**Basics of Devextreme**

**Introduction to DevExtreme**

DevExtreme is a suite of feature-rich UI components by DevExpress that enables developers to build modern, responsive web applications. It supports jQuery, Angular, React, and Vue.

**Key Features:**

* Data visualization (charts, grids, pivot tables, etc.)
* Form elements and editors (DateBox, TextBox, SelectBox, etc.)
* Supports themes and custom styling

**Installation – NuGet Package**

Install DevExtreme in a Visual Studio 2022 project using NuGet:

1. Open **Package Manager Console** (Tools > NuGet Package Manager > Package Manager Console).
2. Run the command

**Install-Package DevExtreme.Web -Version 21.1.3**

1. Include required scripts and styles in your project:

A screen shot of a computer program

AI-generated content may be incorrect.

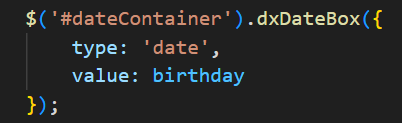
**Widget Basics – jQuery**

DevExtreme provides jQuery-based widgets that can be initialized using the $() function. Each widget has a structured API for configuration, interaction, and event handling.

**Create and Configure a Widget**

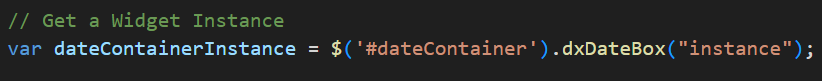
To create a widget, use jQuery to initialize it on an HTML element:





**Get a Widget Instance**

Once a widget is created, you can get its instance for further interactions:



**Get and Set Options**

Modify widget options dynamically:

A screen shot of a computer code

AI-generated content may be incorrect.

Retrieve current options:

A black background with white text

AI-generated content may be incorrect.

**Call Methods**

Invoke widget methods:

A black background with blue and green text

AI-generated content may be incorrect.

**Handle Events**

Bind event handlers to the widget:

A screen shot of a computer code

AI-generated content may be incorrect.

**Destroy a Widget**

Dispose and Remove the widget instance:

A computer code with text

AI-generated content may be incorrect.

**Editors – Overview**

**1. Check Box**

A Check Box represents a binary choice (checked or unchecked), commonly used for options like terms acceptance.

**Options:**

* value: Sets the checkbox state (true, false, undefined).
* disabled: Disables/enables the checkbox.
* elementAttr: Adds custom attributes (e.g., class, name, id) to the DOM element.
* focusStateEnabled: Enables keyboard navigation focus.
* accessKey: Assigns a keyboard shortcut (e.g., Alt + key).
* hint: Displays a tooltip on hover.
* name: Adds a name attribute (useful in forms).
* tabIndex: Sets the order for tab navigation.
* text: Displays a label alongside the checkbox.
* visible: Shows or hides the checkbox.
* rtlEnabled: Supports right-to-left languages.

**Methods:**

* instance(): Retrieves the component’s instance.
* option(property): Gets or sets a property value.
* focus(): Sets focus on the checkbox.
* reset(): Resets the value to its default.
* beginUpdate(): Suspends UI updates for batch changes.
* endUpdate(): Applies batch changes and updates the UI.
* registerKeyHandler(key, handler): Binds a custom keyboard event.
* element(): Returns the component’s root DOM element.
* dispose(): Destroys the component and removes it from the DOM.

**Events:**

* on("valueChanged", handler): Fires when the value changes.
* on("optionChanged", handler): Fires when any option is modified.
* on("focusIn", handler): Fires when the checkbox receives focus.
* on("focusOut", handler): Fires when the checkbox loses focus.
* off(eventName): Unsubscribes from an event.

**2. Date Box**

The Date Box allows date selection with multiple formats.

**Options**

* type: Defines the type of input (date, time, datetime).
* acceptCustomValue: Allows custom values.
* accessKey: Sets a keyboard shortcut (e.g., Alt + t).
* applyValueMode: Controls value submission (instantly, useButtons).
* max / min: Sets allowed date range.
* dateOutOfRangeMessage: Displays a custom error for out-of-range dates.
* disabled: Disables the input.
* disabledDates: Prevents selection of specific dates (e.g., weekends).
* elementAttr: Adds custom HTML attributes.
* height, width: Sets component dimensions.
* hint: Displays a tooltip on hover.
* inputAttr: Adds attributes to the input element.
* openOnFieldClick: Opens the calendar when clicking the field.
* placeholder: Displays a hint inside the input.
* rtlEnabled: Enables right-to-left layout.
* showClearButton: Shows a button to clear the value.
* stylingMode: Sets input style (outlined, underlined, filled).
* useMaskBehavior: Restricts input to the date format.

