to 5 of 5 records

PreviousNext

MASS SELECTING ROWS

1. Update `projectTaskListController.js` to include logic for mass edit on the `tableActionHandler` method as follows.

```
{
  tableActionHandler: function(cmp,event,helper){
    //get the id of the action being fired
    var actionId = event.getParam('actionId');

    if(actionId == 'edittask'){
      //get the row where click happened and its position
      var rowIdx = event.getParam("index");
      var clickedRow = event.getParam('row');

      //store the row and its position will for editing
      cmp.set("v.rowIndex",rowIdx);
      cmp.set("v.selectedTask",clickedRow);

      //set the type of task operation being done
      cmp.set("v.taskOpType",'Edit');

      //Now, Lets open the task modal
      cmp.find("taskEditModal").open();
    }
    else if(actionId == 'addtask'){
      //set the type of task operation being done
      cmp.set("v.taskOpType",'Add');

      //Now, Lets open the task modal
    }
  }
}
```

```

        cmp.find("taskEditModal").open();
    }
    else if(actionId == 'deltask'){
        //get the row where click happened and its position
        var rowIdx = event.getParam("index");
        var clickedRow = event.getParam('row');

        //Call the deleteTask method in the helper
        helper.deleteTask(cmp,clickedRow.Id,rowIdx);
    }
    else if(actionId == 'completeTask'){
        //get the row where click happened and its position
        var rowIdx = event.getParam("index");
        var clickedRow = event.getParam('row');

        //Complete the task
        helper.completeTask(cmp, clickedRow, rowIdx);
    }
}
}}
})

```

Explanation

- When Complete Task button is clicked, a component event of type `ldt:clickedDtAction` is fired which contains information of the action.
 - `actionId="completetask"`.
- The event is handled by calling `tableActionHandler` method which in turn calls the `completeTask` method in the helper.

2. Add the following code to `projectTaskListHelper.js`:

```

({
    completeTasks : function(cmp){

        //Retrieve the selectedTask rows in the table using v.selectedRows of the
        //Datatable Component
        var selectedTasks = cmp.find("taskTable").get("v.selectedRows");

        for(var i = 0;i < selectedTasks.length;i++){
            selectedTasks[i].Stage__c = 'Completed';
            selectedTasks[i] = JSON.parse(JSON.stringify(selectedTasks[i]));
        }

        var action = cmp.get("c.markTasksAsCompleted");

        action.setParams({
            "tasks":selectedTasks
        });

        action.setCallback(this,function(resp){
            var state = resp.getState();
            //if SUCCESS, call the rerenderRows() to reflect the changes in the
            //table

```

```

        if(cmp.isValid() && state === 'SUCCESS'){
            cmp.find("taskTable").rerenderRows();
        }
        else{
            console.log(resp.getError());
        }
    });

    $A.enqueueAction(action);
}
})

```

Explanation

- `completeTask` method in the helper is called and all the selected rows in the table are retrieved by accessing the `selectedRows` attribute of the `DataTable` component.
- Stage field of the selected task records are changed to 'Completed'.
- Then `markTasksAsCompleted` apex controller method is called to update the task records.
- After successfully updating the records, calling the `rerenderRows()` would reflect the changes in the table.

3. Add the below method to `ProjectTaskController` Apex Controller:

```

@AuraEnabled
public static void markTasksAsCompleted(List<Task__c> tasks){
    update tasks;
}

```

Running the component:

Now that we have built `projectTaskList` component, we are going to see how this component can be used in Project's Record Home page and as a Standalone App

1. Add Component to the Record Home Page:

- If you are in the Salesforce Classic view, switch to Lightning Experience view.
- Open a Project detail record.
- Click on the Gear icon  -> Click *Edit Page* to edit the Project Detail Layout.
- Create a new tab called *Project Tasks* and drag the `projectTaskList` component under the custom lightning component section to the Lightning Page Layout editor.

