

DataValve Reference Guide

Andy Gibson

DataValve Reference Guide

Andy Gibson

1.0.0.final

Copyright © 2010 Andy Gibson

Table of Contents

Preface	viii
1. The Itch	viii
2. The Solution	ix
3. Benefits of DataValve	x
3.1. Interchangeable query mechanisms	x
3.2. Easy querying	xi
3.3. Flexible and secure ordering abstraction	xii
3.4. Convert anything into a dataset	xii
1. Introduction	1
1.1. Overview	1
1.2. Architecture	1
1.3. Examples	2
2. Database driven providers	5
2.1. Simple fetches	5
2.2. Restricting results	5
2.3. Parameter resolvers	7
2.3.1. Using your own parameter prefix	8
2.4. Order in the house	8
2.5. Polymorphism	9
2.6. Mapping SQL to objects	9
3. Implementing and testing a data provider	11
3.1. Our first <code>DataProvider</code>	11
3.2. Testing our <code>DataProvider</code>	12
3.3. Using our <code>DataProvider</code>	14
3.3.1. I want it all	16
3.3.2. Iteratively speaking	16
3.3.3. Paging datasets	17
3.3.4. Manually fetching data	18
4. The DataValve architecture	20
4.1. List of Module	20
4.2. Data provider types	20
4.2.1. <code>ParameterizedDataProvider</code>	21
4.2.2. <code>StatementDataProvider</code>	21
4.2.3. <code>QueryDataProvider</code>	21
4.2.4. <code>AbstractQlDataProvider</code>	21
4.2.5. <code>AbstractQueryDataProvider</code>	21
4.3. <code>ObjectDataset</code> Types	21
4.4. <code>ObjectDataset</code> usage	23
4.5. Choosing a dataset type	23
4.6. Extending <code>DataProvider</code> implementations	24
4.6.1. <code>void doPreFetch()</code>	24
4.6.2. <code>Integer doFetchResultCount()</code>	24
4.6.3. <code>List<T> doFetchResultCount()</code>	24
4.6.4. <code>Integer doPostFetchResultCount(Integer resultCount)</code>	24
4.6.5. <code>List<T> doPostFetchResultCount(List<T> results, Paginator paginator)</code>	24
4.6.6. Extending data providers	24
4.7. Extending datasets for custom providers	25
5. Non-database providers	27
5.1. In-memory based datastores	27

5.2. InMemoryAdapterProvider	29
5.3. IntegerDataProvider revisited	30
5.4. File Based Data Providers	31
5.4.1. TextFileProvider	32
6. Odds and Ends	33
6.1. IndexedDataProviderCache	33
6.2. Don't call record count	33
6.3. Seam Usage	33
6.4. DataValve and JSF	33
6.5. Executing code before a fetch	33

List of Figures

1. Projects without using DataValve	ix
2. Projects using DataValve	x
1.1. Architecture overview	2
3.1. Execution results	12
3.2. Demo application results	15
3.3. Output from returning all results	16
3.4. Output from iterating over a paginated dataset	17