**Methods :**

* .instance(): Gets the component’s instance.
* .option(name, value): Gets/sets property values.
* .focus(): Sets focus to the component.
* .blur(): Removes focus from the component.
* .beginUpdate()/.endUpdate(): Suspends and resumes UI updates for batch changes.
* .getButton("clear"): Retrieves the clear button element.
* .close(): Closes the calendar popup.
* .dispose(): Destroys the component and removes it from the DOM.

**Event :**

* .on("valueChanged", handler): Fires when the date changes.
* .on("keyDown", handler): Fires when a key is pressed.
* .on("copy", handler): Fires when content is copied.
* .on("cut", handler): Fires when content is cut.
* .on("paste", handler): Fires when content is pasted.
* .off("eventName"): Unsubscribes from an event.

**3. Drop Down Box**

A Drop Down Box offers selectable options from a list.

**1. Single Selection DropDownBox**

* **items**: Static list of anime titles.
* **placeholder, height, width, hint**: Basic UI configurations.
* **showDropDownButton**: Disabled dropdown button.
* **acceptCustomValue**: Disallows custom input.
* **contentTemplate**: Renders a dxList with single selection.
  + Sets the selected value on click.
  + Closes the dropdown programmatically.

**2. Multiple Selection DropDownBox**

* **items**: List of names.
* **contentTemplate**: Renders a dxList with multiple selection.
  + Displays selected items as a comma-separated string.
* **buttons**: Adds a clear button:
  + Clears selected items in dxList.
  + Resets the dxDropDownBox value.

**Key Methods and Events:**

* .option('value', value): Updates the displayed value.
* .dxList({ selectionMode }): Manages selection types.
* .onSelectionChanged(): Triggers on item selection.
* .content().find().dxList('instance'): Accesses the list component for programmatic updates.

**4. Number Box**

A Number Box restricts input to numeric values.

**Options:**

* value: Initial value (NumberBox: numeric, Phone: string)
* format: Defines input format (e.g., "00.0" for decimals).
* step: Incremental step value for NumberBox.
* showSpinButtons, useLargeSpinButtons: Enable spin controls for NumberBox.
* placeholder: Displays hint text.
* width: Sets input width.
* showClearButton: Shows a clear button inside the input.
* validationRules: Custom validation rules (e.g., regex for phone numbers).

**Methods:**

* .option(name, value): Gets or updates component options.
* .dispose(): Removes the component from memory.
* .focus(), .blur(): Manages input focus.
* .reset(): Clears input value.

**Events:**

* onInitialized: Triggered on component creation.
* onContentReady: Fires when rendering completes.
* onDisposing: Fires on component removal.
* onFocusIn, onFocusOut: Handle focus events.
* onValueChanged: Captures input changes.
* onInput, onKeyDown, onKeyUp: Capture typing events.

**5. Select Box**

Displays selectable options with advanced search and grouping.

**Options**

* **SelectBox (Array Data Source):**
  + dataSource: Uses DevExpress.data.DataSource with array store.
  + displayExpr: Displays the name field.
  + valueExpr: Stores the id field.
  + grouped: Groups options based on anime or genre.
  + groupTemplate: Customizes group header with an icon.
  + fieldTemplate: Displays a custom text box when an item is selected.
  + itemTemplate: Customizes how items are shown in the dropdown.
* **SelectBox (Custom Store from JSON):**
  + dataSource: DevExpress.data.CustomStore with .getJSON().
  + searchEnabled: Enables search.
  + searchExpr, minSearchLength, searchMode, searchTimeout: Configure search behavior.
  + noDataText: Displays when no results are found.
  + itemTemplate: Displays item details.

**Methods**

* option(): Updates control options (e.g., searchExpr).
* repaint(): Redraws the component after option changes.
* dxSelectBox("instance"): Retrieves the component instance.

**Events**

* onValueChanged: Triggered when selection changes (e.g., grouping checkbox toggling anime/genre).
* onInput, onKeyDown, onKeyUp: Captures input and keyboard events.
* onInitialized, onContentReady, onDisposing: Lifecycle events for the components.
* onSearch: Fires during search operations (if configured).

**6. Text Area**

Multi-line text input component.

