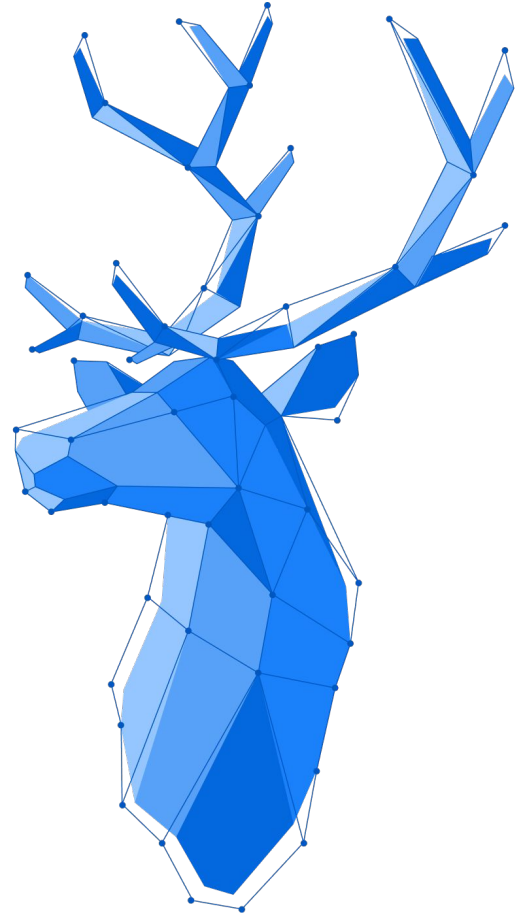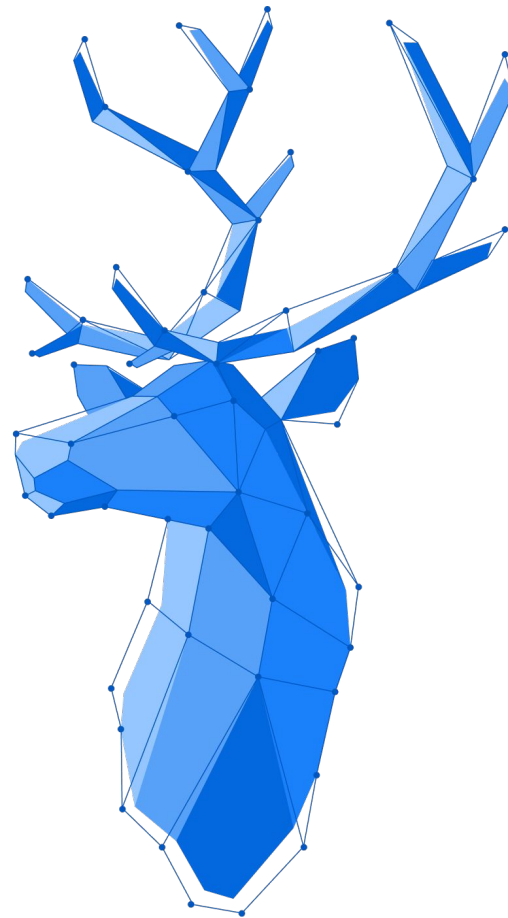# Data lists with Grid: Working with Data Tables

Vaadin Flow

Vaadin Foundation

- Introduction
- Layouting
- Creating Forms
- **Data Lists with Grid**
- Routing and Navigation
- Theming and Styling Applications

vaadin }>

Agenda

- Part 1
  - Grid basics
- Part 2
  - Grid configuration
- Part 3
  - In-Memory Data Providers
- Part 4
  - Lazy Data Providers
- Part 5
  - Grid Pro

Data Lists with Grid, Part 1

Grid basics

vaadin }>

Grid component

	First Name ⇅	Last Name ⇅	Email ⇅
☐	Henry	Carter	henry.carter@example.com
☑	Liam	Perez	liam.perez@example.com
☐	Justin	Garcia	justin.garcia@example.com
☐	Jordan	Howard	jordan.howard@example.com

Grid instantiation

Create columns automatically

Bean Grid: Columns are generated automatically if you pass a class reference in the Grid constructor.

```java
// scans Person class and adds columns automatically
Grid<Person> grid = new Grid<>(Person.class);

// There's a default order, but it's also possible to reorder the generated columns
grid.setColumns("firstName", "lastName", "email", "profession");
```

First name	Last name	Email	Profession

Create columns manually

If there's no class info passed to the constructor, you need to add columns manually. This gives you more flexibility to define how the data should be presented in the columns.

```
// Has nothing to scan -> add columns yourself
Grid<Person> grid = new Grid<>();
```

Adding columns

Add columns using a ValueProvider

Add new column to the grid using an implementation of the ValueProvider interface:

```java
// You can add whatever columns you want using ValueProviders:
public Column<T> addColumn(ValueProvider<T, V> valueProvider)
```

Example:

```java
grid.addColumn(Person::getFirstName);
```

Add columns using property names

Add new columns to the grid by just providing a Java Bean property name. This works only with a Bean Grid:

```java
// add new column using property name
public Column<T> addColumn(String propertyName)
```

Example:

```java
// Works with nested properties
grid.addColumn("address.street");
```

Add columns using a Renderer

Add new columns to the grid using an implementation of a column renderer:

```java
// Or using a renderer
public Column<T> addColumn(Renderer<T> renderer)
```

Example:

```java
grid.addColumn(new TextRenderer<>(
    person -> person.getFirstName() + " " + person.getLastName()
));
```

Built-in Renderers

- TextRenderer (default)
- ComponentRenderer
- LocalDateRenderer
- NumberRenderer
- LocalDateTimeRenderer
- NativeButtonRenderer
- IconRenderer
- LitRenderer
- ColumnPathRenderer
- EditorRenderer

Examples

```java
// Use LocalDateRenderer to customize the birthday format
grid.addColumn(
        new LocalDateRenderer<>(Person::getBirthday, "yyyy-MM-dd")
).setHeader("Date of Birth");


// Use NumberRenderer to output US locale specific format
NumberFormat format = NumberFormat.getInstance(Locale.US);
grid.addColumn(
        new NumberRenderer<>(Person::getEmployeeNumber, format)
).setHeader("Employee number");
```

Date of Birth	Employee number
1953-04-25	12,345
2012-04-25	1,242,131

Examples

```java
// Use LitRenderer to render the full name by combining
// two properties.
grid.addColumn(LitRenderer
        .<Person>of("<b>${item.last}</b>, ${item.first}")
        .withProperty("last", Person::getLastName)
        .withProperty("first", Person::getFirstName)
).setHeader("Full name");
```

Full name
Wilson, Elizabeth
Jones, Mary
Miller, Jennifer

Examples

```
// Render a remove button
grid.addColumn(new ComponentRenderer<>(
        item -> new Button("Remove", e -> { /* action */ })));

// An alternative syntax
grid.addComponentColumn(
        item -> new Button("Remove", e -> { /* action*/ }));
```

Person	Actions
Hi, i'm the component for Jack!	Remove
Hi, i'm the component for Nathan!	Remove
Hi, i'm the component for Andrew!	Remove
Hi, i'm the component for Mickael!	Remove
Hi, i'm the component for Peter!	Remove
Hi, i'm the component for Samuel!	Remove

Column configuration

Column Header

Column header is generated automatically when Grid is created with a class parameter:

```
new Grid<Person>(Person.class)
```

Property "firstName" generates the header "First Name"

Column Header

Column::setHeader() is needed if you manually add the column.

```
grid.addColumn(Person::getLastName).setHeader("Last Name");
```

Column Footer

```java
grid.addColumn(Person::getFirstName).setFooter(
    "Total: " + personList.size() + " people"
);
```

```java
grid.addColumn(Person::getAge).setFooter(
    "Average: " + averageOfAge
);
```

