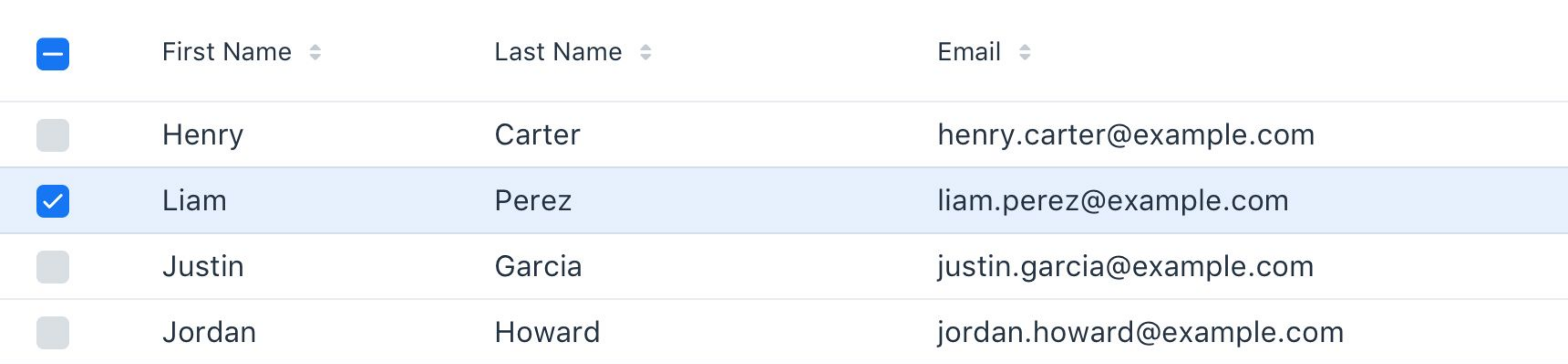
DataLists with Grid

Vaadin 14

Agenda

- Grid
- Exercise 1
- In-Memory Data Provider
- Exercise 2
- Lazy Data Provider
- Exercise 3
- Grid Pro

Grid



⊟	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

Bean Grid: Columns are generated automatically

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

// Possible to reorder generated columns
grid.setColumns("name", "email");
```

Grid instantiation

No class info, nothing to scan, need to add columns manually

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

Adding columns

With a ValueProvider

API

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

Example

```
grid.addColumn(Person::getName)
```

Adding columns

With a property name (**only works with bean Grid**)

API

```
// Or using property name
public Column<T> addColumn(String propertyName)
```

Example

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

Adding columns

With a renderer

API

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

Built-in Renderers

ComponentRenderer

TextRenderer

LocalDateRenderer

NumberRenderer

LocalDateTimeRenderer

NativeButtonRenderer

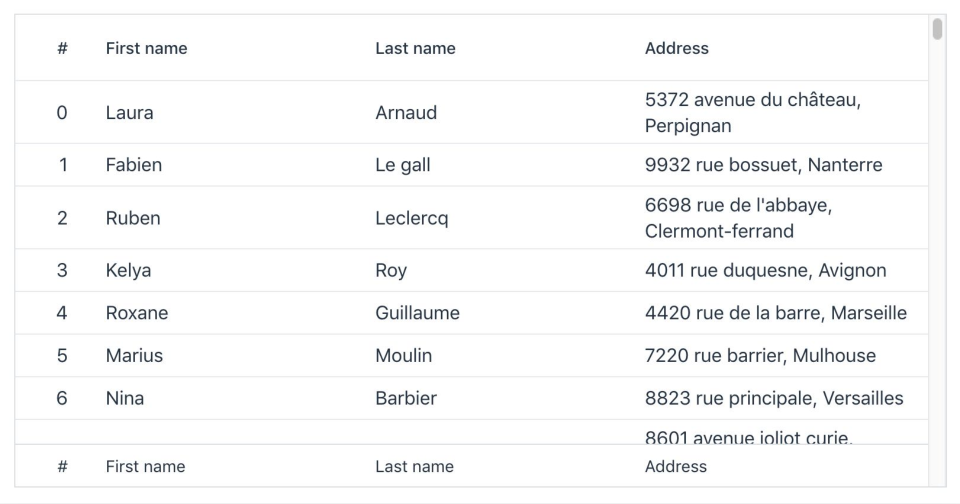
IconRenderer

TemplateRenderer

ColumnPathRenderer

Examples

```
//Use TemplateRenderer to render the index
grid.addColumn(
     TemplateRenderer.of("[[index]]"))
     .setHeader("#");
```

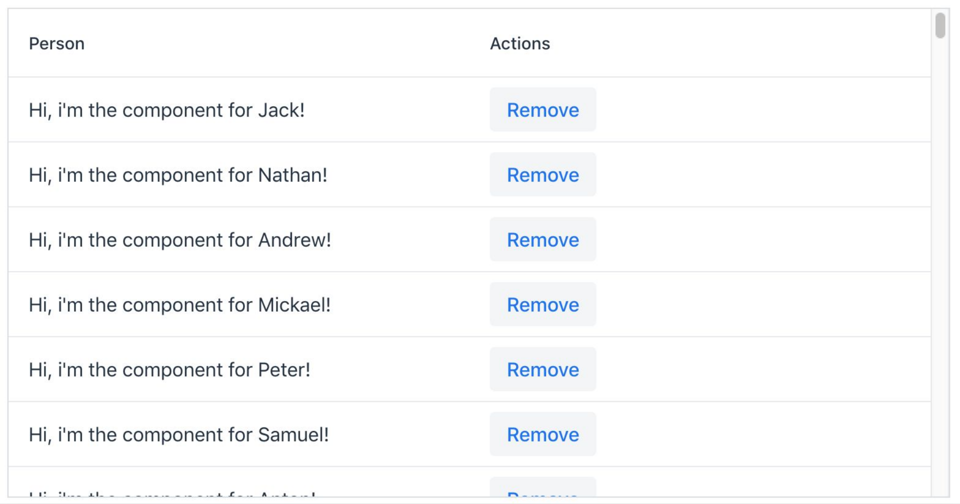


#	First name	Last name	Address
0	Laura	Arnaud	5372 avenue du château, Perpignan
1	Fabien	Le gall	9932 rue bossuet, Nanterre
2	Ruben	Leclercq	6698 rue de l'abbaye, Clermont-ferrand
3	Kelya	Roy	4011 rue duquesne, Avignon
4	Roxane	Guillaume	4420 rue de la barre, Marseille
5	Marius	Moulin	7220 rue barrier, Mulhouse
6	Nina	Barbier	8823 rue principale, Versailles
			8601 avenue joliot curie,
#	First name	Last name	Address

Examples

```
//Render a remove button
grid.addColumn(new ComponentRenderer<>(
     item -> new Button("Remove")));
```