**Options:**

* **Common Options:**
  + stylingMode: Sets the style of the editor (filled, underlined, etc.).
  + placeholder: Displays placeholder text.
* **For #textArea:**
  + height: Sets a fixed height (autoResize is disabled when height is set).
* **For #textArea2:**
  + autoResizeEnabled: Enables auto resizing (false by default).
  + minHeight / maxHeight: Sets height boundaries.
  + maxLength: Limits character input.

**7. Text Box**

Single-line text input component.

**Options:**

* mask: Adds an input mask (e.g., '00/00/0000' for dates).
* maskChar: Character to represent empty spaces ('x').
* showMaskMode: Displays mask on focus or always.
* hint: Tooltip for user guidance.
* maskInvalidMessage: Custom error message for invalid input.
* mode: Input type ('password', 'email', 'text', etc.).
* buttons: Adds buttons inside the input (e.g., password toggle).
* useMaskedValue: Returns value with mask if true.
* maskRules: Custom validation for masked input (e.g., hours/minutes format)

**Methods:**

* option(): Dynamically changes properties (e.g., toggle input mode).
* dxTextBox('instance'): Gets the widget instance.

**Events:**

* **onValueChanged**: Logs previous and current values.
* **onClick** (in button): Toggles visibility for password inputs.

**8. Button**

Triggers actions on user clicks.

**Options:**

* stylingMode: Variants - 'contained', 'outlined', 'text'.
* type: Button types - 'normal', 'success', 'danger', 'default'.
* icon: Displays icons, e.g., 'preferences', custom icons like 'discord.svg'.
* text: Sets button label.

**Methods**: instance(), option(), dispose()

**Events**: onClick, onOptionChanged

**9. File Uploader**

Uploads files with drag-and-drop support.

**Options:**

* uploadUrl: Upload endpoint URL.
* abortUpload: Custom abort handler.
* selectButtonText: Button label for file selection.
* dropZone & dialogTrigger: Enable drag-and-drop and dialog triggers.
* multiple: Allow multiple file uploads.
* uploadMode: Upload trigger behavior (instantly, useButtons, useForm).
* accept: Allowed file types.
* maxFileSize, minFileSize: File size restrictions with custom messages.
* allowCanceling: Enable/disable upload cancellation.
* labelText: Label display for accepted file types.
* chunkSize: Size for chunked uploads.

**Methods & Events:**

* Lifecycle events: onBeforeSend, onUploadStarted, onProgress, onUploaded, onFilesUploaded, onUploadError, onUploadAborted.
* Drop zone interactions: onDropZoneEnter, onDropZoneLeave.
* abortUpload(): Manually stops an ongoing upload.

**10. Validation**

Ensures input accuracy according to rules.

* Options: validationRules, isValid, validationMessage
* Methods: validate(), reset(), option()
* Events: onValidated, onOptionChanged

**11. Radio Group**

Displays multiple options for single selection.

**Options:**

* dataSource: Provides review options with id, text, and color.
* itemTemplate: Customizes item display, applying color styling from itemData.color.
* layout: Displays items in a horizontal layout.

**Methods**: option(), reset(), instance()

**Events**: onValueChanged, onOptionChanged

**Data Layer**

**ArrayStore :**

ArrayStore is a store that provides an interface for loading and editing an in-memory array and handling related events.

**NOTE :**

The **ArrayStore** is immutable. You cannot change its configuration at runtime. However, you can use its methods to manipulate it.

**Options:**

**data** : Specifies the store's associated array.

**errorHandler** : Specifies the function that is executed when the store throws an error.

**key** : Specifies the key property (or properties) that provide(s) key values to access data items. Each key value must be unique.

