UsingDataSources

Updated Today (2 hours ago) by gessedafe

1 DataSources

Crux DataSources are objects capable of providing a set of data to widgets that implement `HasDataSource` interface. DataSources support features like pagination, data sorting and editing.

The following example shows a grid widget associated with a dataSource:

```
<html
   xmlns="http://www.w3.org/1999/xhtml"
   xmlns:c="http://www.sysmap.com.br/crux"
   xmlns:a="http://www.sysmap.com.br/crux/widgets">

      <c:screen useDataSource="simpleGridDataSource" useFormatter="birthday"/>

      <a:grid id="simpleGrid" height="200" width="100%" dataSource="simpleGridDataSource" pageSize="7">
         <a:dataColumn key="name" label="Name"/>
         <a:dataColumn key="phone" label="Phone"/>
         <a:dataColumn key="birthday" label="Birthday" formatter="birthday"/>
      </a:grid>
</html>
```

The DataSource class:

```
@DataSource("simpleGridDataSource")
@DataSourceBinding(identifier="name")
public static class SimpleGridDataSource extends LocalBindableEditablePagedDataSource<Contact> {
   @Create
   protected SimpleGridServiceAsync service;

   public void load()
   {
      service.getContactList(new DataSourceAsyncCallbackAdapter<Contact>(this));
   }
}
```

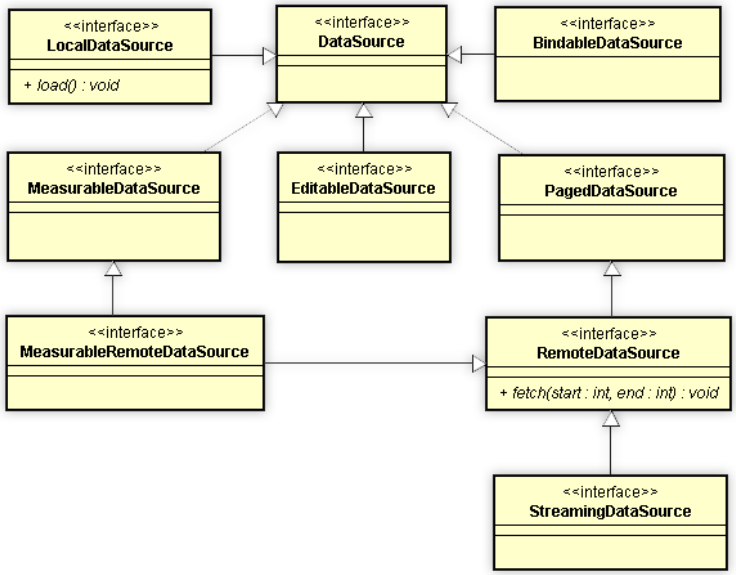
Note that you must declare a DataSource with the annotation `@DataSource` and then import it into your page using the screen attribute `useDataSource`. We can compare it with the Annotation `@Controller` and the `useController` screen attribute.

The main difference between DataSources and Controllers is the final purpose of these two classes. Controllers are used to handle widget events and data sources are used to load a set of data to serve a widget.

All basic DataSources implement the interface `br.com.sysmap.crux.core.client.datasource.DataSource` and one of the two interfaces: `br.com.sysmap.crux.core.client.datasource.LocalDataSource` or `br.com.sysmap.crux.core.client.datasource.RemoteDataSource` .

2 DataSources Hierarchy

There are a lot of DataSource interfaces, provided by ***Crux***, to support extra features, like pagination and data editing. Your DataSource class can implement many of those interfaces. The following figure shows the complete list of available interfaces:



Interface	Description
DataSource	The basic interface for any DataSource. Contains the basic methods to navigating through records and sorting data.
LocalDataSource	Local DataSources loads the data in just one step and keeps it in a local buffer. Contains the method `load()`

PagedDataSource	A DataSource that can divide data in blocks called *pages*. Contains the method `fetch(int start, int end)`, which is called by the associated widget when it needs the information contained in a page.
RemoteDataSource	Remote DataSources are also paged. This interface supports the cases when the data resides on the server and must be fetched on demand. When a page is needed, the DataSource requests it to server. It also keeps a buffer of the already loaded pages.
MeasurableDataSource	A DataSource that can be measured. In other words, when the number of records of the DataSource can be determined. It contains the method `getRecordCount()`
MeasurableRemoteDataSource	A Measurable and Remote Datasource. Contains the methods `load` and `setLoadData`, used to load DataSource configuration, that includes the size of the DataSource
StreamingDataSource	A DataSource that retrieves its data from a stream. The size of this stream is unknow. Streaming DataSources are also paged, but is not possible to know the number of pages before request the last one. All the pages must be requested in order.
EditableDataSource	DataSources that allow records selection or edition. It contains methods to retrieve the modified records
BindableDataSource	The data inside a DataSource can be structured as an array of records or as an array of Objects. BindableDataSources allow you to use an arbitrary Value Object to store the information.

