VAADIN TRAINING

Data Providers

vaadin}>

Overview

Vaadin 8 introduced a completely rewritten data model that replaces the Container API. The new API is based on Java Beans, and is used in all selection fields. The `DataProvider` API allows both in-memory and lazy (paged) data sources and is very simple to use. In addition, all select components also have convenience APIs that allow simple usage with Java collections.

Convenience API for Grid, ComboBox and others

All components have a `setItems()` method that takes a Collection or a Stream:

JAVA

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

grid.setItems(list);
// or, pass a stream directly to the Grid:
addressGrid.setItems(list.stream()
.filter(p -> someFilter(p))
.map(Person::getAddress).distinct().sorted());
```

The DataProvider interface

Super-interface for all implementations

The `DataProvider` interface has basic methods for getting the size and single items from the data source. If your data changes after you create the provider, you need to call refresh:

JAVA

```java
dataProvider.refreshAll();
// or:
dataProvider.refreshItem(personWithNewData);
```

`DataProviders` are differentiated into two categories; in-memory and lazy. All select components can use either one, but only Grid and ComboBox actually utilize lazy-loading. Typically you use a subclass such as `ListDataProvider` or `BackEndDataProvider` (in-memory and lazy, respectively). To create an in-memory data provider from a List you simply do this:

JAVA

```java
ListDataProvider<Person> provider = DataProvider.ofCollection(persons);
grid.setDataProvider(provider);
```

The `ListDataProvider` supports Streams, Arrays and Collections, and should be used for smaller sets of data (less than a few hundred). If you have more data, it makes sense to create a lazy provider instead.

ListDataProvider

Used by all convenience methods in e.g. Grid and ComboBox. Supports sorting and filtering. Typically the `DataProvider` is sorted by the user, but you can do it programmatically with `setSortOrder`:

JAVA

```java
// Overrides previous commands
setSortOrder(ValueProvider<T, V> valueProvider, SortDirection sortDirection)

// Adds to previous commands
addSortOrder(ValueProvider<T, V> valueProvider, SortDirection sortDirection)

// Used like this:
dataProvider.setSortOrder(Consultant::getPriority, SortDirection.DESCENDING);
```

The `ListDataProvider` knows how to sort Comparables out of the box. You can override how the provider is sorted with SortComparators:

JAVA

```java
// Overrides earlier settings
setSortComparator(SerializableComparator<T> sortOrder);

// Adds secondary sorting comparators
addSortComparator(SerializableComparator<T> sortOrder);

// Usage is easy with the Java 8 Comparator helper class:
dataProvider.setSortComparator(
        Comparator.comparing(Consultant::getTaxBracket)::compare);
```

Filtering a `DataProvider` can be done by two different methods; direct filtering and query filtering. Query filtering is mostly used by lazy providers, and the direct sorting for in-memory providers. The direct filtering API looks like this:

JAVA

```java
// Predicate filtering is a lambda that receives the item and returns a boolean:
setFilter(SerializablePredicate<T> filter);
// Used like this:
dataProvider.setFilter(person -> person.getEmail() != null);

// Use ValueProvider if you aren't filtering on the item itself,
// but derived objects (like salary here):
setFilter(ValueProvider<T, V> valueProvider, SerializablePredicate<V> valueFilter);
// Used like this:
dataProvider.setFilter(Person::getSalary, salary -> salary >= 50000);

// OR, you can filter by direct value, which uses Objects.equals to match
// the values:
setFilterByValue(ValueProvider<T, V> valueProvider, V requiredValue);
// Used like this:
dataProvider.setFilterByValue(Person::getBirthDate, null);
```

TreeDataProvider

This data provider is used in the `Tree` and `TreeGrid` components. Since hierarchical data is more complicated than a simple list, the implementation has been split into two parts; the main classes to know are `TreeData` and `TreeDataProvider`.

JAVA

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

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);
```

Sorting and filtering work with the same API as the `ListDataProvider`, but they are ran on each level separately, starting from the root level.