List of Tables

4.1. List of Modules 20

List of Examples

1. Coding search queries the hard way.	xi
2. Coding search queries the easy way.	xii
1.1. The <code>DataProvider</code> interface	1
1.2. Fetching data from a provider	3
1.3. Limiting the data fetched from a data provider	3
1.4. Using an <code>ObjectDataset</code>	4
2.1. Fetching data from a database using JPA	5
2.2. Restricting the data fetched	5
2.3. Restricting results with constant parameters	6
2.4. Restricting with variable parameters	6
2.5. Alternate value parameters	6
2.6. Parameter Resolving with Reflection	7
2.7. Built-in parameter resolution	7
2.8. Using the Seam data provider	8
2.9. Using our own parameter parser.	8
2.10. Ordering on the data	9
2.11. Implementing the SQL to object mapping	10
2.12. SQL to object mapping using an external mapper.	10
3.1. <code>IntegerDataProvider.java</code>	11
3.2. <code>IntegerDataProvider.java</code>	12
3.3. <code>IntegerDataProvider.java</code>	13
3.4. Revised <code>fetchResults</code> method in <code>IntegerDataProvider.java</code>	13
3.5. Additional tests in <code>IntegerDataProviderTest.java</code> to validate the results returned.	14
3.6. Changing our demo application to use an <code>ObjectDataset</code>	15
3.7. Changing our demo application to use an <code>ObjectDataset</code>	15
3.8. Returning all results	16
3.9. Iterating through all results	16
3.10. Attempting manual iteration	17