2.1 DataSource Abstract classes

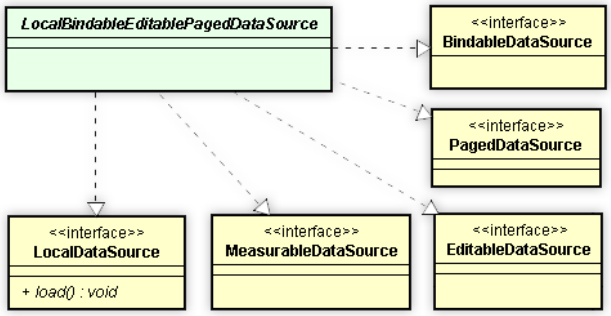
The easiest way of creating a DataSource is extending one of the ***Crux*** provided abstract classes. Those classes implement different combination of the above interfaces to simplify your implementation. Using one of those classes, you only need to implement one method to retrieve the data itself:

- `void load();` - for local DataSources
- `void fetch(int start, int end);` - for remote DataSources

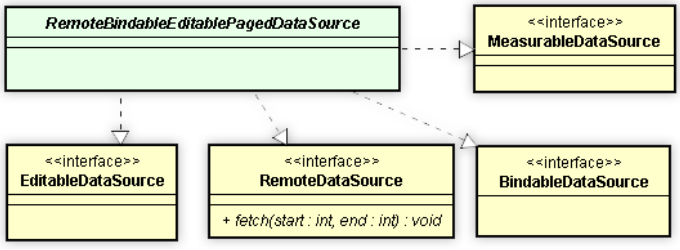
Inside this loader methods (`load` or `fetch`), you must call the `updateData` method after the data is retrieved.

If your DataSource implements the interface `MeasurableRemoteDataSource`, you must implement a `load` method too, in addition of the `fetch` method. The load method is used to load DataSource configuration, including the size of the remote DataSource. Inside this method you must call `setLoadData` once you have loaded the configuration data.

The following figures show some of those classes:



and



The complete list of basic abstract DataSources classes provided by ***Crux***:

Class	**Implemented DataSource Interfaces**
LocaScrollableDataSource	LocalDataSource
LocaPagedDataSource	LocalDataSource, PagedDataSource
LocalEditableScrollableDataSource	LocalDataSource, EditableDataSource
LocalEditablePagedDataSource	LocalDataSource, EditableDataSource, PagedDataSource
LocalBindableScrollableDataSource<T>	LocalDataSource, BindableDataSource<T>
LocalBindablePagedDataSource<T>	LocalDataSource, PagedDataSource, BindableDataSource<T>
LocalBindableEditableScrollableDataSource<T>	LocalDataSource, EditableDataSource, BindableDataSource<T>
LocalBindableEditablePagedDataSource<T>	LocalDataSource, EditableDataSource, PagedDataSource, BindableDataSource<T>
RemotePagedDataSource	MeasurableRemoteDataSource
RemoteStreamingDataSource	StreamingDataSource

RemoteEditablePagedDataSource	MeasurableRemoteDataSource, EditableDataSource
RemoteEditableStreamingDataSource	StreamingDataSource, EditableDataSource,
RemoteBindablePagedDataSource<T>	MeasurableRemoteDataSource, BindableDataSource<T>
RemoteBindableStreamingDataSource<T>	StreamingDataSource, BindableDataSource<T>
RemoteBindableEditablePagedDataSource<T>	MeasurableRemoteDataSource, EditableDataSource, BindableDataSource<T>
RemoteBindableEditableStreamingDataSource<T>	StreamingDataSource, EditableDataSource, BindableDataSource<T>

See the following examples:

```
@DataSource("contactDataSource")
@DataSourceBinding(identifier="name")
public static class RemoteDS extends RemoteBindableEditablePagedDataSource<Contact> {

   @Create
   protected SimpleGridServiceAsync service;

   public void load(){
      RemoteDataSourceConfiguration config = getConfig();
      setLoadData(config);
   }

   public void fetch(int startRecord, int endRecord){
      Contact[] data = getData();
      updateData(data);
   }
}
```

```
@DataSource("contactDataSource")
@DataSourceBinding(identifier="name")
public static class RemoteDS extends RemoteBindableEditableStreamingDataSource<Contact> {

   @Create
   protected SimpleGridServiceAsync service;

   @Parameter(required=true)
   protected String contact;

   public void fetch(int startRecord, int endRecord)
   {
      service.getContactPage(contact, startRecord, endRecord, new DataSourceAsyncCallbackAdapter<Contact>(this));
   }
}
```

3 DataSources Structure

It is possible to inform to ***Crux*** the structure of a DataSource in two different ways:

1. Using the annotation `@DataSourceColumns`
2. Implementing the interface `BindableDataSource<T>`, where T is some class that will contains the data of a row in DataSource.