- Click Activate and then Save button to save the changes

PROJECTTASK COMPONENT IN PROJECT DETAIL PAGE

2. Add to Standalone Lightning App:

- Create a new Lightning App. Open *Developer console* -> *File* -> *New* -> *Lightning Component* name it **ProjectTask**.
- Add the below markup to the `ProjectTask.cmp`.

```
<aura:component controller="ProjectTaskController"
  implements="force:appHostable">
  <ltng:require
    styles="/resource/ldt__SLDS202/assets/styles/salesforce-
      lightning-design-system.css"/>
```

```

<aura:attribute name="projects" type="ldt_Project__c[]"
    access="private"/>
<aura:attribute name="selectedProjId" type="String"
    access="public"/>
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
<div class="slds">
    <section class="project-list">
        <h3 class="slds-section-title--divider">Task Management
            App</h3>
        <div class="slds-p-top--x-small" style="margin-left:
            30px;margin-right: 30px;">
            <div>Select Project:</div>
            <ui:inputSelect class="slds-input selHeight1"
                value="{!v.selectedProjId}">
                <ui:inputSelectOption label="--None--" text="" />
                <aura:iteration items="{!v.projects}"
                    var="project">
                    <ui:inputSelectOption label="{!project.Name}"
                        text="{!project.Id}" />
                </aura:iteration>
            </ui:inputSelect>
            <c:projectTaskList projectId="{!v.selectedProjId}" />
        </div>
    </section>
</div>
</aura:component>

```

- Add below method to the ProjectTaskController.js:

```

({
    doInit : function(cmp, event, helper) {
        var action = cmp.get("c.getProjects");

        action.setCallback(this,function(resp){
            var state = resp.getState();
            if(cmp.isValid() && state === 'SUCCESS'){
                cmp.set("v.projects",resp.getReturnValue());
            }
            else{
                console.log(resp.getError());
            }
        });

        $A.enqueueAction(action);
    },
})

```

- Add the below code to the ProjectTask.css:

```

.THIS .selHeight1{
    height: 2.5em !important;
    width: 300px !important;
    margin-top: 5px;
}

```

}

- Create a new Lightning App. Open *Developer console* -> *File* -> *New* -> *Lightning App* name it **ProjectTaskApp**.
- Add the below markup to the `ProjectTaskApp.App`.

```
<aura:application>
  <c:ProjectTask />
</aura:application>
```

- Open the ProjectTaskApp, by clicking on the Preview button.

TASK MANAGEMENT APP

Select Project:
Project 1 ▼

Project Tasks

Items Per Page: 10 ▼

Search by Column: --None-- ▼

Search:

Add Task Complete Task

☐	ACTION	TASK NAME ↑	START DATE	DUE DATE	STAGE
☐	Edit Del	Task 1	13-Apr-2017	14-Apr-2017	In Progress ▼
☐	Edit Del	Task 2	27-Apr-2017	22-Apr-2017	Completed
☐	Edit Del	Task 3	20-Apr-2017	21-Apr-2017	Completed
☐	Edit Del	Task 4	29-Apr-2017	14-Apr-2017	Started ▼
☐	Edit Del	Task 5	13-Apr-2017	13-Apr-2017	In Progress ▼

Showing 1 to 5 of 5 records

Previous Next

PROJECTTASK COMPONENT IN PROJECTTASKAPP

Change Logs:

Version:1.9.6:

- Added a `css` to disabled checkbox.
- Bug fixes.

Version:1.9.3:

- Added column properties `searchable, visible, sortable, exportable`
- Added column config action `rowActionPosition, rowActionWidth`
- Exposed `noRecordsMessage` attribute.

Version:1.9.2

- Bug fixes.

Version: 1.9.1

- Updated `slds` style to enhance the table

Version: 1.9.0

- Added a cell click Event
- Added a Export feature

Version: 1.8.0

- Fixed the select all issue when the row with checkbox hidden is taken into account
- Added truncation capability to string, rich text area columns.

Version : 1.7.8

- Added 'format' property for number, currency and percentage columns

Version : 1.7.7

- Fixed items per page and pagination info alignment issues.

Version : 1.7.6

- Fixed cmp.set issue when invoking initialize method

Version : 1.7.5

- Add two new aura:methods(updateRowByKey & deleteRowByKey)
- Added 'target' support in 'reference' type column
- Fixed issue in using nested property in 'value' of reference column

Version : 1.7.4

- Refactored logics in the component
- Bug fixes

Version : 1.7.3

- Fixed Cell data disappearing issue when searching.



USA

1765 Greensboro Station Place
Suite 900
McLean, VA 22102, USA
+1-703-222-6032

India

First Floor, 'A' Block South, Tidel Park,
Chennai 600113, India
+91-94450-70999, +91-99447-89924
info@softsquare.biz