3.11. Manually creating provider and pagination instances and fetching results	18
3.12. Multiple views of the same data	19
4.1. Poorly typed provider and dataset usage	22
4.2. Strongly typed data provider and dataset usage	22
4.3. Type safety with the <code>QueryDataProvider</code>	22
4.4. Examples of polymorphic provider references	23
4.5. Writing datasets for custom providers	25
4.6. Using your new data provider dataset	25
4.7. Using polymorphism to avoid type lock-in	26
5.1. In-memory person data provider.	27
5.2. Sorted in-memory person data provider.	28
5.3. Testing the <code>InMemoryPersonProvider</code>	29
5.4. Multiple views using an in-memory adapter.	30
5.5. In-memory implementation of our Integer provider.	30
5.6. Testing our in-memory Integer data provider.	31
5.7. Sorting the Integer data provider.	31
6.1. Setting up the query prior to fetching.	34

Preface

The purpose of this framework is to let developers use different sources of data in a consistent manner that leverages the similarities to create common interfaces to access them. It also aims to augment the functions of existing data access mechanisms, again leveraging the similarities between providers to create common extensions.

1. The Itch

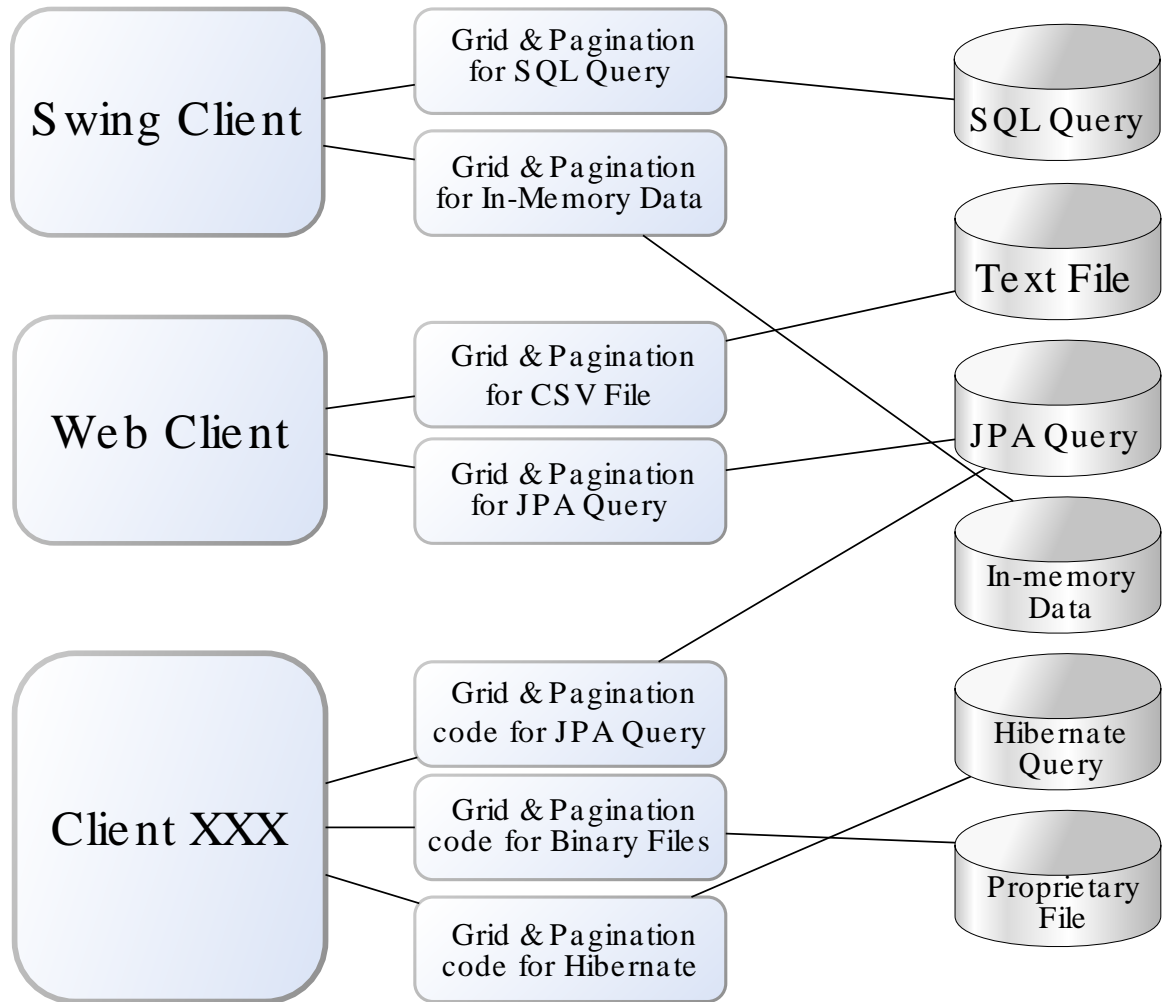
When starting a new data driven application developers typically need some mechanism to query data with the following capabilities:

- Pagination of large datasets
- Dynamic Ordering (for clickable columns)
- Lazy Loading of data for large datasets
- Optional restrictions in the query for when parameter values are missing
- The ability to apply a common front end to the back end data access mechanism so you can reuse both the front and back end to implement forms in minutes.

As a long time Seam user, Seam provides most of this with the `EntityQuery` that can be used to run queries using JPA but it isn't perfect. I extended the Seam `EntityQuery` and blogged about it [<http://www.andygibson.net/blog/index.php/2008/10/02/codeless-ajax-ordered-and-paginated-tables-in-seam/>] and it still remains a popular article today when people are looking for info on JSF and pagination. However this solution only works in Seam projects but the same functionality is needed in Wicket, Spring, JSP and even Swing and console apps. Other frameworks such as Wicket provide a front end and back end interface for handling ordered paginated datasets, but the actual implementation is up to the user as this is beyond the scope of Wicket.

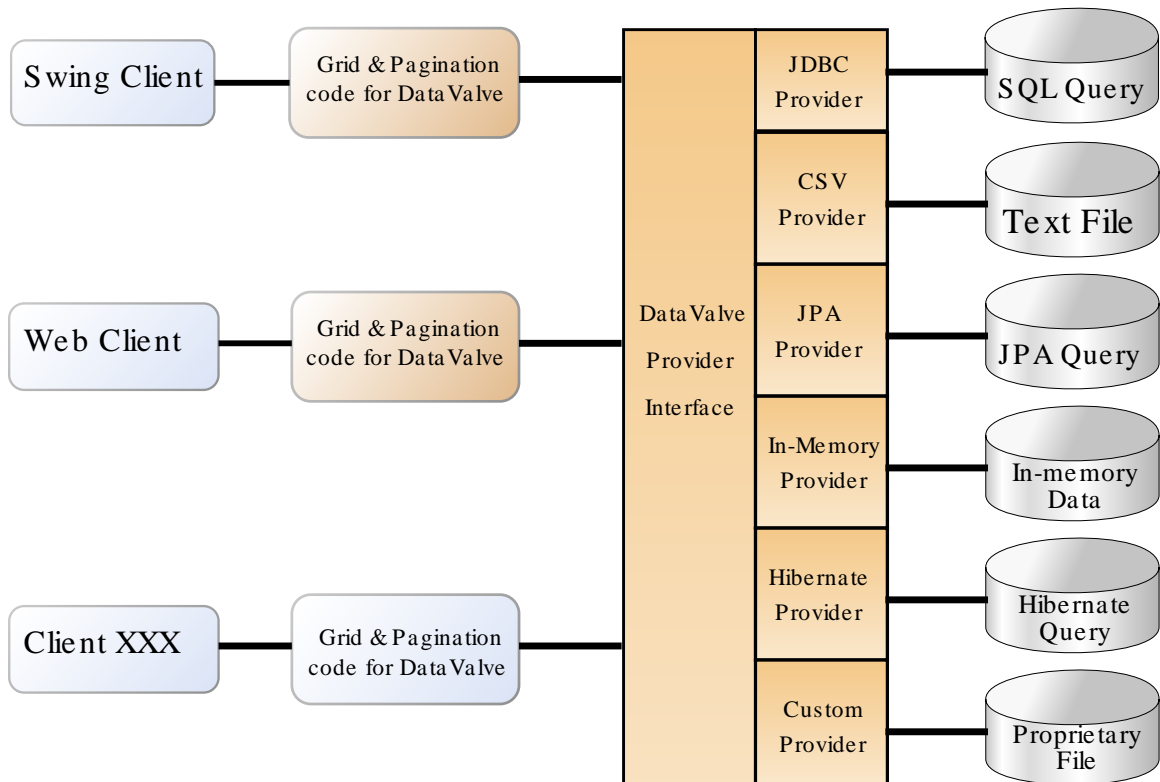
The other source of inspiration is from Borland Delphi which worked on the idea that a common data access interface meant that data driven components could access data without considering where it came from. A standard data interface meant that third party Delphi component developers could create data access components that would work with any kind of data source, from Oracle, MySQL and MS Access to XML or file based datasets as long as there was a data provider implementation for it. When developing in Java, you typically have to code against a common but more specific back end such as a JDBC query and write new interfaces for different view technologies. If you write the front end code for say JPA or Hibernate and you need to access an alternative data source then typically you cannot re-use your existing code for the new data source.

It seems that all these frameworks recognized the need for controlled data access but it was beyond the scope of the individual frameworks to provide a cross framework solution.

Figure 1. Projects without using DataValve

2. The Solution

So, I scratched my own itch and created this project which defines a common API for querying diverse data sources with pagination, ordering, restrictions and parameter management. This lets us write common view code that can interface with queries that are interchangeable. The goal of this project is to bridge the gaps between these different view and data access frameworks.

Figure 2. Projects using DataValve

3. Benefits of DataValve

Even if you still plan on using a single data access mechanism (i.e. Hibernate or JPA) there are still advantages to using DataValve. It's like a screw driver, 99% of the time you use it for one thing only, but when you need it to pry something open or some other function, it works equally well.

3.1. Interchangeable query mechanisms

Most projects plan on using only one data access mechanism but by coding to the DataValve API, you can switch implementations later. You might move from JPA to Hibernate or even to plain JDBC queries and most of the code that uses these queries, including parameterized queries and ordering, will remain the same. There may also be times where you need to access different types of data stores such as a file based datasource which needs accessing, paginating and displaying.

Regardless of where the different types of datastores come from, with DataValve you will be able to access them and control the results using your existing view code to paginate and order the data. Even if you don't see yourself switching data access mechanisms or having to access

diverse data stores, you may want to just switch a query here and there from a pure ORM type query to a JDBC SQL query but still want to interface with the data and use the features available in your other queries.

3.2. Easy querying

DataValve makes it easier to query for the data you need. It features flexible parameter definition by setting the value manually, using EL expressions, extracting the values using reflection or your own custom parameter resolver. Parameters that don't have a value assigned can optionally be left out of the query which can be useful when you are writing search forms where the user may leave out search criteria and you don't want it included in the query. Doing this manually can lead to messy code that is hard to read, test and maintain as you check parameter values, and include or exclude parts of the sql where clause depending on whether there is a value, after which you then have to stitch the query together.