First name	Age
Jack	50
Nathan	20
Andrew	30
Mickael	68
Peter	38
Samuel	53
Anton	37
Aaron	18
Jack	28
Total: 109 people	Average: 46

Column Key

The Column Key is used for retrieving a column with grid.getColumnByKey(key)

Column Key

If the grid was created with a class parameter, the key is generated automatically as the property name.

```java
Grid<Person> grid = new Grid<>(Person.class);
grid.getColumnByKey("firstName");
```

Column Key

The key has to be set explicitly with Column::setKey when the grid was created without a class parameter

```java
Grid<Person> grid = new Grid<>();
grid.addColumn(Person::getFirstName).setKey("firstName");
grid.getColumnByKey("firstName");
```

Accessing a Column without a Key

Note that grid.addColumn() also returns a Column object. Sometimes it's convenient to just hold the column reference directly.

```java
Grid<Person> grid = new Grid<>();
Column nameColumn = grid.addColumn(Person::getName);
```

Column Grouping

You can have multiple header rows and join header cells:

```
HeaderRow halfheaderRow = grid.prependHeaderRow();

Div half1Header = new Div("Half 1");
halfheaderRow
     .join(quarter1, quarter2)
     .setComponent(half1Header);

Div half2Header = new Div("Half 2");
halfheaderRow
     .join(quarter3, quarter4)
     .setComponent(half2Header);
```

	Half 1			Half 2
Year	Quarter 1 ⇕	Quarter 2 ⇕	Quarter 3 ⇕	Quarter 4 ⇕
2017	200	200	200	200
2018	210	210	210	210
2019	220	220	220	220
2020	230	230	230	230
2021	240	240	240	240

Basic data

Populate Data to Grid

Use grid.setItems() to populate data to Grid by passing a Java Collection

```java
Grid<Person> grid = new Grid<>(Person.class);
List<Person> list = getPersons();

// populate Grid with a Collection
grid.setItems(list);
```

Populate Data to Grid

You can also pass a Stream

```java
Grid<Address> addressGrid = new Grid<>(Person.class);
List<Person> list = getPersons();


// populate Grid with a Stream
addressGrid.setItems(list.stream()
    .filter(p -> someFilter(p))
    .map(Person::getAddress).distinct().sorted());
```
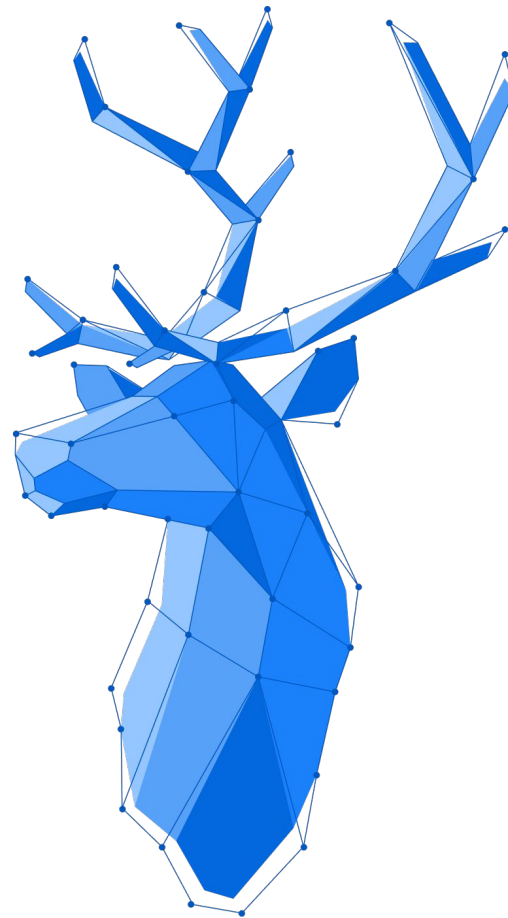
Exercise 1

Populating a Grid

Summary, Part 1

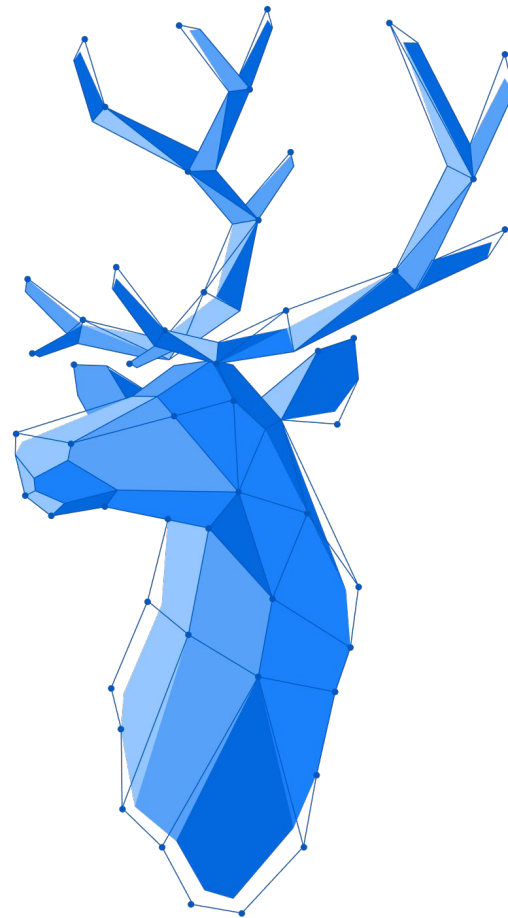- Grid instantiation
- Adding columns
- Column configuration
- Basic data

Data Lists with Grid, Part 2

Grid configuration

vaadin }>

Recap, part 1

- Grid instantiation
- Adding columns
- Basic data

Grid configuration and advanced features

Sorting configuration

Column Sorting

Columns can be made sortable with
`Column::setSortable(true)` if the
data source supports it.

Sorting uses the default Java
comparator for the column type (falling
back to String sorting).

First name ⇅	Last name ⇅	Age ⇅	Address ⇅
Jack	Giles	50	Washington 12080
Nathan	Patterson	20	Washington 12080
Andrew	Bauer	30	New York 12080
Mickael	Blackwell	68	Washington 12080
Peter	Buchanan	38	New York 93849
Samuel	Lee	53	New York 86829
Anton	Ross	27	New York

Custom Column Sorting

An explicit comparator for defining item order can be set with Column::setComparator()

```
Grid<Person> grid = new Grid<>(Person.class);
// Sorting from the first name column sorts by age instead
grid.getColumnByKey("firstName").setComparator(Comparator.comparing(Person::getAge));
```

Note: this only applies for in-memory data; we'll discuss lazy sorting later.

Multi Sorting
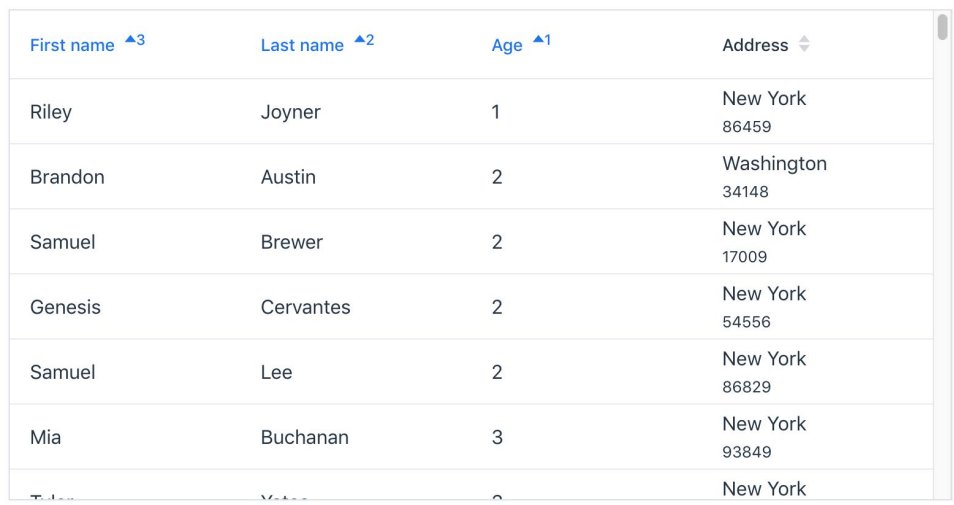
Enable multi-sorting with

```
grid.setMultiSort(true)
```