**onInserted** : A function that is executed after a data item is added to the store.

**onInserting :** A function that is executed before a data item is added to the store.

**onLoaded** : A function that is executed after data is loaded to the store.

**onLoading :** A function that is executed before data is loaded to the store.

**onModified** : A function that is executed after a data item is added, updated, or removed from the store.

**onModifying** : A function that is executed before a data item is added, updated, or removed from the store.

**onPush** : The function executed before changes are pushed to the store.

**onRemoved** : A function that is executed after a data item is removed from the store.

**onRemoving** : A function that is executed before a data item is removed from the store.

**onUpdated** : A function that is executed after a data item is updated in the store.

**onUpdating** : A function that is executed before a data item is updated in the store.

**Methods :**

**byKey(key):** Gets a data item with a specific key.

**clear():** Clears all the ArrayStore's associated data.

**insert(values):** Adds a data item to the store.

**key():** Gets the key property (or properties) as specified in the key property.

**keyOf(obj):** Gets a data item's key value.

**load():** Starts loading data.

**load(options):** Starts loading data.

**off(eventName):** Detaches all event handlers from a single event.

**off(eventName, eventHandler):** Detaches a particular event handler from a single event.

**on(eventName, eventHandler):** Subscribes to an event.

**on(events):** Subscribes to events.

**push(changes):** Pushes data changes to the store and notifies the DataSource.

**remove(key):** Removes a data item with a specific key from the store.

**totalCount(options):** Gets the total count of items the load() function returns.

**update(key, values):** Updates a data item with a specific key.

**CustomStore**

A CustomStore in DevExtreme is used when you need to load, modify, or manage data from a custom data source rather than a standard endpoint or predefined data store. It gives you full control over data operations like loading, filtering, grouping, and sorting.

**Options :**   
**byKey:** Specifies a custom implementation of the byKey(key) method.

**cacheRawData:** Specifies whether raw data should be saved in the cache. Applies only if loadMode is "raw".

**insert:** Specifies a custom implementation of the insert(values) method.

**key:** Specifies the key property (or properties) that provide key values to access data items. Each key value must be unique.

**load:** Specifies a custom implementation of the load(options) method.

**loadMode:** Specifies how data returned by the load() function is treated.

**remove:** Specifies a custom implementation of the remove(key) method.

**totalCount:** Specifies a custom implementation of the totalCount(options) method.

**update:** Specifies a custom implementation of the update(key, values) method.

**Methods :**

**clearRawDataCache** : Deletes data from the cache. Takes effect only if the cacheRawData property is true.

**DataSource:**The DataSource is an object that provides an API for processing data from an underlying store.

**Options**:

**filter** : Specifies data filtering conditions.

**group** : Specifies data grouping properties.

**map** : Specifies an item mapping function.

**onChanged** : A function that is executed after data is loaded.

**onLoadError** : A function that is executed when data loading fails.

**onLoadingChanged** : A function that is executed when the data loading status changes.

**pageSize** : Specifies the maximum number of data items per page. Applies only if paginate is true.

**paginate** : Specifies whether the DataSource loads data items by pages or all at once. Defaults to false if group is set; otherwise, true.

**postProcess** : Specifies a post-processing function.

**requireTotalCount** : Specifies whether the DataSource requests the total count of data items in the storage.

**reshapeOnPush** : Specifies whether to reapply sorting, filtering, grouping, and other data processing operations after receiving a push.

**searchExpr** : Specifies the fields to search.

**searchOperation** : Specifies the comparison operation used in searching. Accepted values: "=", "<>", ">", ">=", "<", "<=", "startswith", "endswith", "contains", "notcontains".

**searchValue** : Specifies the value to which the search expression is compared.

**select** : Specifies the fields to select from data objects.

**sort** : Specifies data sorting properties.

**store** : Configures the store underlying the DataSource.

**Methods:**

filter() : Gets the filter property's value.

filter(filterExpr) : Sets the filter property's value.

group() : Gets the group property's value.

group(groupExpr) : Sets the group property's value.

isLastPage() : Checks whether the count of items on the current page is less than the pageSize. Takes effect only with enabled paging.

isLoaded() : Checks whether data is loaded in the DataSource.

isLoading() : Checks whether data is being loaded in the DataSource.

items() : Gets an array of data items on the current page.

key() : Gets the value of the underlying store's key property.

load() : Starts loading data.

loadOptions() : Gets an object with current data processing settings.

pageIndex() : Gets the current page index.

pageIndex(newIndex) : Sets the index of the page that should be loaded on the next load() method call.

pageSize() : Gets the page size.

pageSize(value) : Sets the page size.

paginate() : Gets the paginate property's value.

paginate(value) : Sets the paginate property's value.

reload() : Clears currently loaded DataSource items and calls the load() method.

requireTotalCount() : Gets the requireTotalCount property's value.

requireTotalCount(value) : Sets the requireTotalCount property's value.

searchExpr() : Gets the searchExpr property's value.

searchExpr(expr) : Sets the searchExpr property's value.

searchOperation() : Gets the searchOperation property's value.

searchOperation(op) : Sets the searchOperation property's value.

searchValue() : Gets the searchValue property's value.

searchValue(value) : Sets the searchValue property's value.

select() : Gets the select property's value.

select(expr) : Sets the select property's value.

sort() : Gets the sort property's value.

sort(sortExpr) : Sets the sort property's value.

store() : Gets the instance of the store underlying the DataSource.

totalCount() : Gets the number of data items in the store after the last load() operation without paging. Takes effect only if requireTotalCount is true.