Example 1. Coding search queries the hard way.

```
String sql = "";
boolean hasClause = false;
if (searchCriteria.id != null
    && searchCriteria.id.length() != 0) {
    if (sql.length() != 0) {
        sql = sql + " AND ";
    }
    sql= sql+ "p.id = :id";
    paramsMap("id",searchCriteria.id);
}

if (searchCriteria.firstName != null
    && searchCriteria.firstName.length() != 0) {
    if (sql.length() != 0) {
        sql = sql + " AND ";
    }
    sql= sql+ "p.firstName = :firstName";
    paramsMap("firstName",searchCriteria.firstName);
}
//check we have at least one restriction
if (sql.length() != 0) {
    sql = " WHERE " + sql;
}
//build the select statement
sql = [select statement] + sql;

//build the query
qry = createQuery(sql);
//add the parameters into the query
for (String key : paramsMap.keySet) {
    qry.setParameter(key,paramsMap.get(key))
}
//get the results
List<Person> results = qry.resultList();
```

With DataValve, you can define the query, specify how the parameters are resolved and the data provider will take care of generating the right request so that the only restrictions included in the query are ones with parameter values assigned.

Example 2. Coding search queries the easy way.

```
//define the parameterized restrictions
qry.addRestriction("p.id = :id");
qry.addRestriction("p.firstName = :firstName");
qry.addRestriction("p.lastName = :lastName");
//set the parameter resolver
ParameterResolver resolver =
    new ReflectionParameterResolver(searchCriteria));
qry.addParameterResolver(resolver);
//get results
List<Person> results = qry.resultList();
```

3.3. Flexible and secure ordering abstraction

DataValve lets you order your data based on order key values as opposed to explicit table column values which could be prone to SQL injection attacks. By using an `orderBy` value which is translated server side into a sorting representation, you can control what values are used for setting the order of the dataset ensuring no harmful SQL is inserted into your query.

3.4. Convert anything into a dataset

At the heart of DataValve is a simple API which allows you to turn anything into a source of data that can be accessed and paginated from existing view code. This can range from custom datasources or third party applications to text, csv, remote data points, binary files or even just a static in-memory list of objects.

Chapter 1. Introduction

We'll start by having a high level look at DataValve and how it is organized and how it is used.

1.1. Overview

DataValve is divided into three main interfaces which are used to fetch data. The `Paginator` interface defines the state about our dataset such as the order information, the page size and the index of the current results. The `DataProvider` interface defines two methods which are used to fetch the total number of rows in the complete list of data, and the list (or a subset) of items in the dataset. The final interface is the `ObjectDataset` interface which extends the `Paginator` interface and adds methods for managing a set of results once they have been obtained from the `DataProvider`.

Example 1.1. The `DataProvider` interface

```
public interface DataProvider<T> {
    Integer fetchResultCount();
    List<T> fetchResults(Paginator paginator);
}
```