First name ▲3	Last name ▲2	Age ▲1	Address ⬍
Riley	Joyner	1	New York 86459
Brandon	Austin	2	Washington 34148
Samuel	Brewer	2	New York 17009
Genesis	Cervantes	2	New York 54556
Samuel	Lee	2	New York 86829
Mia	Buchanan	3	New York 93849
Tyler	Yates	3	New York

Multi Sorting

You can also configure the order of multi-sort priority. Default behavior adds new properties to the front of the priority list.

```
// add properties to the
// end of the priority list
grid.setMultiSort(true,
  Grid.MultiSortPriority.APPEND);

// default behavior, pictured right
grid.setMultiSort(true,
  Grid.MultiSortPriority.PREPEND);
```


First name ▲3	Last name ▲2	Age ▲1	Address ⇅
Riley	Joyner	1	New York 86459
Brandon	Austin	2	Washington 34148
Samuel	Brewer	2	New York 17009
Genesis	Cervantes	2	New York 54556
Samuel	Lee	2	New York 86829
Mia	Buchanan	3	New York 93849
Tyler	Yates	3	New York

Prepend significa que se empieza por el orden más importante (el primero)

Dynamic Height

By default, Grid has a height of 400px

First Name	Last Name	Age	Address	Phone Number
Lucas	Kane	68	12080 Washin...	127-942-237
Peter	Buchanan	38	93849 New Yo...	201-793-488
Samuel	Lee	53	86829 New Yo...	043-713-538
Anton	Ross	37	63521 New York	150-813-6462
Aaron	Atkinson	18	25415 Washin...	321-679-8544
Jack	Woodward	28	95632 New York	187-338-588

Automatic height by rows

If you only have a small number items, you may want to make the Grid only take as much height as the rows in it need.

The Grid itself will not get a scrollbar like this, but the containing layout might.

```
// Let the height defined by the number of rows
grid.setAllRowsVisible(true);
```

First Name	Last Name	Age	Address	Phone Number
Lucas	Kane	68	12080 Washin...	127-942-237
Peter	Buchanan	38	93849 New Yo...	201-793-488
Samuel	Lee	53	86829 New Yo...	043-713-538
Anton	Ross	37	63521 New York	150-813-6462
Aaron	Atkinson	18	25415 Washin...	321-679-8544
Jack	Woodward	28	95632 New York	187-338-588

Selection Mode

Single Selection (default)

```
grid.setSelectionMode(
     Grid.SelectionMode.SINGLE);
```

First name	Age
Jack	50
Nathan	20
Andrew	30
Mickael	68
Peter	38
Samuel	53
Anton	37
Aaron	18
Jack	28
Elizabeth	11

Selection Mode

Multi Selection

```
grid.setSelectionMode(
    Grid.SelectionMode.MULTI);
```

	First name	Age
☑	Jack	50
☑	Nathan	20
☐	Andrew	30
☐	Mickael	68
☐	Peter	38
☐	Samuel	53
☐	Anton	37
☐	Aaron	18
☐	Jack	28
☐	Elizabeth	11

Set selected values

You can also make selection programmatically

```
grid.select();
grid.asSingleSelect().setValue();
grid.asMultiSelect().setValue();
```

Get selected values

Note that for multiselection, the value is a Set<T>

```
grid.asSingleSelect().getValue();
grid.asMultiSelect().getValue();
grid.getSelectedItems();
```

Grid as a Field

Grid doesn't implement the HasValue interface, so cannot be used as Field directly. To bind it, you can use grid.asSingleSelect() and grid.asMultiSelect(); both implement HasValue.

```
SingleSelect<Grid<Person>, Person> selected = grid.asSingleSelect();
binder.forField(selected).bind(...);

MultiSelect<Grid<Person>, Person> selected = grid.asMultiSelect();
binder.forField(selected).bind(...);
```
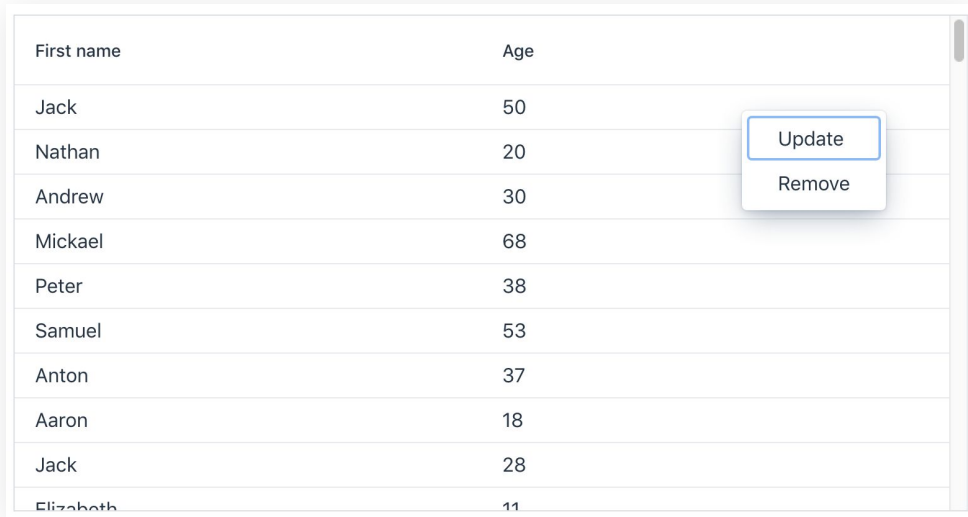
Click Listeners

Grid can listen for both single and double click events. You will get access to the column object and the clicked row bean from the event.

```
grid.addItemClickListener(event -> {
    Grid.Column<Person> column = event.getColumn();  Obtiene la columna
    Person item = event.getItem();   Obtiene la fila
});

grid.addItemDoubleClickListener();
```

Context Menu

You can react to context clicks with a custom menu:

```java
GridContextMenu<Person> contextMenu =
        new GridContextMenu<>(grid);
contextMenu.addItem("Update", e ->{...});
contextMenu.addItem("Remove", e ->{...});
```

First name	Age
Jack	50
Nathan	20
Andrew	30
Mickael	68
Peter	38
Samuel	53
Anton	37
Aaron	18
Jack	28
Elizabeth	11

Update
Remove

Tooltips

You can generate Tooltips on rows and cells:

```
grid.setTooltipGenerator(person -> {
    return "AKA: "+ person.getNickname();
});
```

```
grid.getColumnByKey("birthday")
    .setTooltipGenerator(
      person -> "Age: " + getPersonAge(person)
    );
```

First name	Last name	Birthday
Aria	Bailey	1984-01-13
Aaliyah	Butler	1977- Age: 39

Tooltips

Tooltips have many uses:
- For providing additional details on the contents of a cell
- For providing the full text of a cell if it's too long to fit feasibly into the cell itself
- For providing textual explanations for non-text content, such as status icons

Note that tooltips can only contain plan text, no HTML content.