```
//An alternative
grid.addComponentColumn(item->
   new Button("Remove"));
```



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
Hi, i'm the component for Anton!	Remove

Column - Header

Column header is generated automatically when Grid is created with class parameter, e.g. new Grid<Person>(Person.class)

Person.name -> Name

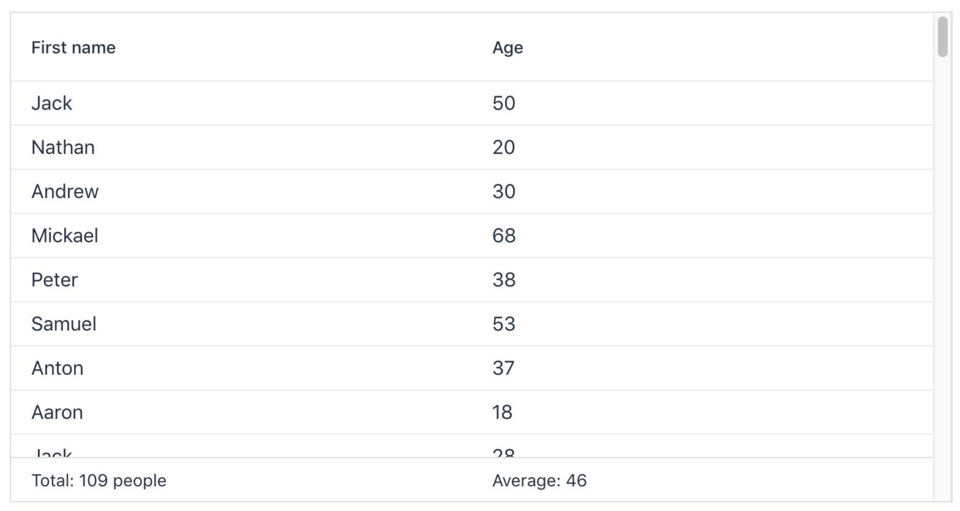
Column - Header

Column::setHeader() is needed for manually added column.

```
grid.addColumn(Person::getName).setHeader("Customer Name");
```

Column Footer

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

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

Column Key is used for retrieving a column i.e. Grid::getColumnByKey()

Column - Key

key is generated automatically, which is the property name if the grid was created with class parameter, e.g. new Grid<Person>(Person.class)

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

Column - Key

key has to be set explicitly with Column::setKey when the grid was created without class parameter, e.g. new Grid<Person>()

```
Grid<Person> grid = new Grid<>();
grid.addColumn(Person::getName).setKey("name");
grid.getColumnByKey("name");
```

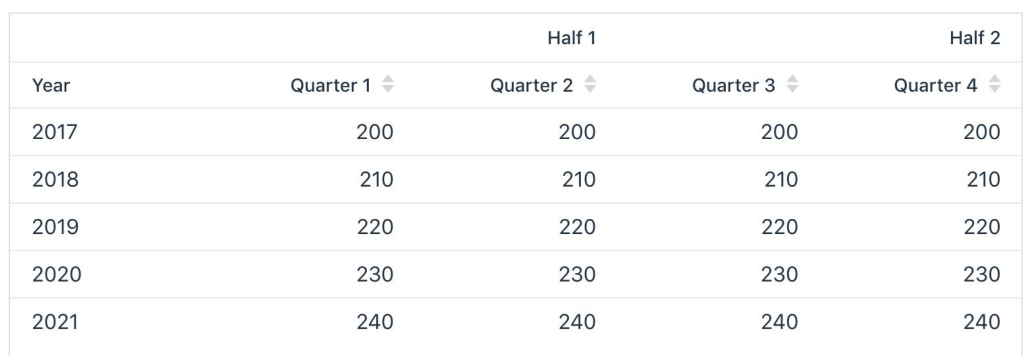
Column - Key

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

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

Column Grouping

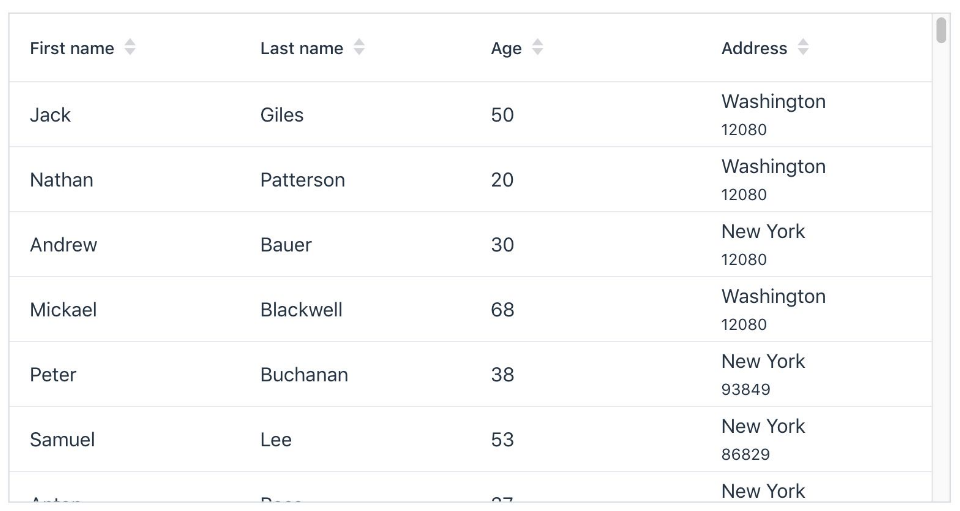
```
HeaderRow halfheaderRow = grid.prependHeaderRow();

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

Column Sorting

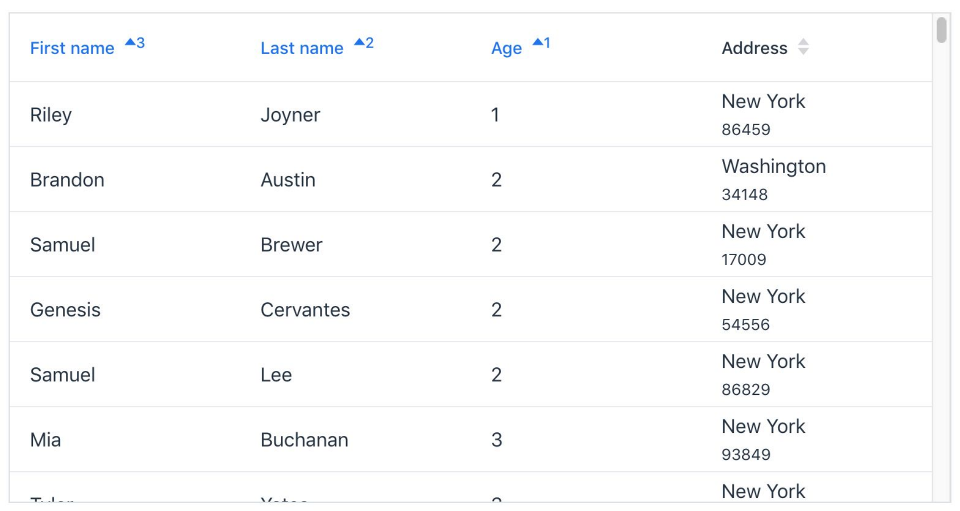
Column::setSortable(true) will enable sorting from UI, which uses a default comparator



