To implement autocomplete functionality for all columns (StockCode, Description, TotalCost) in a Telerik ASP.NET Core Grid while avoiding code duplication, we can use a **common model** and a **shared autocomplete editor template**. The approach will dynamically determine which column's data is being queried and use a Dapper call to fetch relevant data.

Here’s how you can achieve this with a clean and reusable design:

**1. High-Level Approach**

* Use a **common editor template** for autocomplete across all columns.
* Pass the column name (or a key) to the controller to determine which database query to execute.
* Use a single Dapper query method to fetch distinct values for the column.
* Dynamically handle data types (string, int, decimal) based on the column.

**2. Grid Configuration**

Let’s define the grid where all three columns (StockCode, Description, TotalCost) have inline autocomplete editors.

**View: Razor Page (Index.cshtml)**

razor

CopyEdit

@(Html.Kendo().Grid<StockModel>()

.Name("Grid")

.Columns(columns =>

{

columns.Bound(c => c.StockCode)

.Title("Stock Code")

.EditorTemplateName("CommonAutoComplete") // Shared autocomplete editor

.Width(150);

columns.Bound(c => c.Description)

.Title("Description")

.EditorTemplateName("CommonAutoComplete") // Shared autocomplete editor

.Width(200);

columns.Bound(c => c.TotalCost)

.Title("Total Cost")

.EditorTemplateName("CommonAutoComplete") // Shared autocomplete editor

.Width(150);

columns.Command(command => command.Edit()).Width(150); // Inline edit button

})

.Editable(editable => editable.Mode(GridEditMode.InLine)) // Inline editing mode

.Pageable()

.Sortable()

.DataSource(dataSource => dataSource

.Ajax()

.Read(read => read.Action("GetGridData", "Grid"))

.Update(update => update.Action("UpdateGridData", "Grid"))

)

)

**Key Points:**

* All three columns (StockCode, Description, TotalCost) use the **same shared editor template** (CommonAutoComplete).
* Inline editing is enabled using GridEditMode.InLine.

**3. Common Editor Template**

The editor template will dynamically fetch data for the respective column based on its field name.

**Editor Template: /Views/Shared/EditorTemplates/CommonAutoComplete.cshtml**

razor

CopyEdit

@model object

@(Html.Kendo().AutoComplete()

.Name(ViewData["ColumnName"].ToString()) // Dynamically set the name based on the column

.Filter("contains") // Filter type

.DataTextField("Value") // The field name of the result returned from the controller

.MinLength(2) // Start searching after 2 characters

.DataSource(source => source

.Read(read => read.Action("GetColumnSuggestions", "Grid") // Call to fetch data

.Data("onAutocompleteData")) // Pass column name dynamically

)

.HtmlAttributes(new { @class = "k-input" })

)

<script>

// Function to send the column name to the controller

function onAutocompleteData() {

return {

columnName: '@ViewData["ColumnName"]' // Column name from ViewData

};

}

</script>

**Key Points:**

* ViewData["ColumnName"] dynamically sets the field name (e.g., StockCode, Description, TotalCost).
* The controller action (GetColumnSuggestions) dynamically retrieves suggestions based on the column name.
* The onAutocompleteData() JavaScript function passes the column name to the controller.

**4. Controller Logic**

The controller will handle the dynamic fetching of data for the columns using Dapper.

**Controller: GridController.cs**

csharp

CopyEdit

using Dapper;

using Microsoft.AspNetCore.Mvc;

using System.Data;

using System.Data.SqlClient;

public class GridController : Controller

{

private readonly string \_connectionString = "YourDatabaseConnectionString";

// Action to provide grid data

public IActionResult GetGridData()

{

// Mock data for the grid

var data = new[]

{

new StockModel { StockCode = 101, Description = "Item 101", TotalCost = 150.50M },

new StockModel { StockCode = 102, Description = "Item 102", TotalCost = 250.75M }

};

return Json(data);

}

// Action to update grid data

[HttpPost]

public IActionResult UpdateGridData([FromBody] StockModel model)

{

// Update logic here (e.g., update database using Dapper)

return Json(model);

}

// Action to fetch autocomplete suggestions

public IActionResult GetColumnSuggestions(string columnName, string text)

{

if (string.IsNullOrWhiteSpace(columnName) || string.IsNullOrWhiteSpace(text))

{

return Json(new List<object>()); // Return an empty list if input is invalid

}

using (IDbConnection db = new SqlConnection(\_connectionString))

{

string query = columnName switch

{

"StockCode" => "SELECT DISTINCT StockCode AS Value FROM StockCodes WHERE CAST(StockCode AS NVARCHAR) LIKE @SearchText",

"Description" => "SELECT DISTINCT Description AS Value FROM StockDescriptions WHERE Description LIKE @SearchText",

"TotalCost" => "SELECT DISTINCT TotalCost AS Value FROM StockCosts WHERE CAST(TotalCost AS NVARCHAR) LIKE @SearchText",

\_ => throw new ArgumentException("Invalid column name.")

};

var parameters = new { SearchText = "%" + text + "%" };

var result = db.Query<object>(query, parameters).ToList();

return Json(result);

}

}

}

**Key Points:**

* The GetColumnSuggestions action dynamically switches the query based on the columnName parameter.
* Dapper is used to fetch distinct values for the column (StockCode, Description, TotalCost).
* The SQL query converts non-string fields (e.g., int, decimal) to NVARCHAR to handle filtering with LIKE.

**5. Model Definition**

**Model: StockModel.cs**

csharp

CopyEdit

public class StockModel

{

public int StockCode { get; set; }

public string Description { get; set; }

public decimal TotalCost { get; set; }

}

**6. SQL Schema Example**

Here’s a rough idea of what your database tables might look like:

**StockCodes Table**

| **StockCode** |
| --- |
| 101 |
| 102 |

**StockDescriptions Table**

| **Description** |
| --- |
| Item 101 |
| Item 102 |

**StockCosts Table**

| **TotalCost** |
| --- |
| 150.50 |
| 250.75 |

**7. Final Output**

With the above setup:

* All three columns (StockCode, Description, TotalCost) will use a shared autocomplete editor.
* The autocomplete dropdown will fetch data dynamically based on the column name.
* Dapper queries ensure high performance for fetching distinct values.