Item Details

A Grid row can be configured to expand and show extra Item Details. The content can be anything that can be supplied by a Renderer.

```
// Use any renderer for the item details.
grid.setItemDetailsRenderer(...);
```

First name	Age
Jack	50

Hi! My name is **Jack!**

First name	Age
Nathan	20
Andrew	30
Mickael	68
Peter	38
Samuel	53

Item Details

By default, when clicking on a row, two things happen: item is selected, details view is shown. To only show details without selection, use

```
grid.setSelectionMode(Grid.SelectionMode.NONE);
```

Item Details

By default, the details are opened and closed by clicking the rows. To show/hide the details programmatically, use `setDetailsVisible(item, true/false)`

```java
// Disable the default way of opening item details
grid.setDetailsVisibleOnClick(false);
// Open Details from a button
grid.addColumn(new NativeButtonRenderer<>("Details", item -> grid
    .setDetailsVisible(item, !grid.isDetailsVisible(item))));
```

Theme Variants

Use Theme variants to quickly change the look and feel.

```
grid.addThemeVariants(...);

// No borders for the Grid
GridVariant.LUMO_NO_BORDER
// No borders for the rows
GridVariant.LUMO_NO_ROW_BORDER
// Show a column border
GridVariant.LUMO_COLUMN_BORDER
// Show alternating row colors
GridVariant.LUMO_STRIPES
// Less padding
GridVariant.LUMO_COMPACT
// Allow cell contents to grow and wrap
GridVariant.LUMO_WRAP_CELL_CONTENT
// Material-style column divider
GridVariant.MATERIAL_COLUMN_DIVIDERS
```

First name	age
Jack	50
Nathan	20
Andrew	30
Mickael	68
Peter	38
Samuel	53
Anton	37
Aaron	18
Jack	28
Elizabeth	11

Tree Grid

Use TreeGrid to display hierarchical data. Aside from the hierarchy, TreeGrid works the same as a normal Grid.

```
TreeGrid<Person> treeGrid =
    new TreeGrid();
treeGrid.setItems(persons,
    item -> childMap.get(item));
```

Inline editing

Inline Editing

Grid has an Editor for inline editing

`grid.getEditor();`

First name	Subscriber
Jack	☐
Nathan	false
Andrew	false
Mickael	false
Peter	true
Samuel	false
Anton	false
Aaron	false
Jack	false

Inline Editing

Set the component for editing through a Grid.Column

```
TextField field = new TextField();
nameColumn.setEditorComponent(field);

Checkbox checkbox = new Checkbox();
subscriberColumn.setEditorComponent(checkbox);
```

First name	Subscriber
Jack	☐
Nathan	false
Andrew	false
Mickael	false
Peter	true
Samuel	false
Anton	false
Aaron	false
Jack	false

Inline Editing

Use Binder to do data binding

```java
Binder<Person> binder =
        new Binder<>(Person.class);
grid.getEditor().setBinder(binder);
binder.bind(field, "firstName");
binder.bind(checkbox, "subscriber");
```

First name	Subscriber
Jack	
Nathan	false
Andrew	false
Mickael	false
Peter	true
Samuel	false
Anton	false
Aaron	false
Jack	false

Inline Editing

Open editor on double click

```
grid.addItemDoubleClickListener(
    event -> {
        grid.getEditor()
                .editItem(event.getItem());
    }
);
```

First name	Subscriber
Jack	☐
Nathan	false
Andrew	false
Mickael	false
Peter	true
Samuel	false
Anton	false
Aaron	false
Jack	false

Grid Editor

By default, changes reflect to row beans directly. You can also enable Buffered Mode to only save changes when instructed.

```
grid.getEditor().setBuffered(true);
```
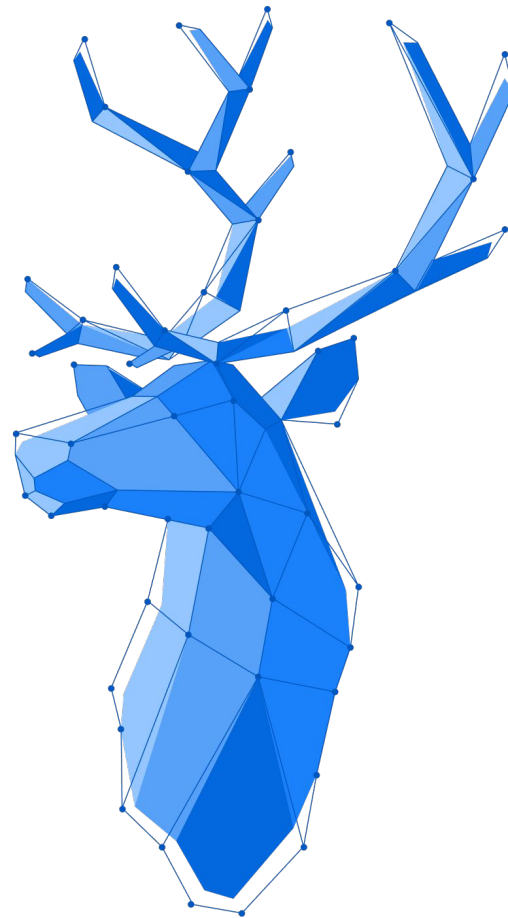
Note: Buffered Mode only changes how the binder works: setBean() vs readBean() and writeBean().

The Edit/Save/Cancel buttons need to be implemented manually. The Editor gives access to the current item and the Binder.

Summary, Part 2

- Sorting configuration
- Sizing
- Selection and listeners
- Tooltips
- Item Details
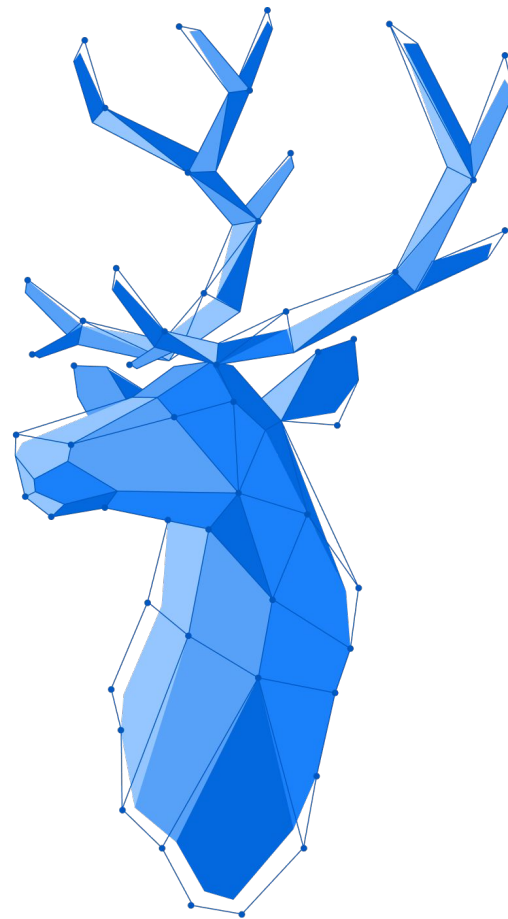- Tree Grid
- Inline editing

Data Lists with Grid, Part 3

In-Memory Data Providers

vaadin]>

Recap, part 1 and 2

- Grid instantiation
- Adding columns
- Grid configuration
- Inline editing

Data providers

Populate data to Grid

What happens when you call grid.setItems()?

```
grid.setItems(items);
```

realmente hace

```
grid.setItems(DataProvider.ofCollection(items));
```

DataProvider interface

DataProvider is a common interface for fetching data from backend, which is used by listing components.

There are In-Memory and Lazy Loading variants

In-Memory DataProvider

Both ListDataProvider and TreeDataProvider are In-Memory DataProviders.

The default DataProvider for all Components when calling setItems() with a List or varargs is in-memory

In-Memory DataProviders support sorting and filtering out of the box.

Sorting and filtering

Sort based on the bean

Usually, we sort through the Component that we use (e.g. a Grid). But you can sort the DataProvider directly if you want. This might make sense if you use the same DataProvider for multiple components, like a Grid and a Chart.