**LocalStore :**

The LocalStore is a store that provides an interface for loading and editing data from HTML Web Storage (also known as window.localStorage) and handling related events.

Options:

data : Specifies the store's associated array.

flushInterval : Specifies a delay in milliseconds between when data changes and the moment these changes are saved in the local storage. Applies only if immediate is false.

Immediate : Specifies whether the LocalStore saves changes in the local storage immediately.

Key : Specifies the key property (or properties) that provide(s) key values to access data items. Each key value must be unique.

Name : Specifies the name under which data should be saved in the local storage. The dx-data-localStore- prefix will be added to the name.

**Query :**

The Query is an object that provides a chainable interface for making data queries.

**Methods:**

aggregate(seed, step, finalize) : Calculates a custom summary for all data items.

aggregate(step) : Calculates a custom summary for all data items.

avg() : Calculates the average of all values. Applies only to numeric arrays.

avg(getter) : Calculates the average of all values found using a getter.

count() : Calculates the number of data items.

enumerate() : Executes the Query. This is an asynchronous alternative to the toArray() method.

filter(criteria) : Filters data items using a filter expression.

filter(predicate) : Filters data items using a custom function.

groupBy(getter) : Groups data items by the specified getter.

max() : Calculates the maximum value. Applies only to numeric arrays.

max(getter) : Calculates the maximum of all values found using a getter.

min() : Calculates the minimum value. Applies only to numeric arrays.

min(getter) : Calculates the minimum of all values found using a getter.

select(getter) : Selects individual fields from data objects.

slice(skip, take) : Gets a specified number of data items starting from a given index.

sortBy(getter) : Sorts data items by the specified getter in ascending order.

sortBy(getter, desc) : Sorts data items by the specified getter in the specified sorting order.

sum() : Calculates the sum of all values.

sum(getter) : Calculates the sum of all values found using a getter.

thenBy(getter) : Sorts data items by one more getter in ascending order.

thenBy(getter, desc) : Sorts data items by one more getter in the specified sorting order.

toArray() : Gets data items associated with the Query. This is a synchronous alternative to the enumerate() method.

**DataGrid**

**Data Binding :**

**Simple Array:**

You can use the JavaScript DataGrid component to display and edit data from an array of objects. Use the dataSource and keyExpr properties to specify the bound array and its key field.

Use the columns property to specify a list of data fields you want to display within the JavaScript DataGrid as columns. If the **columns** property is not specified, the component creates columns for all data fields available in objects from the bound array.

**Ajax Request:**

We can create customStore with load , insert , update and remove that have ajax request of get , post, put and delete.

For enabling editing and adding we can true allowAdding and allowUpdating in editing.

**Paging and Scrolling :**

**Record Paging :**

DataGrid splits records into multiple pages.Users can use a scroll bar or a pager to navigate between pages.

For enabling paging:

    paging: {

      enabled: true, // default "true"

      pageIndex: 3, // default "0"

      pageSize: 5, // default 20

    },

For Configuring pager:

    pager: {

      visible: true, // default "auto"

      showPageSizeSelector: true,

      allowedPageSizes: [5, 10, 15, "all"],

      displayMode: "full", // "full" | "compact" | "adaptive"

      showInfo: true,

      infoText: "Page {0} of {1}", // {0} - current page, {1} - total pages ,{2} - total records

      showNavigationButtons: true,

    },

**Virtual and Infinite Scrolling :**

If the DataGrid component is bound to a large dataset, you can enable the virtual scroll feature to optimize data load times and improve user navigation. The component calculates the overall number of visible rows and displays a scrollbar that allows users to navigate to any section of rows.

If the DataGrid component is bound to a large dataset, you can enable infinite scroll mode to optimize data load times and improve user navigation. The component initially displays one page of rows. When users scroll to the end of the view, the JavaScript DataGrid loads an additional page

For configuration :

    scrolling: {

      mode: "virtual", // 'infinite' | 'standard' | 'virtual'

      useNative: false,

      showScrollbar: "onHover", // 'always' | 'never' | 'onHover' | 'onScroll'

      preloadEnabled: true, // pages are loaded in advance for smoother scrolling

      scrollByContent: true, // scroll content with a swipe gesture (useNative : false)

      scrollByThumb: true, // scroll content with the scrollbar (useNative : false)

      rowRenderingMode: "virtual",

      columnRenderingMode: "virtual",

      // standard: Renders all columns at once

      // virtual: Renders only those columns that get into the viewport.