Every row on a DataSource class must define one identifier field. It is needed to identify a row even if all the set of data were reordered.

See the examples in next two sections.

3.1 Defining Columns

You can define the DataSource structure informing directly all columns that compose the DataSource, as you can see in the following example:

```
@DataSource("simpleDataSource")
@DataSourceColumns(identifier="name", columns={
   @DataSourceColumn("address"),
   @DataSourceColumn(value="dateOfBirth", type=Date.class)
})
public static class SimpleDataSource extends LocalEditablePagedDataSource {

   @Create
   protected SimpleGridServiceAsync service;

   public void load()
   {
      EditableDataSourceRecord[] data = getData();
      updateData(data);
   }
}
```

The annotation `@DataSourceColumns` contains the fields:

- identifier - Tell ***Crux*** which field is the row identifier.
- columns - An array of `@DataSourceColumn`. Each of them defines a column on the DataSource.

The class `DataSourceRecord` is used to represent a record in the DataSource. This class defines a field `identifier` and a list of objects that contain the values of the columns (following the order in which the columns were declared). `EditableDataSourceRecord` is used when the DataSource supports editing.

3.2 Using a Class

You can define the DataSource structure informing a class. ***Crux*** will define columns following the names of the fields of the informed class. The type of each column will be the same type of the respective field. See the following example:

```
@DataSource("simpleGridDataSource")
@DataSourceBinding(identifier="name")
public static class SimpleGridDataSource extends LocalBindableEditablePagedDataSource<Contact> {

   @Create
   protected SimpleGridServiceAsync service;

   public void load()
   {
      Contact[] data = getData();
```

```
        updateData(data);
    }
}
```

```
public class Contact implements Serializable {

        private String name;
        private String phone;
        private Date birthday;

    // Getter and Setters ....
}
```

The annotation `@DataSourceBinding` must be used to inform which field in the Value Object will be used as row identifier ("name" in the above example).

That annotation can also be used to restrict which fields will be used to store column values inside the DataSource. It has two fields with this purpose:

1. includeFields - an array that inform the names of the fields of the binding class that will be used by the DataSource.
2. excludeFields - an array that inform the names of the fields of the binding class that will not be used by the DataSource.

Example:

```
@DataSource("simpleGridDataSource")
@DataSourceBinding(identifier="name", excludeFields={"birthday"})
public static class SimpleGridDataSource extends LocalBindableEditablePagedDataSource<Contact> {

    @Create
    protected SimpleGridServiceAsync service;

    public void load()
    {
        Contact[] data = getData();
        updateData(data);
    }
}
```

4 Handling Service Responses

As exposed in the previous sections, once you retrieve the data in the DataSource loader methods, you must call the `updateData` method of the DataSource.

To turn easier the task of load data inside the DataSource, ***Crux*** provides a special `AsyncCallback` implementation (DataSourceAsyncCallbackAdapter) that automatically takes the result of a service call and passes it to the `updateData` method.

See the following examples:

```
@DataSource("simpleDataSource")
@DataSourceColumns(identifier="name", columns={
    @DataSourceColumn("address"),
    @DataSourceColumn(value="dateOfBirth", type=Date.class)
})
public static class SimpleDataSource extends LocalEditablePagedDataSource {

    @Create
    protected SimpleGridServiceAsync service;

    public void load()
    {
        service.getContactList(new DataSourceAsyncCallbackAdapter<EditableDataSourceRecord>(this));
    }
}
```

and

```
@DataSource("simpleGridDataSource")
@DataSourceBinding(identifier="name")
public static class SimpleGridDataSource extends LocalBindableEditablePagedDataSource<Contact> {

    @Create
    protected SimpleGridServiceAsync service;

    public void load()
    {
        service.getContactList(new DataSourceAsyncCallbackAdapter<Contact>(this));
    }
}
```

Another `AsyncCallback` implementation is provided to do the same job to the `RemoteDataSourceConfiguration`, that is called `RemoteDataSourceLoadAsyncCallbackAdapter`. See the following example.

```
@DataSource("contactDataSource")
@DataSourceBinding(identifier="name")
public static class RemoteDS extends RemoteBindableEditablePagedDataSource<Contact> {

    @Create
    protected SimpleGridServiceAsync service;

    public void load(){
        service.getContactCount(new RemoteDataSourceLoadAsyncCallbackAdapter(this));
    }

    public void fetch(int startRecord, int endRecord){
        service.getContactPage(startRecord, endRecord, new DataSourceAsyncCallbackAdapter<Contact>(this));
    }
}
```