Compare based on the bean:

```
// overrides the previously set comparator
InMemoryDataProvider#setSortComparator(SerializableComparator<T> comparator)
// add a new comparator
InMemoryDataProvider#addSortComparator(SerializableComparator<T> comparator)
```

```
// used like this, note the ::compare
dataProvider.setSortComparator(Comparator.comparing(Person::getName)::compare);
```

Sort based on a property

Usually, we sort through the Component that we use (e.g. a Grid). But you can sort the DataProvider directly if you want. Compare based on a property:

```
// overrides the previously set sort order
InMemoryDataProvider#setSortOrder(ValueProvider<T, V> valueProvider, SortDirection sortDirection)
// add a new sort order
InMemoryDataProvider#addSortOrder(ValueProvider<T, V> valueProvider, SortDirection sortDirection)



// used like this
dataProvider.setSortOrder(Person::getName, SortDirection.ASCENDING);
```

Filtering based on the bean

In-Memory DataProviders support direct filtering. It's possible to filter based on the bean.

```
InMemoryDataProvider#setFilter(SerializablePredicate<T> filter);
```

```
// an example
dataProvider.setFilter(person -> person.getEmail() != null);
```

Filtering based on a value

In-Memory DataProviders support direct filtering. It is possible to filter based on a certain value.

```
InMemoryDataProvider#setFilterByValue(ValueProvider<T, V> valueProvider, V requiredValue);
```

```
// an example
dataProvider.setFilterByValue(Person::getBirthDate, null);
```

Solo mostramos personas con cumpleaños a null

TreeDataProvider

ListDataProvicer needs a List/Collection for the backing data.
TreeDataProvider needs a **TreeData** for the backing data.
 Esto es porque no hay una estructura de datos propia de Tree en Java.

```java
// Get root level projects
Collection<Project> projects = service.getAllProjects();

TreeData<Project> data = new TreeData<>();
// add root level items - parent is null
data.addItems(null, projects);

// add children for the first level of items
projects.forEach(project -> data.addItems(project, project.getChildren()));

// construct the data provider for the hierarchical data we've built
TreeDataProvider<Project> dataProvider = new TreeDataProvider<>(data);
```

Updating data

Update data in a DataProvider

If you update any of the data from a provider, the provider will not know of the change. Whenever you need to refresh a provider, call:

```
dataProvider.refreshAll();

// or:    Para un solo item
dataProvider.refreshItem(item)
```

Update data in a DataProvider

For dataProvider.refreshItem(item) to work properly, the old and new instances should be considered equal. Either they are the same instance, or the class implements hashCode() and equals() methods.

Alternatively, make your own DataProvider which overrides the getId() method.

DataView API

DataView

The Grid#setItems methods all return an implementation of the DataView interface. Every DataView allows you to set an identifier provider, do the refreshItem / refreshAll requests, and fetch an item at a specific index.

The concrete implementation classes such as GridLazyDataView and GridListDataView offer more methods for accessing and possibly modifying the data depending on how the actual loading of the data works.

DataView

For example, it's possible to get the next or previous entry for a specific item with the DataView. The following example shows how you can programmatically select the next item in a Grid:

```java
List<Person> allPersons = repo.findAll();
GridListDataView<Person> gridDataView = grid.setItems(allPersons);

Button selectNext = new Button("Next", e -> {
    grid.asSingleSelect().getOptionalValue().ifPresent(p -> {
        gridDataView.getNextItem(p).ifPresent(
            next -> grid.select(next)
        );
    });
});
```

DataView

You can also use the DataView for either manipulating the content of the underlying data collection, or to refresh the UI if one of the items has been changed:

```
GridListDataView<Person> gridDataView = grid.setItems(allPersons);
gridDataView.addItem(new Person(...));
gridDataView.refreshAll(); // this will update the UI in the browser, showing the new item

...

gridDataView.removeItem(removedPerson);
gridDataView.refreshAll(); // this will update the UI in the browser - removing one row

...

updatedPerson.setEmail("newemail@vaadin.com");
gridDataView.refreshItem(updatedPerson); // this will show new email for this row in the browser
```
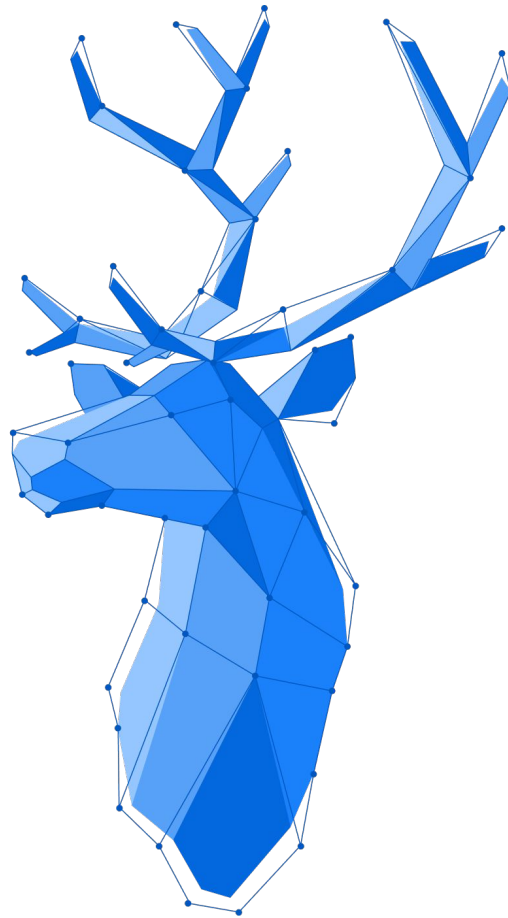
Exercise 2

Filtering a DataProvider

Summary, Part 3

- Data providers
- Sorting and filtering
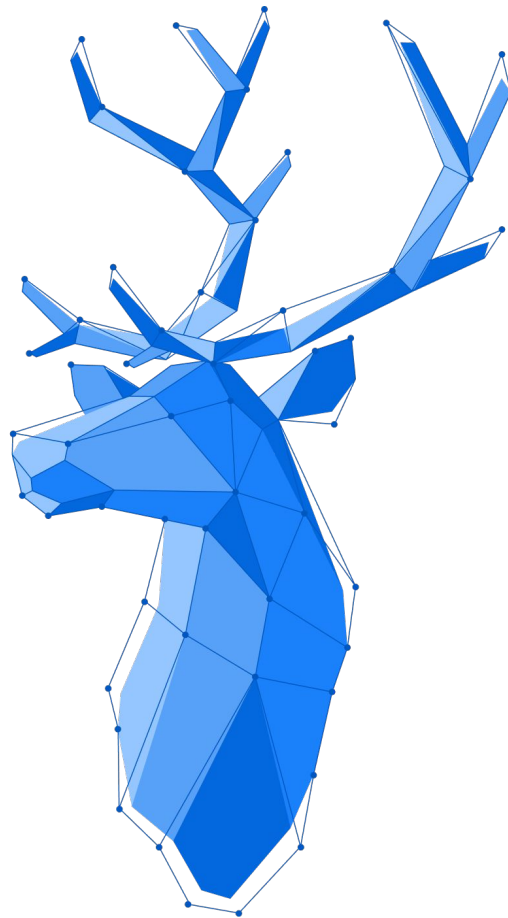- Updating data
- DataView API

Data Lists with Grid, Part 4

Lazy Data Providers

vaadin]>

Recap, parts 1-3

- Grid instantiation
- Adding columns
- Grid configuration
- Inline editing
- Data providers
- Sorting and filtering
- Updating data
- DataView API

Lazy Data Providers

What information do you need for lazy loading?

Query object

A Query object is provided to you with information needed for lazy loading. Offset for the index of the first item to be retrieved; Limit is the number of items to be retrieved.

```
public class Query<VALUETYPE, FILTERTYPE> {
  public int getLimit();
  public int getOffset();
  public List<QuerySortOrder> getSortOrders();
  public Optional<FILTERTYPE> getFilter();
}
```

FetchCallback

Instead of passing all the data to a DataProvider directly, lazy loading needs to pass a callback which fetches a stream of items based on a query.