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
			New York

Multi Sorting

```
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

Column Sorting

An explicit comparator can be set with Column::setComparator()

```java
Grid<Person> grid = new Grid<>(Person.class);
grid.getColumnByKey("name").setComparator(Comparator.comparing(Person::getName));
```

Populate Data to Grid

Use grid.setItems()

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

//With collection
grid.setItems(list);
```

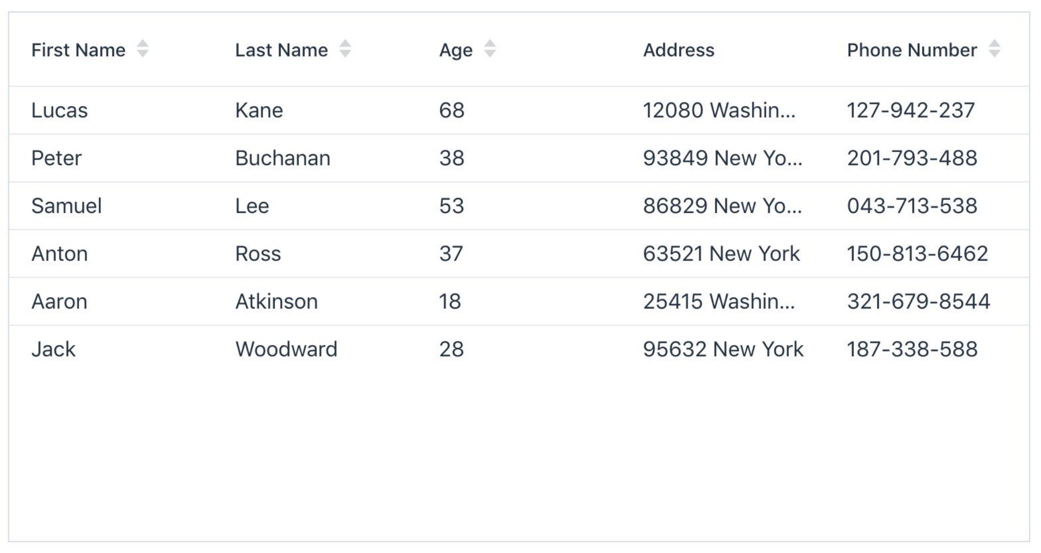
Populate Data to Grid

Use grid.setItems()

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


// With Stream
addressGrid.setItems(list.stream()
     .filter(p -> someFilter(p))
     .map(Person::getAddress).distinct().sorted());
```

Dynamic Height

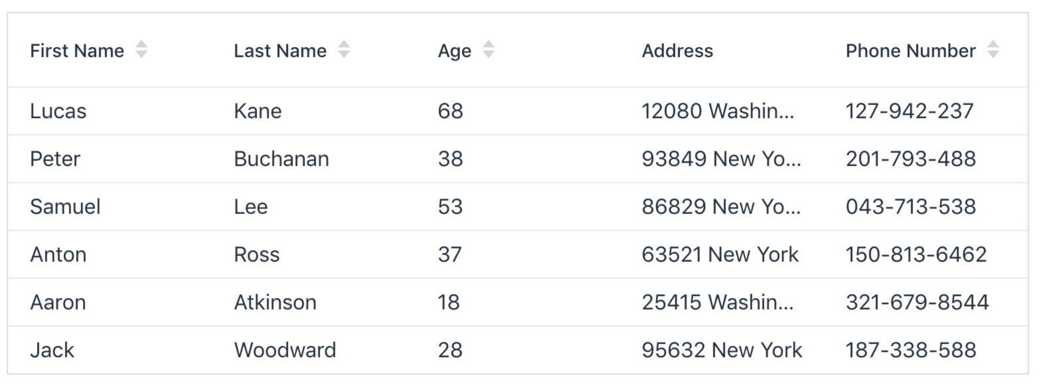
By default, Grid has a height of 400px



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

Dynamic Height

```
// Let the height defined by the number of rows
addressGrid.setHeightByRows(true);
```

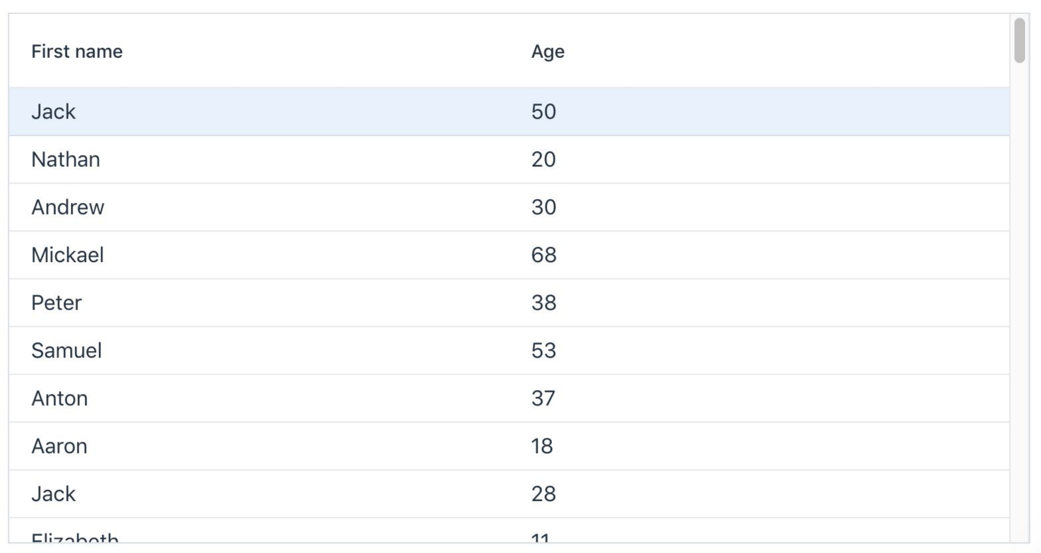


First Name	Last Name	Age	Address	Phone Number
Lucas	Kane	68	12080 Washin...	127-942-237
Peter	Buchanan	38	93849 New Yo...	201-793-488
Samuel	Lee	53	86829 New Yo...	043-713-538
Anton	Ross	37	63521 New York	150-813-6462
Aaron	Atkinson	18	25415 Washin...	321-679-8544
Jack	Woodward	28	95632 New York	187-338-588

Selection Mode

Single Selection (default)

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

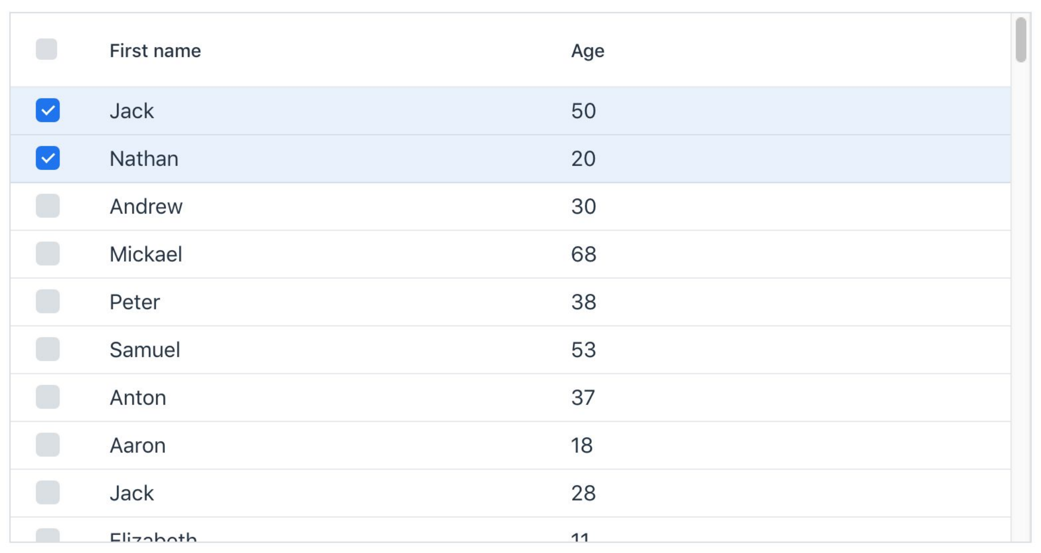


First name	Age
Jack	50
Nathan	20
Andrew	30
Mickael	68
Peter	38
Samuel	53
Anton	37
Aaron	18
Jack	28
Elizabeth	11

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

Selection

To Select

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

Selection

To get selected value. Note that for multiselction, the value is a Set<T>, which is unordered.