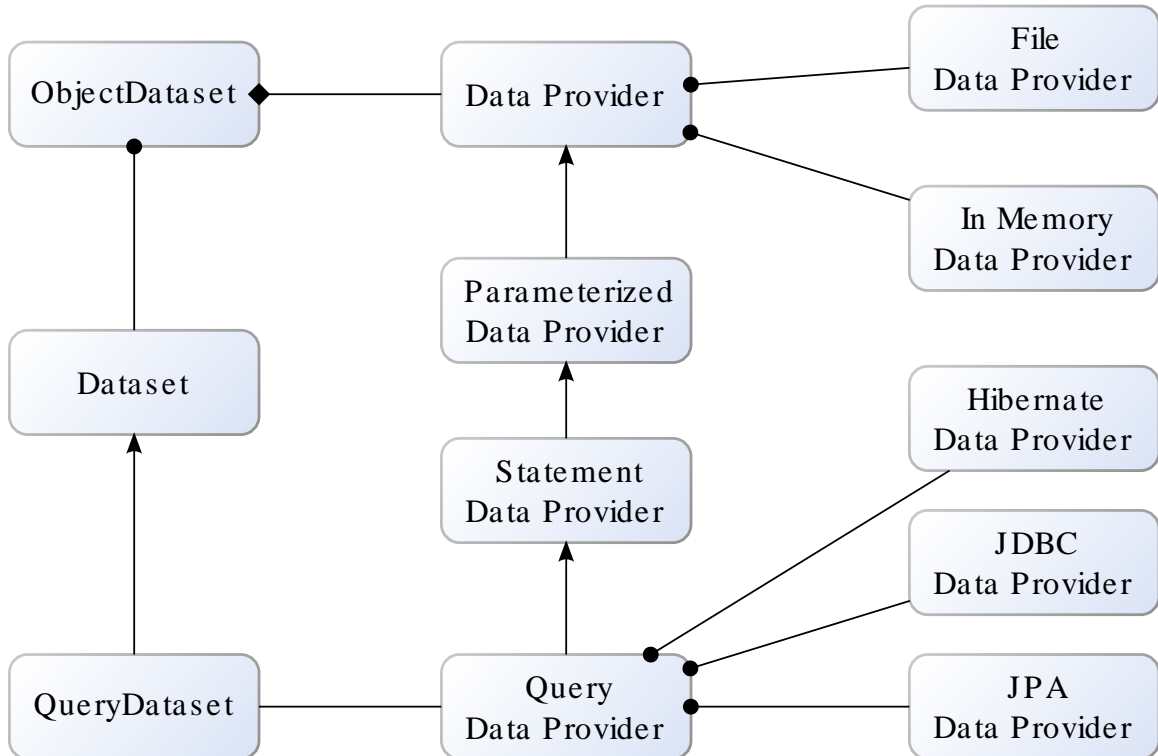
When we fetch the results, we pass in the paginator so the `DataProvider` implementation knows which rows to start returning rows from, how many rows to return and it can also set the flag on the paginator indicating whether there are more results or not.

This interface is very straightforward and its easy to see how you can implement any kind of data provider using such a simple interface. Once new data providers have been created, for the most part, they can be substituted for any other `DataProvider`.

1.2. Architecture

We extend the `DataProvider` to introduce features to interface with more complex data providers. We could use the `DataProvider` type, but then we don't get any access to the extended data provider features without casting which ties our code to that implementation and is messy to boot. Also, by creating an interface hierarchy, we can create an implementation hierarchy to share functionality with subclasses.

Simple data providers that are in memory or file based use the basic `DataProvider` interface. Where the real complexity comes in is with query language based data providers. While DataValve has been designed to be expandable, it can support different query language data providers, but the main goal for this concept was accomodating JPA, Hibernate and SQL based data providers. Most of the functionality for a query language based provider is defined in the `QueryDataProvider` interface.

Figure 1.1. Architecture overview

In this diagram you can see that the `DataProvider` subclasses define the interfaces for the multiple provider implementations. `ObjectDataset` and subclasses references a `DataProvider` implementation that is used to provide data to the dataset.

1.3. Examples

This section covers some basic examples of how each part of the core API is used. The core API consists of the `Paginator`, the `DataProvider` and the `ObjectDataset` interfaces. The `Paginator` is like a bookmark for a dataset and holds the current record index, page size and ordering information as well as flags indicating whether there is a previous or next page available. The `DataProvider` is the stateless part of the framework that provides the data based on the state of the paginator passed in. For each type of data store you want to access, we write a `DataProvider` implementation for it. The `ObjectDataset` is a higher level wrapper that extends the `Paginator` interface and adds functions to manage the results returned from a provider. The `ObjectDataset` has a `provider` attribute that it uses to fetch the data.

Example 1.2. Fetching data from a provider

```
DataProvider prov = new CustomDataProvider();
Paginator paginator = new DefaultPaginator();
List<ResultClass> results = prov.fetchResults(paginator);
Integer count = prov.getResultCount();
```

This is a simple example where we create an instance of a custom provider that knows how to fetch the data. We create an instance of a `Paginator` and pass that in to the data provider when we fetch the results to tell the provider which set of data to return. In this case, we returned all the data but by setting properties on the paginator we can tell the provider just to return a subset and which row to start from. The provider is also responsible for setting the `nextAvailable` flag on the paginator to indicate if there are more results available.