The fetch callback will be called when there is a need to load more data from the backend, e.g. when user scrolls a Grid.

```java
// API
public interface FetchCallback<VALUETYPE, FILTERTYPE> {
    Stream<VALUETYPE> fetch(Query<VALUETYPE, FILTERTYPE> query);
}

// Example
FetchCallback<Person, Void> fetchCallback =
    query -> service.getPersons(query.getOffset(), query.getLimit());
```

CountCallback

In addition to a FetchCallBack, a CountCallBack is also needed. CountCallback counts the number of items in the backend. It's needed to be able to render the scroll bar correctly.

```java
// API
public interface CountCallback<VALUETYPE, FILTERTYPE> extends Serializable {
    int count(Query<VALUETYPE, FILTERTYPE> query);
}


// Example
CountCallback<Person, Void> countCallback = query -> backend.countPersons();
```

AbstractBackendDataProvider

An abstract superclass that requires you to implement two methods: one for **fetching** items from a back end, the other for **counting** the number of available items.

```java
public class PersonBackendDataProvider extends AbstractBackEndDataProvider<Person, Void> {

    private PersonService backend = new PersonService();

    @Override
    protected Stream<Person> fetchFromBackEnd(Query<Person, Void> query) {
        return backend.getPersons(query.getLimit(), query.getOffset());
    }

    @Override
    protected int sizeInBackEnd(Query<Person, Void> query) {
        return backend.countPersons();
    }
}
```

CallbackDataProvider

You can also implement a lazy dataprovider more compactly by providing **two callbacks**: a **FetchCallback** and a **CountCallback**. They do the same things as AbstractBackendDataProvider's required methods.

```
// API
DataProvider.fromCallbacks(
        FetchCallback<T, Void> fetchCallback,
        CountCallback<T, Void> countCallback)
);


// Example
CallbackDataProvider<Person, Void> dataProvider = DataProvider.fromCallbacks(
        query -> backend.getPersons(query.getOffset(), query.getLimit()),
        query -> backend.countPersons()
);
```

Count callbacks are optional

If you don't know the exact amount of items in your backend, or it's costly to get the count, you can omit the count callback. When a user scrolls to the end of the scrollable area, Grid polls your callbacks for more items. If new items are found, these are added to the component. This causes the scrollbar to jump a bit as new items are added on the fly.

The user experience can be improved by providing an estimate of the actual number of items in the binding code through the DataView.

```java
GridLazyDataView<Person> dataView = grid.setItems(query -> {
    return getPersonService()
            .fetchPersons(query.getOffset(), query.getLimit())
            .stream();
});

dataView.setItemCountEstimate(1000);
dataView.setItemCountEstimateIncrease(500);
```

Lazy Filtering and Sorting

Filtering CallbackDataProviders

Use DataProvider.fromFilteringCallbacks() to create a CallbackDataProvider that supports filtering. The generic parameter F determines the type of the Filter.

```
// API
DataProvider.fromFilteringCallbacks(
    FetchCallback<T, F> fetchCallback,
    CountCallback<T, F> countCallback)
);


// Same example as before with fromFilteringCallbacks. This does not yet do any filtering.
DataProvider<Person, String> dataProvider = DataProvider.fromFilteringCallbacks(
    query -> backend.getPersons(query.getOffset(), query.getLimit()),
    query -> backend.countPersons()
);
```

Filtering Lazy Data

The callbacks need to use the filter. Note that query.getFilter() return an Optional.

```java
// Now filter information is passed to the backend
DataProvider<Person, String> dataProvider = DataProvider.fromFilteringCallbacks(
    query -> backend.getPersons(
                    query.getOffset(),
                    query.getLimit(),
                    query.getFilter().orElse(null)
                ),
    query -> backend.countPersons(query.getFilter().orElse(null))
);
```

Filtering Lazy Data

setFilter() API is only available in In-Memory DataProviders.

```java
// Now filter information is passed to the backend
DataProvider<Person, String> dataProvider = DataProvider.fromFilteringCallbacks(
    query -> backend.getPersons(
                    query.getOffset(),
                    query.getLimit(),
                    query.getFilter().orElse(null)
            ),
    query -> backend.countPersons(query.getFilter().orElse(null))
);

// No such API, won't compile
dataProvider.setFilter();
```

Filtering Lazy Data

To be able to set a filter to other DataProviders, e.g. a CallbackDataProvider, need to first wrap into a ConfigurableDataProvider by using **withConfigureableFilter()** method.

```java
// Now filter information is passed to the backend
DataProvider<Person, String> dataProvider = DataProvider.fromFilteringCallbacks(
      query -> backend.getPersons(
                          query.getOffset(),
                          query.getLimit(),
                          query.getFilter().orElse(null)
                    ),
      query -> backend.countPersons(query.getFilter().orElse(null))
);


ConfigurableFilterDataProvider<Person, Void, String> filteredDataProvider =
dataProvider.withConfigurableFilter();
```

El genérico de enmedio siempre va a ser Void. El último es el tipo del Filtro.

Filtering Lazy Data

setFilter() API is now available in the ConfigurableDataProvider.

```java
// Now filter information is passed to the backend
DataProvider<Person, String> dataProvider = DataProvider.fromFilteringCallbacks(
    . . .
);


ConfigurableFilterDataProvider<Person, Void, String> filteredDataProvider =
dataProvider.withConfigurableFilter();

TextField textField = new TextField();
textField.addValueChangeListener(event -> filteredDataProvider.setFilter(event.getValue()));
```

Sorting Lazy Data

It makes no sense to sort this on the front end, since we don't have all of the data. Ergo, you need to sort the data in the backend, inside the callbacks.

```java
public Stream<Consultant> getPersons(Query<Consultant, Void> query) {
    List<QuerySortOrder> sorting = query.getSortOrders();

    // add sort info to your data fetch
}
```

Sorting Lazy Data

In the backend, use query to get the sort orders.

```java
public Stream<Consultant> getPersons(Query<Consultant, Void> query) {
    List<QuerySortOrder> sorting = query.getSortOrders();

    // for each order, add a sort command to e.g. your SQL

    QuerySortOrder<String> order1 = sorting.get(0);
    String propertyName = order1.getSorted();
    SortDirection direction = order1.getDirection();
    …
}
```

Sorting Lazy Data (from Grid)

The sort type is defined by the Component, usually a String. If a column has a key, that will be used as the default. Overriding works too:

```java
// Simple columns:
grid.addColumn(Consultant::getSalary).setSortProperty("salary");


// Combined columns:
grid.addColumn(c -> c.getFirstName() + " " + c.getLastName())
    .setSortProperty("lastName", "firstName");
```

Paged Data

Paged data repositories

Some repositories fetch data by Pages. The Query object contains methods for getting page index and page size for these cases.

```
grid.setItems(query ->
  service.getPageData(query.getPage(), query.getPageSize()).stream());
```

Spring Data helpers

If you're using a paged Spring Data based backend, you can use methods in VaadinSpringDataHelpers class. Using the fromPagingRepository() shorthand method, you can create a pageable sortable data binding directly to your repository.

```
// simple one-liner to use a paging Spring Data repository
grid.setItems(VaadinSpringDataHelpers.fromPagingRepository(personRepository));
```

Spring Data helpers

VaadinSpringDataHelpers methods toSpringPageRequest() and toSpringDataSort() convert Vaadin specific query hints to their Spring Data relatives. The fromPagingRepository() shorthand uses these methods under the hood.

```java
// create Pageable request manually, use helpers to create the Sort object for Spring Data
grid.setItems(query -> service.list(
  PageRequest.of(query.getPage(), query.getPageSize(),
    VaadinSpringDataHelpers.toSpringDataSort(query)))
.stream());

// search repository manually, use helpers to create a Page Request with the Query
grid.setItems(query -> personRepository.findAll(
    VaadinSpringDataHelpers.toSpringPageRequest(query)).stream()
);
```