```
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. But can get as a SingleSelct or MultiSelect, 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

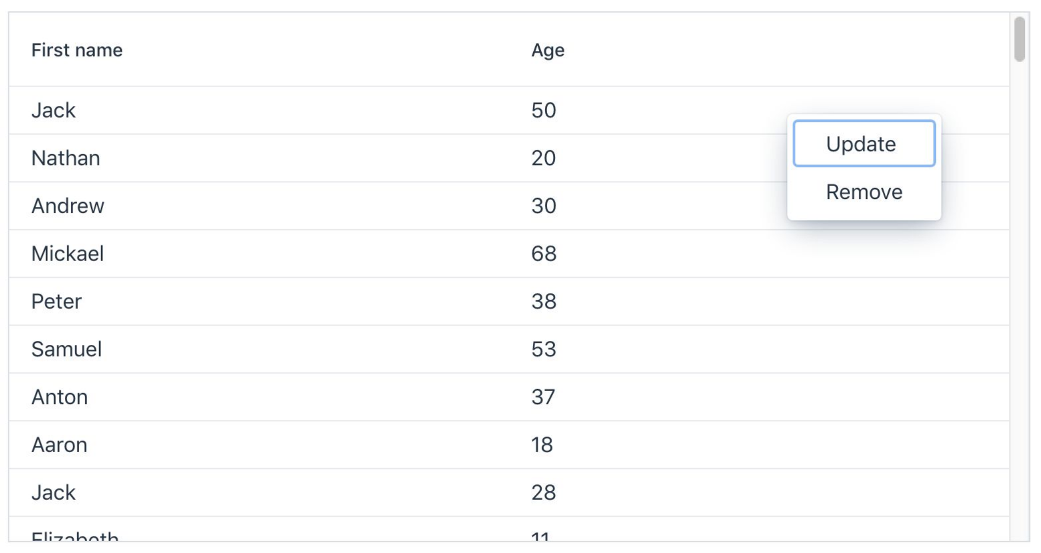
Grid can listen for both single and double click events

```
grid.addItemClickListener();
grid.addItemDoubleClickListener();
```

Context Menu

You like right-click?

```
GridContextMenu<Person> contextMenu =
     new GridContextMenu<>(grid);
contextMenu.addItem("Update", ...);
contextMenu.addItem("Remove", ...);
```

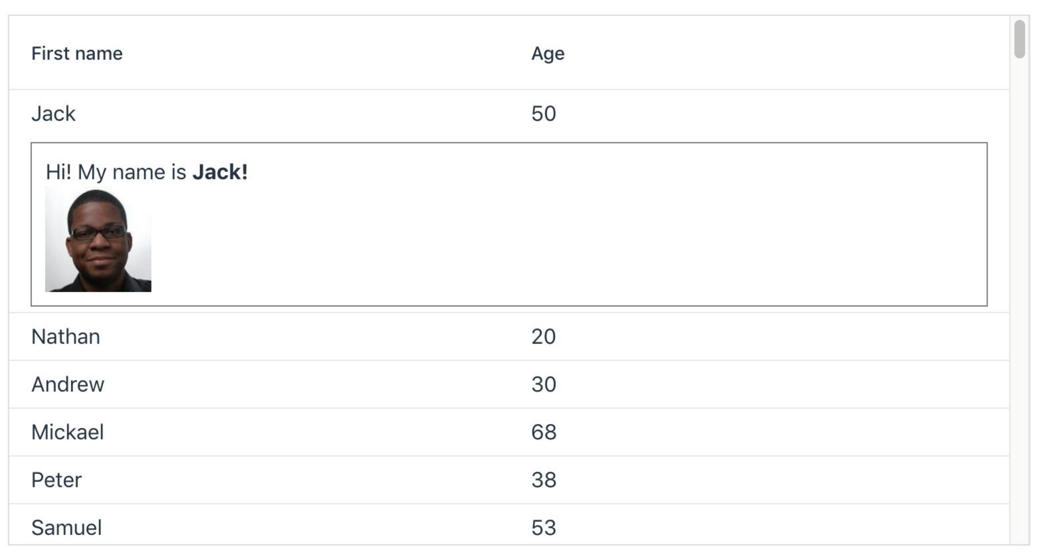


First name	Age
Jack	50
Nathan	20
Andrew	30
Mickael	68
Peter	38
Samuel	53
Anton	37
Aaron	18
Jack	28
Elizabeth	11

Item Details

Sometimes, instead of double-click or right-click, a better way would be to show a details view on click.