Lazy Data Providers

Primary implementation class is `CallbackDataProvider`, that gives a hint on how it works. To create one, you call this API:

JAVA

```java
// T is the data type, the second type is for the filter, explained later.
// We’ll use Void for now.
DataProvider.fromCallbacks(
        FetchCallback<T, Void> fetchCallback,
        CountCallback<T, Void> countCallback));
```

T is the data type, the second type is for the filter, explained later. We'll use Void for now.

JAVA

```java
// And an example:
DataProvider<Person, Void> data = DataProvider.fromCallbacks(
        query -> backend.getPersons(query.getOffset(), query.getLimit()),
        query -> backend.countPersons());
```

The implementation is fully callback-based. The first callback should return a Stream of the actual items, the second an integer.

Because of the different ways backends can be implemented, Vaadin does not place any restrictions on how data is fetched. You can fetch data from a database, from a REST service, or EJBs.

The Query object contains the following data that should be used when fetching data:

JAVA

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

Limit and Offset are simple paging parameters. Note that the offset is the item index, not page number.

Since Vaadin doesn't have the full list of data at hand (that is the pint of lazy loading), it can't sort or filter the data effectively. Both sorting and Filtering need to happen in the backend. `QuerySortOrder` contains all data needed to do that.

For example, if we allow sorting from a Grid and have a lazy provider, we need this:

JAVA

```java
// UI code
grid.addColumn(Consultant::getSalary).setSortProperty("salary");

// Backend code
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(); // "salary"
        SortDirection direction = order1.getDirection();
        ...
}
```

If you want to filter a `CallBackDataProvider`, you need to define the type of the filter. Most of the time, a String is used:

JAVA

```java
// getFilter returns a String. This can be used e.g. for ComboBox providers,
// where the component has built-in String filtering
DataProvider<Person, String> data = DataProvider.fromFilteringCallbacks(
        query -> backend.getPersons(query.getFilter()),
        query -> backend.countPersons(query.getFilter()));
```

The filter type can be any class you want. If you need multiple filter parameters, you can do this to enable it:

JAVA

```
// Our filter type class with three properties
public class MyFilterValueHolder {
        String nameFilter;
        Integer maxAge;
        Date birthDay;
}

// UI code looks like this:
DataProvider<Person, MyFilterValueHolder> data =
        DataProvider.fromFilteringCallbacks(
        query -> backend.getPersons(query),
        query -> backend.countPersons(query));

// And backend loks like this:
public Stream<Person> getPersons(Query<Person, MyFilterValueHolder> query) {
        Optional<MyFilterValueHolder> filters = query.getFilter();
        ...
}
```

To use the filter from code, do this:

JAVA

```
// Create the provider, similar to above
DataProvider<Person, MyFilterValueHolder> data =
        DataProvider.fromFilteringCallbacks(…);

// Enable programmatic update of filter
ConfigurableFilterDataProvider<Person, Void, MyFilterValueHolder>
        filteredData = data.withConfigurableFilter();
grid.setDataProvider(filteredData);

// And update the filters whenever you need to
myFilterTextField.addValueChangeListener(e -> {
        MyFilterValueHolder filters = new MyFilterValueHolder();
        filters.setNameFilter(myFilterTextField.getValue());
        filteredData.setFilter(filters);
        // component will be updated automatically
});
```

For ComboBox, things are a bit simpler since we always use a String filter. If your provider already has a String filter type, you are good to go. If your provider has some other type, you'll need to convert the String from the ComboBox to whatever type your provider understands:

JAVA

```java
// create provider with Integer filter type
DataProvider<Person, Integer> actualProvider = ...
// convenience method for filter conversion
ComboBox.setDataProvider(provider, filterString -> {
       return Integer.parse(filterString);
});
```

Hierarchical Lazy Providers

The new APIs also allow hierarchical data to be loaded lazily from the back end. The base class for this is the `AbstractBackEndHierarchicalDataProvider`. There are three methods to implement:

- **`boolean hasChildren(T item)`**
  An often-called method that should return whether a node has child nodes or not.