Hierarchical Lazy Data

Hierarchical Lazy Data

Make a class that extends from AbstractBackEndHierarchicalDataProvider, which takes two generic parameters, one for the data type, one for the filter type.

```java
// Base class
public abstract class AbstractBackEndHierarchicalDataProvider<T, F>



// Example
public class LazyTreeDataProvider extends AbstractBackEndHierarchicalDataProvider<Person, String> {

}
```

Hierarchical Lazy Data

Need to implement three methods. The fetch and count callbacks now based on HierachicalQuery

```java
public class LazyTreeDataProvider extends AbstractBackEndHierarchicalDataProvider<Person, String> {
    @Override
    protected Stream<Person> fetchChildrenFromBackEnd(HierarchicalQuery<Person, String> query) {
        ...
    }

    @Override
    public int getChildCount(HierarchicalQuery<Person, String> query) {
        ...
    }

    @Override
    public boolean hasChildren(Person item) {
        ...
    }
}
```

HierarchicalQuery

HierarchicalQuery extends from Query, it also has getParent() API for getting the parent node.

```
public class HierarchicalQuery<T, F> extends Query<T, F> {
    public T getParent();
}
```

Hierarchical Lazy Data

All three methods are based on a certain node, which you can get from
HierachicalQuery::getParent().
- The **count** method returns the number of **immediate** child items.
- The **fetch** method returns the **immediate** child items based on offset and limit.
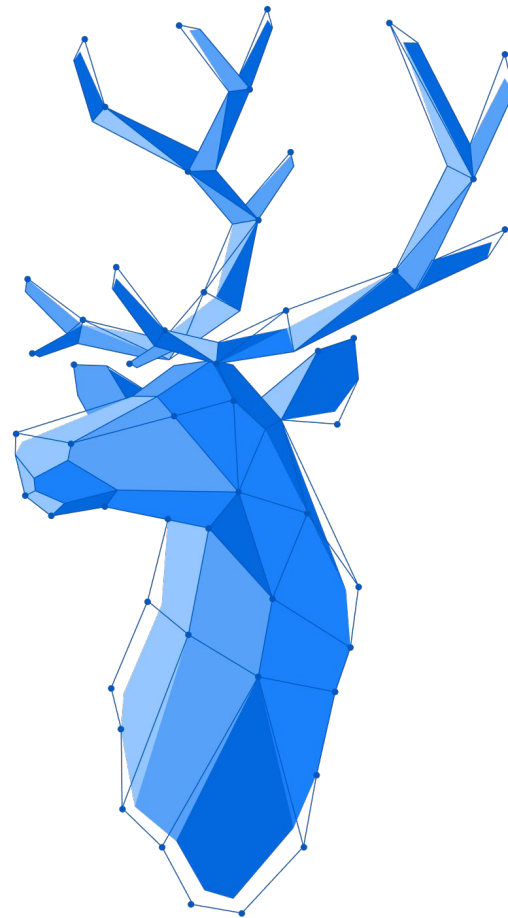- The **hasChildren** method is used for checking if a given item should be expandable.

```java
public class LazyTreeDataProvider extends AbstractBackEndHierarchicalDataProvider<Person, String> {

  protected Stream<Person> fetchChildrenFromBackEnd(HierarchicalQuery<Person, String> query)

  public int getChildCount(HierarchicalQuery<Person, String> query)

  public boolean hasChildren(Person item)
}
```

Exercise 3

Filtering a back end data provider

Summary, Part 4

- Lazy data providers
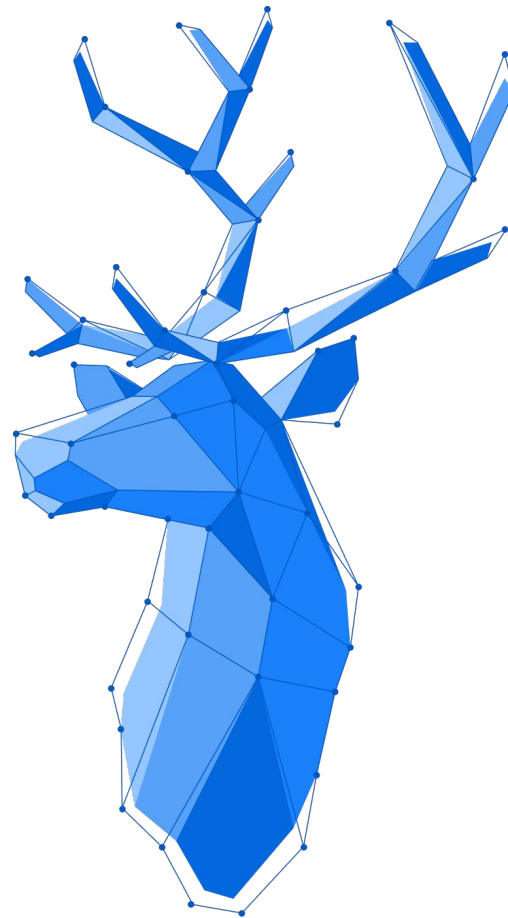- Lazy filtering and sorting
- Paged data
- Hierarchical lazy data

Data Lists with Grid, Part 5

Grid Pro

Recap, parts 1 - 4

- Grid configuration
- Inline editing
- Data providers
- Lazy-loaded data

Grid Pro

What is Grid Pro?

A **commercial** extension of the Grid component.

```
public class GridPro<E> extends Grid<E>
```

Inline editing

Grid editor only supports whole **row** editing.

Grid Pro supports individual **cell** editing and **keyboard navigation**.

Editable Columns

Add an editable column with gridPro.**addEditColumn()**

```java
GridPro<Person> grid = new GridPro<>();

grid.addEditColumn(Person::getName);
```

Editable Columns

addEditColumn() vs addColumn()

```java
GridPro<Person> grid = new GridPro<>();

// addColumn() returns a regular Grid Column
Grid.Column<Person> nameColumn = grid.addColumn(Person::getName);

// addEditColumn() returns a EditColumnConfigurator, which is not a Column
EditColumnConfigurator<Person> emailEditColumnConfigurator = grid.addEditColumn(Person::getEmail);
```

Edit Column Configuration

Use EditColumnConfigurator to config an editor for the content

```java
GridPro<Person> grid = new GridPro<>();

//A text editor
grid.addEditColumn(Person::getEmail).text(..);

//A select editor
grid.addEditColumn(Person::getEmail).select(..);

//A checkbox editor
grid.addEditColumn(Person::isSubscriber).checkbox(..);

//A custom editor!!!
grid.addEditColumn(Person::getEmail).custom(..);
```

Updating data

When configuring an editor, need to pass in a callback function to be called when the item is changed. DataProvider.refreshItem() will be called automatically after the callback.

```java
GridPro<Person> grid = new GridPro<>();

//The item updater receives two arguments: item, newValue.
grid.addEditColumn(Person::getEmail)
    .text((person, newEmail) -> person.setEmail(newEmail));
```

Column Configuration

The editor configuring method returns a Column.

```java
GridPro<Person> grid = new GridPro<>();


Grid.Column<Person> column =
        grid.addEditColumn(Person::getEmail)
                .text((person, newEmail) -> person.setEmail(newEmail));

column.setHeader("Email");
```
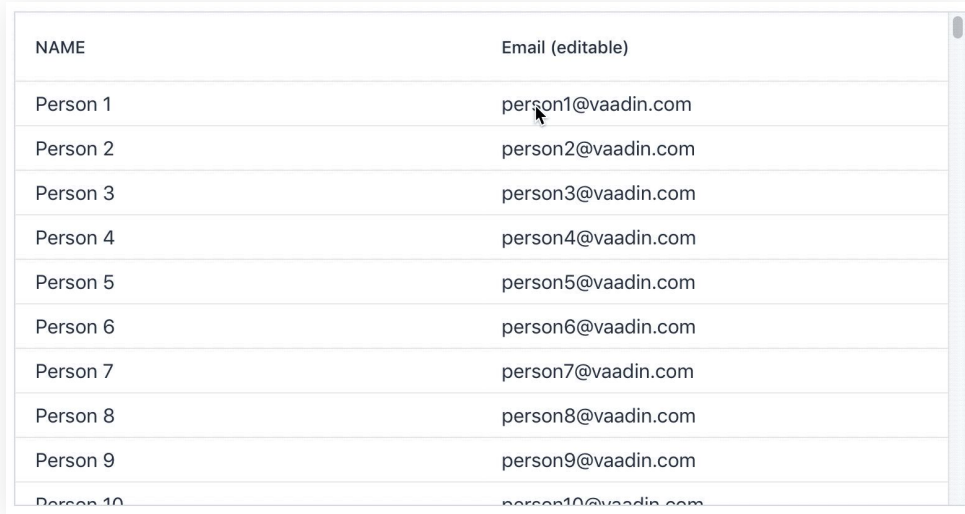
Inline Editing