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



First name	Age
Jack	50
Hi! My name is **Jack!**	
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

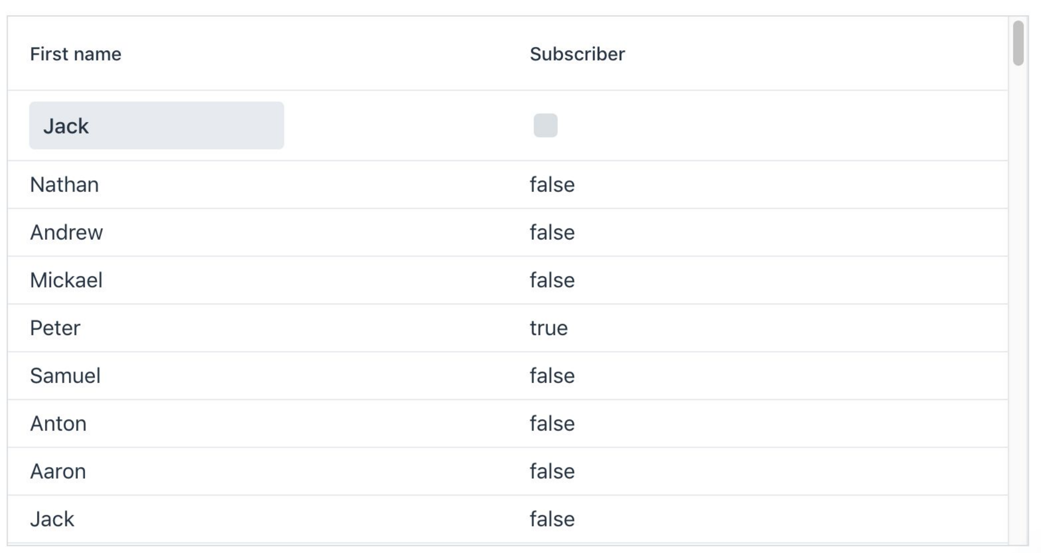
By default, the details are opened and closed by clicking the rows. To show/hide the details programmatically, use

```
// Disable the default way of opening item details:
grid.setDetailsVisibleOnClick(false);

grid.addColumn(new NativeButtonRenderer<>("Details", item -> grid
     .setDetailsVisible(item, !grid.isDetailsVisible(item))));
```

Inline Editing

Grid has an Editor for inline editing

```
grid.getEditor();
```

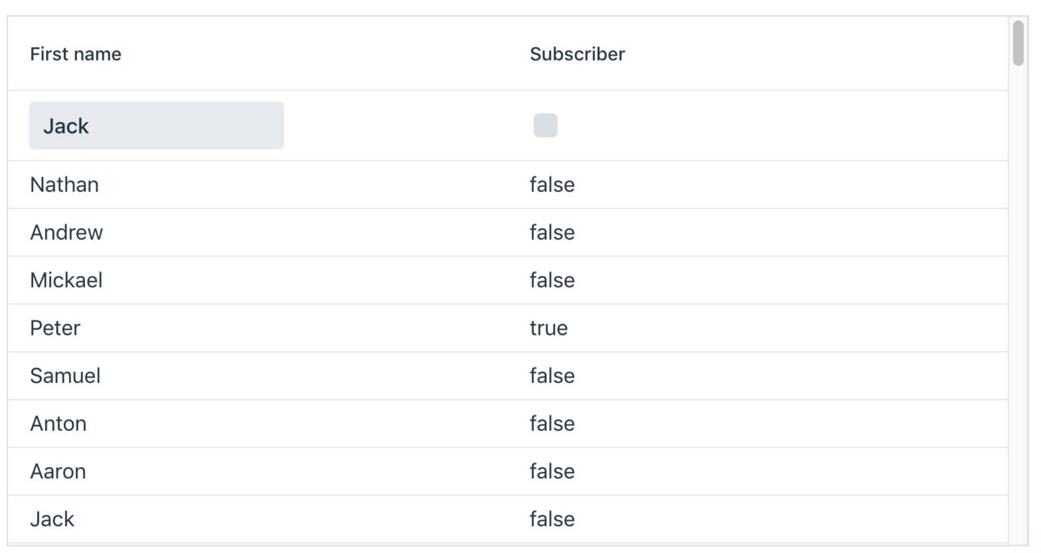


First name	Subscriber
Jack	□
Nathan	false
Andrew	false
Mickael	false
Peter	true
Samuel	false
Anton	false
Aaron	false
Jack	false

Inline Editing

Set a component for editing a Column

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

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

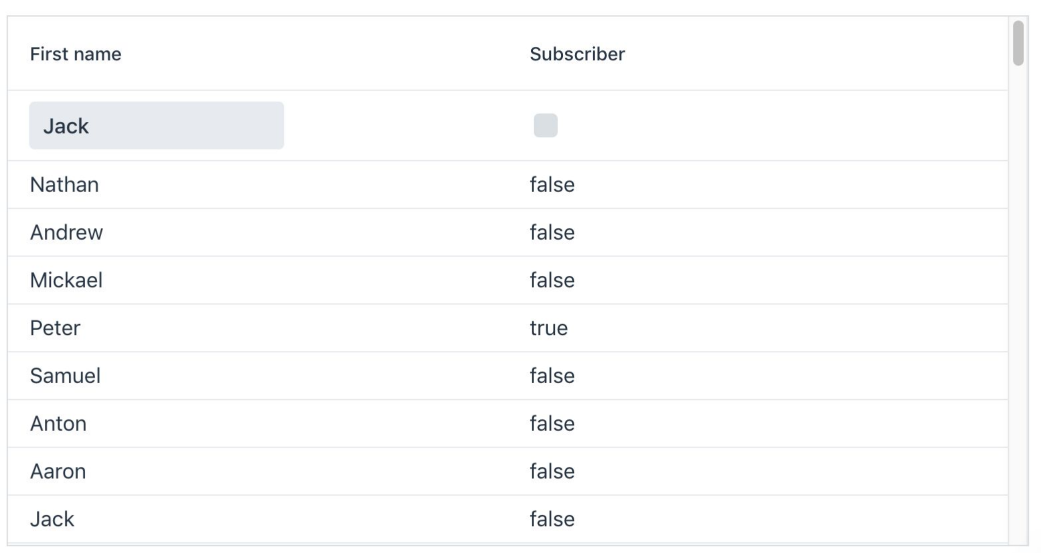


First name	Subscriber
Jack	□
Nathan	false
Andrew	false
Mickael	false
Peter	true
Samuel	false
Anton	false
Aaron	false
Jack	false