      // Infinite: Each next page is loaded once the scrollbar reaches the end of its scale.

    },

**Editing :**

Row Editing :

The DataGrid allows users to edit data in multiple modes. This demo shows the *"row"* edit mode. To start editing any row, click *"Edit"* in it. Only one row can be in the edit state at a time.

Cell Editing:

Users can modify DataGrid data cell by cell. In this mode, only one cell can be in the edit state at a time.DataGrid saves changes immediately after the focus leaves cell.

For configuration of row or cell mode :

      editing: {

        mode: "cell", // "cell" or "row "

        allowUpdating: true,

        allowDeleting: true,

        allowAdding: true,

        useIcons: true, // to use icons insted of text

        selectTextOnEditStart: true, // to select all text in cell when start editing

      },

Batch Editing :

We can use batch edit mode to defer saving multiple cell changes. Changes are stored in a buffer and can be discarded before a user clicks the **Save** button.

Form Editing :

The DataGrid can use the Form component to arrange cell editors when users modify a row. The Form allows users to edit values from visible and hidden columns .

Popup Editing :

DataGrid can display a popup edit form. The form can include any fields from the bound data source, regardless of whether the corresponding column is visible in the grid

For configuration of Batch , Form or Popup :

editing: {

        mode: "batch",

        allowUpdating: true,

        allowDeleting: true,

        allowAdding: true,

        useIcons: true,

        selectTextOnEditStart: true,

        // only for form and popup modes

        form: {

          // removed id in editing

          items: [

            {

              itemType: "group",

              caption: "Anime Info",

              // items: ["name", "anime", "genre"]  // for simple

              // for specifying editorType

              items: [

                { dataField: "name" },

                { dataField: "anime" },

                {

                  dataField: "genre",

                  editorType: "dxSelectBox",

                  editorOptions: {

                    items: [

                      "Action",

                      "Adventure",

                      "Magical Girl",

                      "Thriller",

                      "Comedy",

                    ],

                  },

                },

              ],

            },

          ],

        },

        // only for Popup

        popup: {

          showTitle: true,

          title: "Anime Info",

          width: 500,

          height: 300,

        },

      },

So as given in code, we can remove or add hidden column in from and also configure editorType and edtiorOptions for Form and Popup.

DataGrid allows you to validate user input. You can apply pre-defined **validation rules** or custom rules to individual columns.

Example configuration :

{

        dataField: "name",

        dataType: "string",

        // data validation

        validationRules: [

          { type: "required" },

          {

            type: "stringLength",

            min: 2,

            message: "Name at least have 2 letters",

          },

        ],

      },

We can assign a **lookup** editor to a column. This editor displays a drop-down list populated with values from the specified data source. Users can filter the drop-down list to quickly locate required values.

For configuration:

 {

        dataField: "genre",

        dataType: "string",

        // Cascading Lookups

        // to set anime "null" when genre is selected

        setCellValue: (rowData, value) => {

          rowData.genre = value;

          rowData.anime = null;

        },

        // adds dropDown in UI

        lookup: {

          dataSource: genre,

        },

      },

      {

        dataField: "anime",

        dataType: "string",

        // Cascading Lookups

        // to filter anime according genre selected

        lookup: {

          dataSource: (options) => {

            return {

              store: anime,

              filter: options.data ? ["genre", "=", options.data.genre] : null,

            };

          },

          displayExpr: "anime",

          valueExpr: "anime",

        },

      },

**Grouping :**

Record Grouping :

DataGrid allows users to group data against a single column or multiple columns.

To group data, users can drag and drop column headers onto and from the area above the grid. This area is called the groupPanel. Set its **visible** property to **true** to show it.

For grouping configuration :

    grouping: {

      allowCollapsing: true, // can collaps a group if true

      autoExpandAll: false, // all collaped in default if false

      expandMode: "rowClick", //  'buttonClick' | 'rowClick'

      contextMenuEnabled: true, // group/ungroup with rightClick on column

    },

    groupPanel: {

      visible: true, // to enable group panel

      allowColumnDragging: true, // enable/disable dragging

      emptyPanelText: "Drag column here",

    },

**Filtering :**

The filter row, located under the column headers, allows a user to type any value and select filter operations.

With the header filter, which is called by clicking the filter icon in any column header, a user can choose from a set of predefined values.