Put everything together

```java
GridPro<Person> grid = new GridPro<>();
grid.addEditColumn(Person::getEmail)
    .text((item, newValue) ->
        item.setEmail(newValue))
    .setHeader("Email (editable)");
```

Start editing

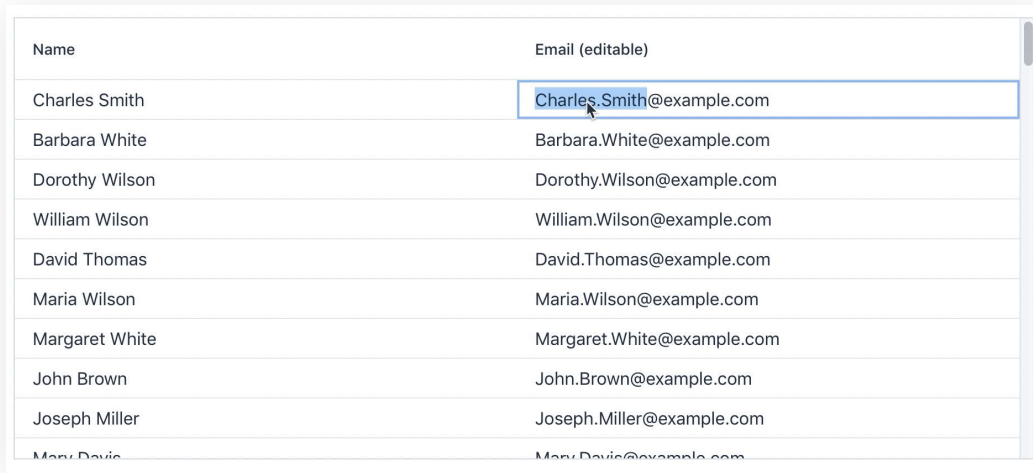To edit a cell, either **double click** or press the **Enter** key

NAME	Email (editable)
Person 1	person1@vaadin.com
Person 2	person2@vaadin.com
Person 3	person3@vaadin.com
Person 4	person4@vaadin.com
Person 5	person5@vaadin.com
Person 6	person6@vaadin.com
Person 7	person7@vaadin.com
Person 8	person8@vaadin.com
Person 9	person9@vaadin.com
Person 10	person10@vaadin.com

Tab Navigation

When in edit mode

Tab: move to next edit cell

Shift+Tab: move to previous
edit cell

| Name	Email (editable)
Charles Smith	Charles.Smith@example.com
Barbara White	Barbara.White@example.com
Dorothy Wilson	Dorothy.Wilson@example.com
William Wilson	William.Wilson@example.com
David Thomas	David.Thomas@example.com
Maria Wilson	Maria.Wilson@example.com
Margaret White	Margaret.White@example.com
John Brown	John.Brown@example.com
Joseph Miller	Joseph.Miller@example.com
Mary Davis	Mary.Davis@example.com

Enter Navigation

You can also make the Enter key to navigate to next cell in edit mode:

```
grid.setEnterNextRow(true);
```

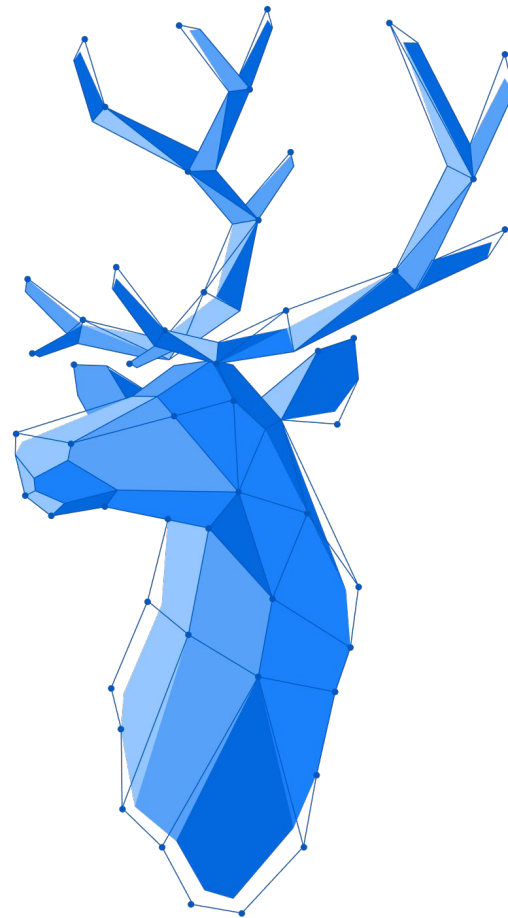| Name	Email (editable)
Maria Wilson	Maria.Wilson@example.com
Christopher Miller	Christopher.Miller@example.com
Richard Miller	Richard.Miller@example.com
Patricia Taylor	Patricia.Taylor@example.com
Richard Wilson	Richard.Wilson@example.com
Charles Jones	Charles.Jones@example.com
Dorothy Taylor	Dorothy.Taylor@example.com
Lisa Wilson	Lisa.Wilson@example.com
Lisa Miller	Lisa.Miller@example.com
Lisa Taylor	Lisa.Taylor@example.com

Summary, Part 5

- Grid Pro
- Inline Editing
- Editable Columns
- Navigation
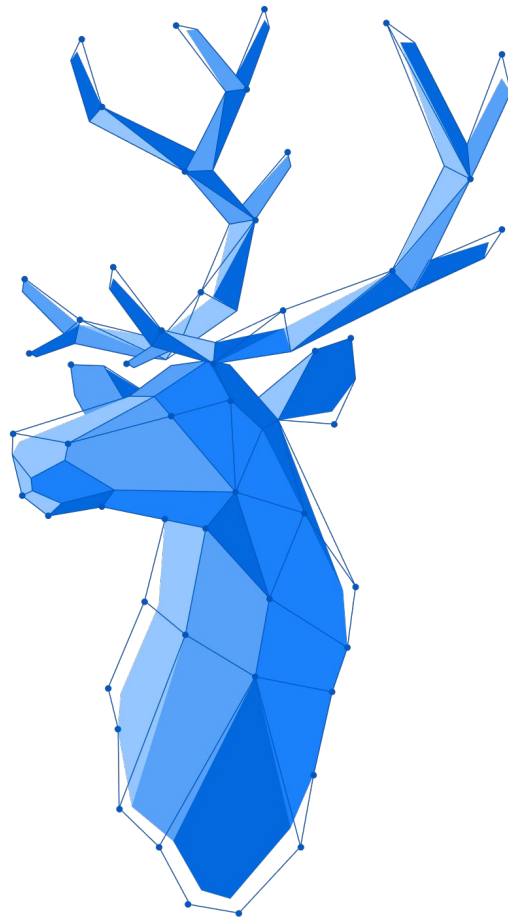
Data Lists with Grid Summary

- Grid
- In-Memory Data Provider
- Lazy Data Provider
- Grid Pro

Thank you!