Inline Editing

Use Binder to do data binding

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

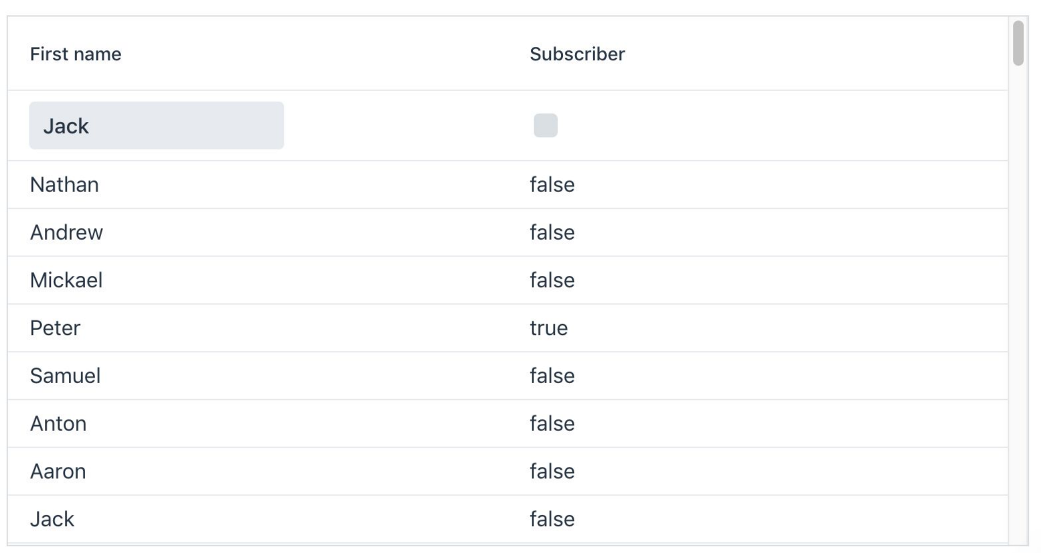


First name	Subscriber
Jack	□
Nathan	false
Andrew	false
Mickael	false
Peter	true
Samuel	false
Anton	false
Aaron	false
Jack	false

Inline Editing

Open editor on double click

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



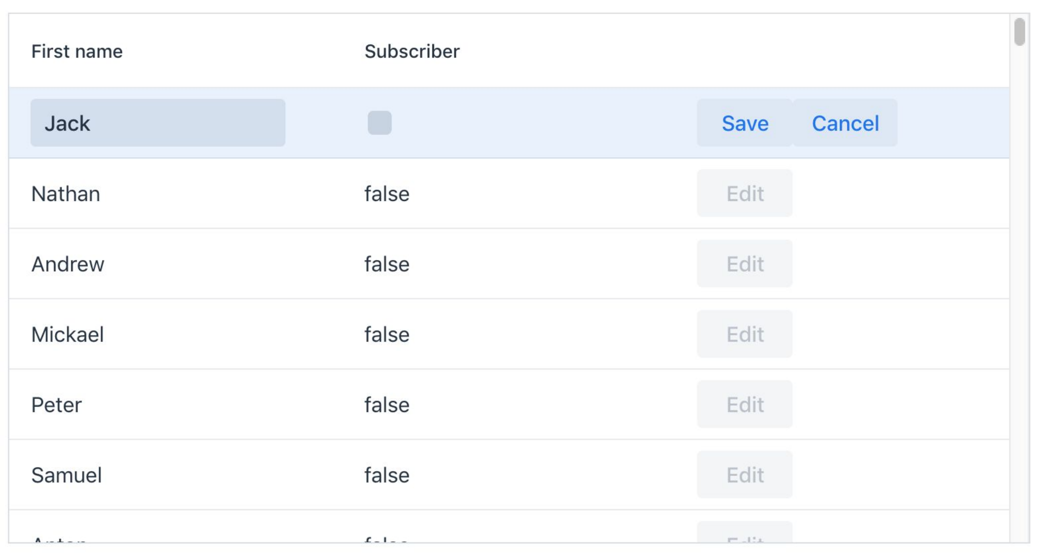
First name	Subscriber
Jack	☐
Nathan	false
Andrew	false
Mickael	false
Peter	true
Samuel	false
Anton	false
Aaron	false
Jack	false

Grid Editor

Buffered Mode

```
grid.getEditor().setBuffered(true);
```

Note: buffered mode only changes how the binder works: setBean() vs read/writeBean(). Edit/Save/Cancel buttons need to be built manually.

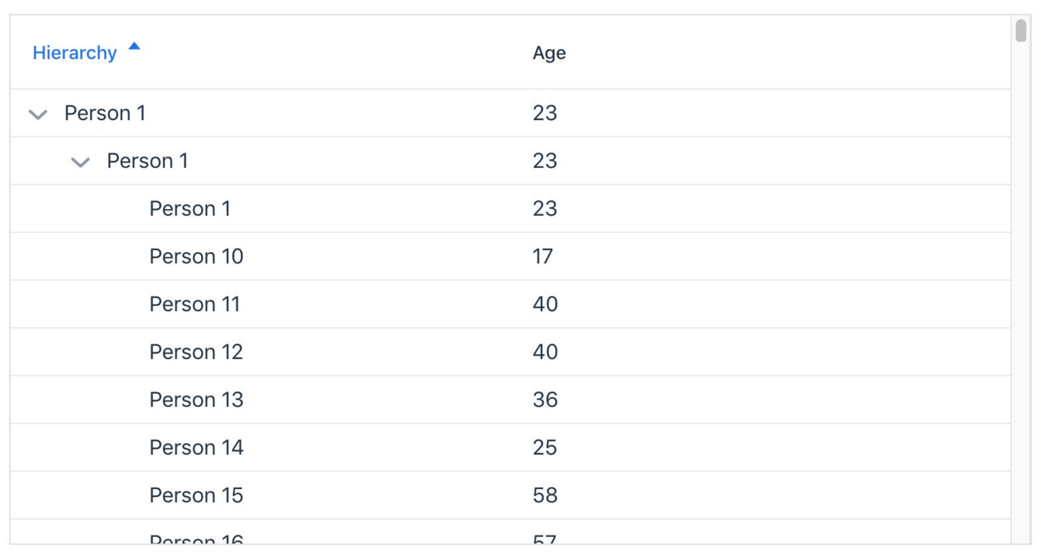


First name	Subscriber	
Jack	☐	Save Cancel
Nathan	false	Edit
Andrew	false	Edit
Mickael	false	Edit
Peter	false	Edit
Samuel	false	Edit
Anton	false	Edit

Tree Grid

Use TreeGrid to display hierarchical data.