- **`int getChildCount(HierarchicalQuery<T, F> query)`**
  Should return the number of children for a specific node, taking filters into account.

- **`Stream<T> fetchChildren(HierarchicalQuery<T, F> query)`**
  Should return the actual nodes, taking filters, sorting and paging into account.

Grid API

In addition to using the `DataProvider` API, the Grid has been extensively reworked.

The Grid Column API

There are two methods of creating a Grid. One creates columns for you, the other doesn't.

JAVA

```java
Grid<Person> grid = new Grid<>(Person.class); // scans Person and adds columns
        Grid<Person> grid = new Grid<>(); // Has nothing to scan, add cols yourself
```

The first constructor creates columns, their IDs and captions for you. You can use the field name to access a column:

JAVA

```java
// When Grid scans your Bean it creates columns for you.
// You can get the column by name:
public Column<T, ?> getColumn(String propertyName)
```

To add columns, use the following API:

JAVA

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

// You can also specify a custom Renderer
public Column<T, V> addColumn(
        ValueProvider<T, V> valueProvider,
        AbstractRenderer<? super T, ? super V> renderer)
```

The `ValueProvider` interface is simply a lambda:

JAVA

```java
// get value by explicit ValueProvider, these are the same thing:
grid.addColumn(Product::getPrice).setCaption(“Price");
        grid.addColumn(product -> product.getPrice()).setCaption("Price");
```

The default renderer is called the `TextRenderer`, but there are others:

- `TextRenderer`
- `HtmlRenderer`
- `NumberRenderer`
- `DateRenderer`
- `ButtonRenderer`
- `ImageRenderer`
- `ProgressBarRenderer`

JAVA

```java
// Renderer examples:
grid.addColumn(
        person -> new ThemeResource("img/"+person.getLastname()+".jpg"),
        new ImageRenderer());

grid.addColumn(person -> "Delete",
        new ButtonRenderer(clickEvent -> {
               people.remove(clickEvent.getItem());
               grid.setItems(people);
        }));
```

The Grid supports sorting of columns, both with in-memory and lazy data providers, as long as the provider supports it too.

JAVA

```java
public void sort(String columnId)
public void sort(String columnId, SortDirection direction)

public void sort(Column<T, ?> column)
public void sort(Column<T, ?> column, SortDirection direction)

// Sorting multiple columns uses a Builder
public void setSortOrder(GridSortOrderBuilder<T> builder)
```

Grid columns also support resize, hide, widths, and expand ratios

Other useful Grid methods:

- `public GridSelectionModel<T> setSelectionMode(SelectionMode selectionMode)`
- `public SingleSelect<T> asSingleSelect()`
- `public MultiSelect<T> asMultiSelect()`

- `public void scrollToStart()`
- `public void scrollToEnd()`
- `public void scrollTo(int row)`

- `public HeaderRow appendHeaderRow()`
- `public FooterRow appendFooterRow()`

- `public void setFrozenColumnCount(int numberOfColumns)`

- `public void setStyleGenerator(StyleGenerator<T> styleGenerator)`
- `public void setDescriptionGenerator(DescriptionGenerator<T> descriptionGenerator)`

The Row Editor

The Grid has a built-in editor feature that you can use for inline editing of any cell. The editor is based on a Binder. Similar to Binder, you'll need to bind any editors yourself. To enable the editor, yo'll need to do two things. First, enable editing:

JAVA

```java
Grid<Person> grid = new Grid<>(Person.class);
grid.getEditor().setEnabled(true);
```

Then, define an editor component for those columns you want to edit:

JAVA

```java
// The column definition itself already has a reference to the getter,
// but we need to provide a setter too