The search panel at the top of the JavaScript DataGrid allows searching for values in all columns at once.

Filter row and filter panel configuration :

 filterRow: {

      visible: true, // default false

      applyFilter: "onClick", // onClick | auto

      // showOperationChooser: false, // default true

    },

    filterPanel: {

      // displays the applied filter expression

      visible: true, // default false

      customizeText: (e) => {

        // console.log(e);

        return `Filter : ${e.text}`;

      },

      filterEnabled: false, // by default filter appiled or not

    },

Header filter configuration:

 headerFilter: {

      visible: true,

      allowSearch: true,

      searchTimeout: 100, //  when the search is executed.

      height: 300, // 325, 315 (Material)

    },

**Sorting:**

DataGrid component can sort values by a single or multiple columns. Use the **sorting.mode** property to specify the sort mode.

Sorting configuration :

 sorting: {

        // use "shift" to sort

        // use "ctrl" to remove sort

        mode: "multiple", //  "single" (default) | "multiple" | "none"

        showSortIndexes: true, // sorting order number , only for 'multiple'

      },

**Selection :**

DataGrid allows users to select only one row at a time. To enable this mode, set the **selection**.**mode** property to *"single"*. Press Ctrl to unselect a row.

Sets **selection**.**mode** to *"multiple"*. This allows users to select multiple rows using checkboxes or keyboard shortcuts.

Selection configuration :

      selection: {

        mode: "multiple", // single | multiple | none (default)

        allowSelectAll: true, // can/can't select all

        selectAllMode: "page", // page | allPages (default)

        showCheckBoxesMode: "always", // always | none | onClick (default) | onLongTap (long mouse click)

      },

**Columns :**

**Columns based on a Data Source :** To use data source object fields to automatically generate columns, pass an array of objects to the dataSource property and specify the keyExpr .

**Multi-Level Headers:** DataGrid component allows you to group multiple columns under one header

**Column Resizing :**DataGrid sets the same width for all columns, but you can change column widths by enabling resizing.

**Command Column Customization :**DataGrid supports multiple predefined types of command columns. Each column type supports one type of action (edit, select, drag, etc.)

Columns has many options and we can configure using those.

**State Persistence :**

DataGrid can persist its state. If a user sorts and filters data, groups, reorders and resizes columns, or makes other changes, the component saves these modifications and restores them when a user reloads the page.

Configurations for state persistence :

      stateStoring: {

        type: "sessionStorage", //  'custom' | 'localStorage' (default)| 'sessionStorage'

        enabled: true,

        storageKey: "gridStorageKey",

      },

**Appearance :**

To change appearance of DataGrid :

      showColumnLines: true,

      showRowLines: true,

      rowAlternationEnabled: true,

      showBorders: true,

**Template :**

**Column templates** allow you to display custom content in column cells. Use the cellTemplate property to configure these templates.

**Row templates** allow you to display custom content in data rows.

DataGrid comes with vast capabilities for setting up custom templates for grid cells. We can use **onCellPrepared** property provide a custom template for cells.

DataGrid includes an integrated toolbar that displays predefined and custom controls. In 21.2 version , we can customize toolbar using  **onToolbarPreparing**.

**Data Summaries :**

**Total Summaries :** To display total summaries, populate the summary.totalItems array with configuration objects. Each object should specify a column that supplies data for summary calculation and a summaryType—an aggregate function that should be applied to this data.

**Group Summaries :** To configure group summaries, populate the summary.groupItems array with summary configuration objects. Each object should specify a column that supplies data for summary calculation and a summaryType—an aggregate function that should be applied to this data.

**Custom Summaries :** DataGrid can calculate custom summaries on the client or server side. For client side using  summaryType: "custom".

**Master-Detail :**

Master-Detail View allows users to expand a row ('master') and display the related content ('details') in the expanded section. To configure this feature, set the masterDetail object's enabled property to **true** and use a template to specify detail section content.

To configure master-detail :

masterDetail: {

      enabled: true,

      //   autoExpandAll: true, // expanded or collapsed

      template: (container, info) => {

        container.append(

          // custom template...

        );

      },

    },

**Export :**

Excel :

To enable export in the DataGrid, reference or import the ExcelJS and FileSaver libraries. Set export.enabled to true.

Pdf: DataGrid allows you to export its contents to a PDF document.