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

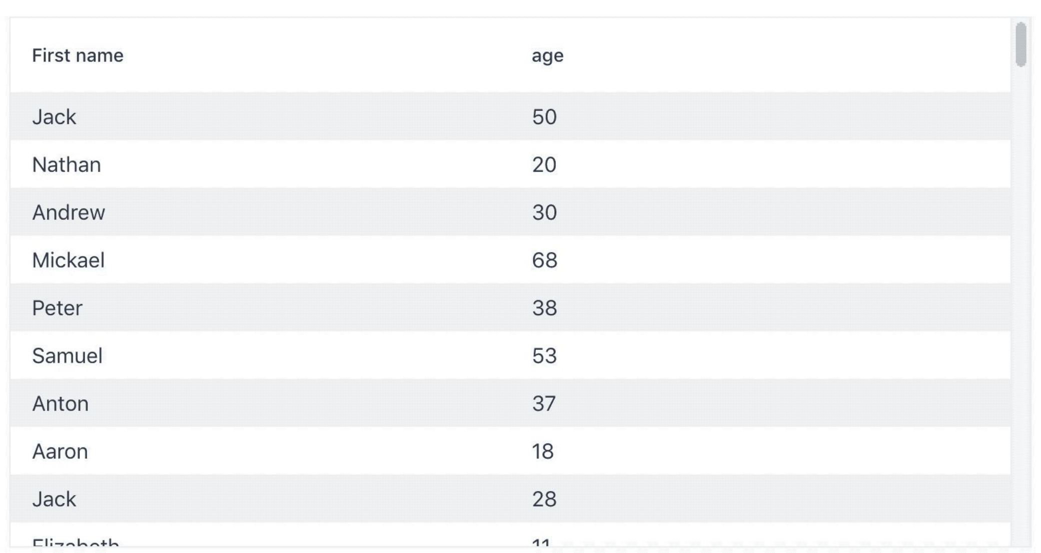


Hierarchy ▲	Age
⌄ Person 1	23
⌄ Person 1	23
Person 1	23
Person 10	17
Person 11	40
Person 12	40
Person 13	36
Person 14	25
Person 15	58
Person 16	57

Theme Variants

Use Theme variants to quickly change the look and feel

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

GridVariant.LUMO_NO_BORDER
GridVariant.LUMO_NO_ROW_BORDER
GridVariant.LUMO_COLUMN_BORDER
GridVariant.LUMO_STRIPES
GridVariant.LUMO_COMPACT
GridVariant.LUMO_WRAP_CELL_CONTENT
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

Exercise 1

In-Memory DataProvider

Populate data to Grid

What happens when calling grid.setItems()?

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

```
grid.setDataProvider(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()

Supports sorting and filtering

Sorting

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 the bean:

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

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

Sorting

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
setSortOrder(ValueProvider<T, V> valueProvider, SortDirection sortDirection)
// add a new sort order
addSortOrder(ValueProvider<T, V> valueProvider, SortDirection sortDirection)

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

Filtering

In-Memory DataProviders support direct filtering. Possible to filter based on the bean.

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

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

Filtering

In-Memory DataProviders support direct filtering. Possible to filter based on a certain value.

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

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

TreeDataProvider

ListDataProvicer needs a List/Collection for the backing data.

TreeDataProvider needs a **TreeData** for the backing data.

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

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

// add children for the root level 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);
```

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:
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 the same instance, or the class implements hashCode() and equals() methods.

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

Exercise 2

Lazy Data Providers

What information do you need for lazy loading?

Query object

Query object providers 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.

```
// API
public interface FetchCallback<VALUETYPE, FILTERTYPE> {

   Stream<T> 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.

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

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

CallbackDataProvider

A lazy data provider that uses **two callbacks**: one for **fetching** items from a back end, the other for **counting** the number of available items.

```
// 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());
```

Filtering Lazy Data

Use DataProvider.fromFilteringCallbacks() to create a CallbackDAtaProvider that supports filtering

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

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

Filtering Lazy Data

The fetch callback needs to use the filter. Note that query.getFilter() return an Optional.

```
DataProvider<Person, String> dataProvider = DataProvider.fromFilteringCallbacks(
     query -> backend.getPersons(query.getOffset(), query.getLimit(), query.getFilter().orElse(null)),
     query -> backend.countPersons());
```

Filtering Lazy Data

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

```
DataProvider<Person, String> dataProvider = DataProvider.fromFilteringCallbacks(
     query -> backend.getPersons(query.getOffset(), query.getLimit(), query.getFilter().orElse(null)),
     query -> backend.countPersons());

//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.

```
DataProvider<Person, String> dataProvider = DataProvider.fromFilteringCallbacks(
     query -> backend.getPersons(query.getOffset(), query.getLimit(), query.getFilter().orElse(null)),
     query -> backend.countPersons());

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

Filtering Lazy Data

setFilter() API is now available in the ConfigurableDataProvider.

```
DataProvider<Person, String> dataProvider = DataProvider.fromFilteringCallbacks(
     query -> backend.getPersons(query.getOffset(), query.getLimit(), query.getFilter().orElse(null)),
     query -> backend.countPersons());

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.

```
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.

```
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:

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

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

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.

```
// 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 the three methods are based on a certain node, which 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.

```
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

Grid Pro

What is Grid Pro?

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

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

Inline editing

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

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

Inline editing

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

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

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

Inline editing

addEditColumn() vs addColumn()

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

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

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

Inline editing

Use EditColumnConfigurator to config an editor for the content

```
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(..);
```

Inline editing

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));
```

Inline editing

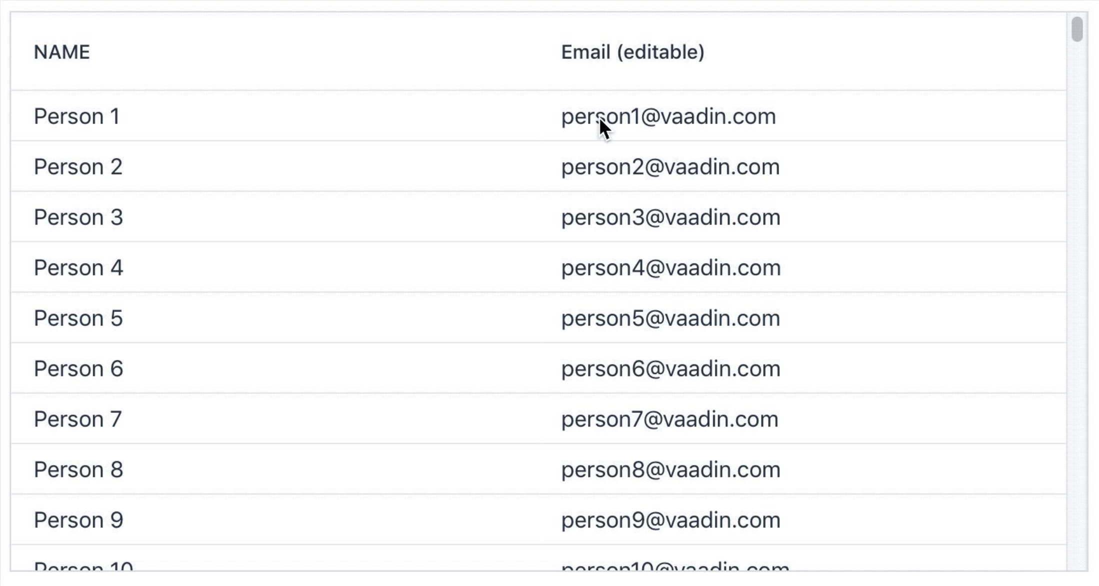
The editor configuring method returns a Column.