.setEditorComponent(new TextField(), Person::setName);
```

If your column isn't of String type, you'll need to configure Converters, similar to Binder:

JAVA

```
Grid<Person> grid = new Grid<>(Person.class);
Binder<Person> binder = grid.getEditor().getBinder();
Binding<Person, Integer> binding = binder.forField(new TextField())
        .withConverter(new StringToIntegerConverter("Not a valid number”))
        .bind(“age");
grid.getColumn(“age”).setEditorBinding(binding);
grid.getEditor().setEnabled(true);
```

Other useful Editor methods:

- `setBuffered(boolean buffered);`
- `setSaveCaption(String saveCaption);`
- `setCancelCaption(String cancelCaption);`
- `save();`
- `cancel();`
- `addSaveListener(EditorSaveListener<T> listener);`
- `addCancelListener(EditorCancelListener<T> listener);`
- `setErrorGenerator(EditorErrorGenerator<T> errorGenerator);`

The Details Row

You can add any Vaadin component in between Grid rows by using the Details row generator.

JAVA

```
grid.setDetailsGenerator(person -> {
        return new VerticalLayout(new Label(person.getDescription()));
});
```

The visibility of the details component is controlled programmatically:

JAVA

```
grid.addItemClickListener(e ->
        grid.setDetailsVisible(e.getItem(), !grid.isDetailsVisible(e.getItem())));
```

ComboBox API

Uses the same `DataProvider` API that Grid uses, with the same convenience methods:

JAVA

```java
public void setItems(Collection<T> items)
public void setItems(T... items)
```

To customize the caption, use an ItemCaptionGenerator:

JAVA

```java
combobox.setItemCaptionGenerator(
        person -> person.getFullName() + "(" + person.getEmail() + ")"
);
```

ComboBox, unlike Grid, has built-in filtering based on a String. By default, the filtering uses `String.contains()` on the item caption, in case insensitive mode. With an in-memory dataprovider, you can override this very easily:

JAVA

```java
combobox.setItems(
        this::filterComboboxCaptions, // Java 8 method reference to method below
        itemsList);
...
private boolean filterComboboxCaptions(String caption, String filter) { ... }
```

If the data you need to match the filter is not in the caption, you can tell the `DataProvider` where to find it:

JAVA

```java
ListDataProvider.filteringByPrefix(ValueProvider<T, String> valueProvider)
ListDataProvider.filteringBySubstring(ValueProvider<T, String> valueProvider)

// For example
DataProvider<Person, String> filteredProvider =
oldProvider.filteringBySubstring(Person::getFullName)
```

You can also provide a custom filter:

JAVA

```
comboBox.setDataProvider(oldDataProvider.filteringBy(
        (person, filterText) -> {
                if (person.getName().contains(filterText)) {
                        return true;
                }
                if (person.getEmail().contains(filterText)) {
                        return true;
                }
                return false;
        }
));
```

The ComboBox can allow users to provide new items with a NewItemHandler.

JAVA

```
// ListDataProvider example
ComboBox<String> tags = new ComboBox<>(existingTags);
tags.setNewItemHandler(newValue -> {
        existingTags.add(newValue);
        tags.setDataProvider(DataProvider.ofCollection(existingTags);
        tags.clear();
});

// BackEndDataProvider example
ComboBox<String> tags = new ComboBox<>();
        tags.setDataProvider(backendProvider);
        tags.setNewItemHandler(newValue -> {
                // saves the new tag
                backendService.addTag(newValue);
                // the new tag should now be picked up by the provider
                backendProvider.refreshAll();
});
```

Convenient ComboBox methods:

- `public void setEmptySelectionAllowed(boolean emptySelectionAllowed)`
- `public void setEmptySelectionCaption(String caption)`

- `public void setPlaceholder(String placeholder)`

- `public void setStyleGenerator(StyleGenerator<T> itemStyleGenerator)`
- `public void setItemIconGenerator(IconGenerator<T> itemIconGenerator)`