To enable PDF export operations, import the jsPDF library and set the export.enabled property to true

Cdn require for excel and pdf :

    <!-- Excel Export -->

    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-polyfill/7.4.0/polyfill.min.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/exceljs/4.1.1/exceljs.min.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/FileSaver.js/2.0.2/FileSaver.min.js"></script>

    <!-- pdf Export  -->

    <script src="https://cdnjs.cloudflare.com/ajax/libs/jspdf/2.5.1/jspdf.umd.min.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/jspdf-autotable/3.7.1/jspdf.plugin.autotable.min.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/jszip/3.10.1/jszip.min.js"></script>

Common configuration :

      // for enabling exporting

      export: {

        enabled: true,

        allowExportSelectedData: true,

      },

After this configuration , we can configure excel or pdf export using onExporting.

**Adaptability :**

DataGrid automatically adapts its layout to screens with different sizes.

If columns do not fit the selected orientation, the DataGrid hides them one by one, starting with the rightmost column. Information from the hidden columns is still available in adaptive detail rows.

To enable this feature, set the columnHidingEnabled property to **true**.

**Navigation**

DevExtreme Navigation components aids navigation in applications. DevExtreme provides various navigation controls, including Accordions, Menus, Drawers, TabPanels, and Toolbars. They suit any design and can be easily customized. This example demonstrates the TreeView and TabPanel controls.

**Menu :**

**Important Options**

* orientation: Specifies the orientation of the menu. Possible values are "horizontal" and "vertical".
* submenuDirection: Specifies the direction in which submenus are displayed. Possible values are "auto", "leftOrTop", and "rightOrBottom".
* showFirstSubmenuMode: Configures the mode in which the first submenu is shown. It includes the following options:
  + name: Specifies the mode name. Possible values are "onClick" and "onHover".
  + delay: Specifies the delay for showing and hiding the submenu.

**Methods**

* getItem(index): Returns the item at the specified index.
* getSubMenuInstance(rootElement): Returns the submenu instance for the specified root element.

**Events**

* onSubmenuShowing: Raised before a submenu is shown.
* onSubmenuShown: Raised after a submenu is shown.
* onSubmenuHiding: Raised before a submenu is hidden.
* onSubmenuHidden: Raised after a submenu is hidden.

**TreeView**

**Important Options**

* dataSource: Configures the data source for the tree view.
* searchEnabled: Enables the search functionality within the tree view.
* showCheckBoxesMode: Configures the visibility of checkboxes in the tree view. Possible values are "none", "normal", and "selectAll".

**Methods**

* getNodes(): Returns all nodes in the tree view.
* getNode(index): Returns the node at the specified index.
* selectItem(itemElement): Selects the specified item element.

**Events**

* onItemSelectionChanged: Raised when an item is selected or unselected.
* onItemExpanded: Raised when an item is expanded.
* onItemCollapsed: Raised when an item is collapsed.

**Dialogues & Notification Documentation**

**Load Indicator**

The Load Indicator component notifies users that a process is in progress.

**Important Options**

* indicatorSrc: Specifies a custom indicator image source.

**Methods**

* show(): Shows the load indicator.
* hide(): Hides the load indicator.

**Events**

* onContentReady: Raised when the content is ready.
* onDisposing: Raised before the widget is removed.

**Load Panel**

The Load Panel is a UI component notifying the viewer that a process is in progress.

**Important Options**

* message: Specifies the message displayed on the load panel.
* shadingColor: Specifies the shading color.
* showIndicator: Specifies whether to show the loading indicator.
* showPane: Specifies whether to show the pane.

**Methods**

* show(): Shows the load panel.
* hide(): Hides the load panel.

**Popup**

The Popup component is a UI component that displays a modal window.

**Important Options**

* title: Specifies the title of the popup.
* showTitle: Specifies whether to show the title.
* visible: Specifies whether the popup is visible.
* showCloseButton: Specifies whether to show the close button.

**Methods**

* show(): Shows the popup.
* hide(): Hides the popup.

**Popover**

The Popover component is a UI component that displays a small overlay window.

**Important Options**

* position: Specifies the position of the popover.
* visible: Specifies whether the popover is visible.

**Methods**

* show(): Shows the popover.
* hide(): Hides the popover.

**Toast**

The Toast component is a UI component that displays a notification message.

**Important Options**

* message: Specifies the message displayed in the toast.
* type: Specifies the type of the toast. Possible values are "info", "warning", "error", "success".
* displayTime: Specifies the time span in milliseconds during which the toast is visible.
* position: Specifies the position of the toast.

**Methods**

* show(): Shows the toast.
* hide(): Hides the toast.