```
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

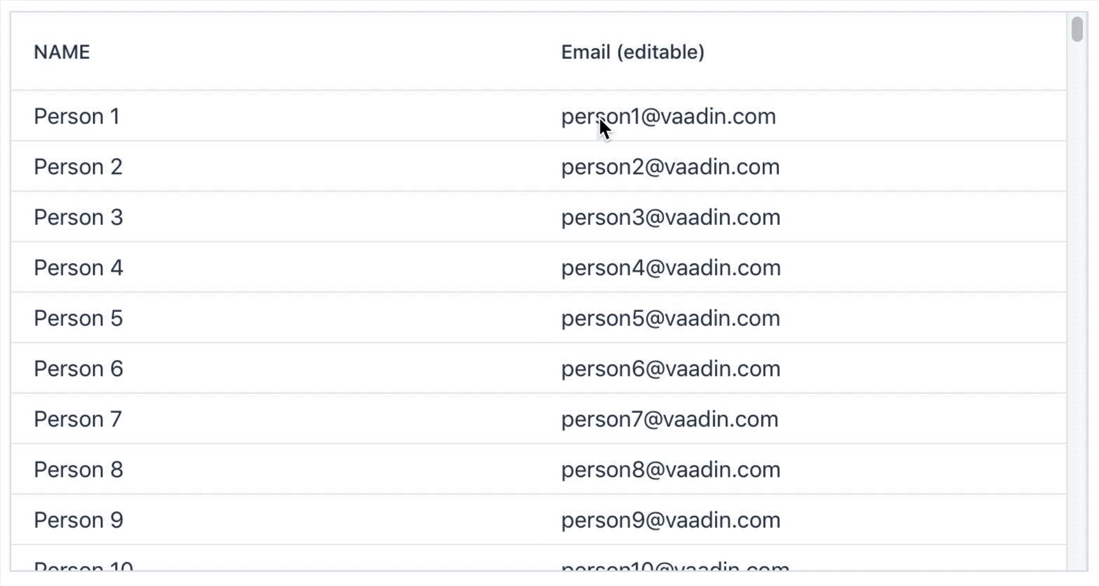
```
GridPro<Person> grid = new GridPro<>();
grid.addEditColumn(Person::getEmail)
     .text((item, newValue) ->
           item.setEmail(newValue))
     .setHeader("Email (editable)");
```



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

Inline editing

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



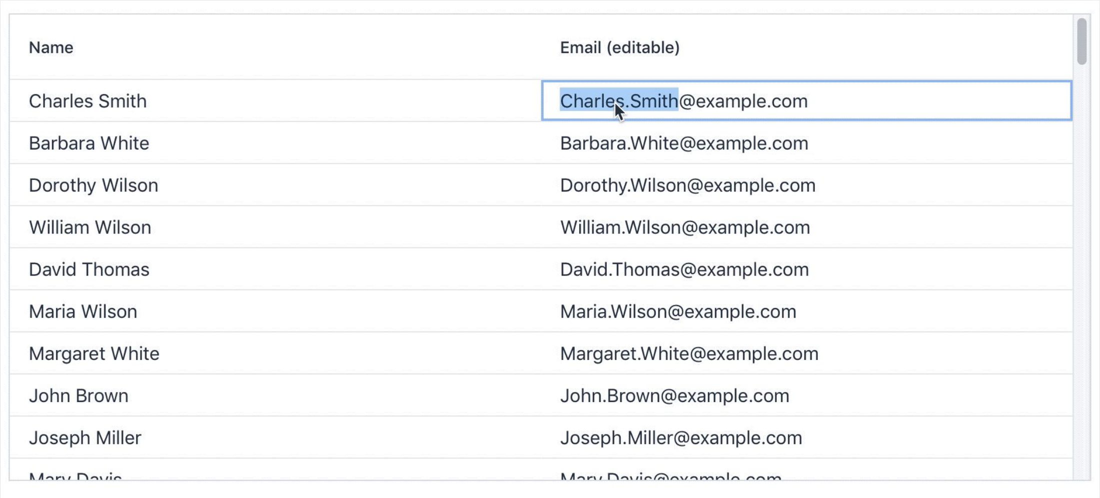
NAME	Email (editable)
Person 1	person1@vaadin.com
Person 2	person2@vaadin.com
Person 3	person3@vaadin.com
Person 4	person4@vaadin.com
Person 5	person5@vaadin.com
Person 6	person6@vaadin.com
Person 7	person7@vaadin.com
Person 8	person8@vaadin.com
Person 9	person9@vaadin.com
Person 10	person10@vaadin.com

Tab Navigation

When in edit mode

Tab: move to next edit cell

Shift+Tab: move to previous edit cell

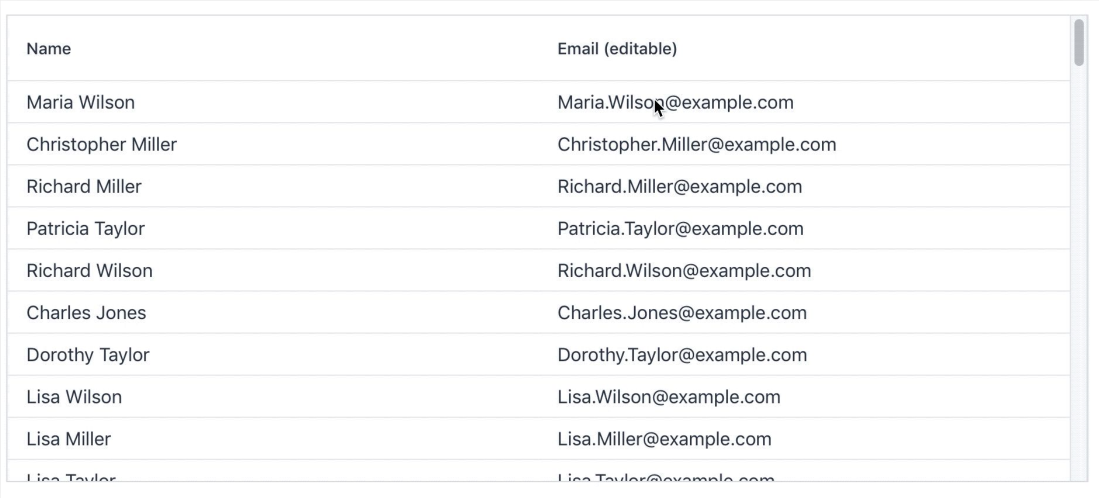


Name	Email (editable)
Charles Smith	Charles.Smith@example.com
Barbara White	Barbara.White@example.com
Dorothy Wilson	Dorothy.Wilson@example.com
William Wilson	William.Wilson@example.com
David Thomas	David.Thomas@example.com
Maria Wilson	Maria.Wilson@example.com
Margaret White	Margaret.White@example.com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

Enter Navigation

Could also use 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

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

Feedback

bit.ly/vaadin-training