Example 1.3. Limiting the data fetched from a data provider

```
DataProvider<SomeClass> prov = new SomeClassDataProvider();
Paginator paginator = new DefaultPaginator();
paginator.setMaxRows(10);
paginator.setFirstResult(35);
List<SomeClass> results = prov.fetchResults(paginator);
Integer count = prov.fetchResultCount();
```

Here we set the maximum number of rows to return to 10 and we want the first result to start at item number 35. The implementation of the `SomeClassDataProvider` is responsible for returning the correct subset of data to the caller when `fetchResults` is called.

If you were to call `fetchResults` a second time, the provider would go off and re-fetch the data executing whatever process it does to fetch the data. It does not cache the data and should be implemented statelessly such that one call to the provider is independent of any other calls to the provider.

The third part of the API is an `ObjectDataset` which combines both the provider and paginator information into one class and can be used to manage the fetched data. This component should be considered stateful as it holds on to the results and caches them. It also implements the `Iterable` interface to let you iterate over the dataset while taking pagination into account, so while you iterate over each item, they are loaded in a batch at a time based on the `maxRows` attribute. Datasets implement the `Paginator` interface and have the pagination attributes built in. Datasets have a `provider` attribute that is used to fetch the actual data so we can re-use our dataset implementation for any kind of data provider. When you change the active page, the page size or the first result returned the internal copy of the current data is invalidated and causes it to be re-fetched from the provider when it is next requested.

Example 1.4. Using an ObjectDataset

```
DataProvider<SomeClass> provider = new SomeClassDataProvider();
ObjectDataset<SomeClass> dataset = new DataProviderDataset(provider);
dataset.setMaxRows(10);
dataset.setFirstResult(35);
List<SomeClass> results = dataset.getResults();

// we can iterate over the dataset and it will automatically load the results
// in pages of 10 results each.
for (SomeClass item : dataset) {
    ...some code...
}
```

Repeated calls to `getResults()` without changing the pagination information will cause the exact same results to be returned since the current page of results is managed by the `ObjectDataset`.

Chapter 2. Database driven providers

While DataValve offers a way to access many different types of data stores, in most cases, you will want to fetch data from a database. DataValve comes with data providers for fetching data using JPA, Hibernate or just plain SQL via JDBC. These different providers are based off and implement the `QueryDataProvider` interface. This interface extends the `DataProvider` interface and adds features such as ordering, restrictions and parameters. Since the different data providers implement the same interface, they can be interchanged with very little code changes if any.

2.1. Simple fetches

Let's start by considering some simple cases of fetching data using a data driven query. First we create the data provider that will fetch the data for us and then we create the dataset that we will use to access the data.

Example 2.1. Fetching data from a database using JPA

```
QueryProvider<Person> provider = new JpaDataProvider<Person>(em);
QueryDataset<Person> dataset = new QueryDataset<Person>(provider);
provider.setSelectStatement("select p from Person p");
provider.setCountStatement("select count(p) from Person p");
List<Person> results = dataset.getResultList();
for (Person p : results) {
    System.out.println(p.getName());
}
```

We define a new `JpaDataProvider` and pass it our `EntityManager` in the constructor. We create a new `QueryDataset` that we use to access the data and return the results to us. We could do this directly on the provider if we had created a `Paginator` instance to pass to it. Instead, we let the dataset implementation take care of it. We define the select and count statements separately since the select statements may contain fetch information that is invalid in the count statement.

2.2. Restricting results

We can add restrictions to the query so we only return a set of rows that we want. In this case, we want to return people in a specific department. The above code to create and initialize the data provider has been rolled into a single method that returns the person data provider.

Example 2.2. Restricting the data fetched

```
QueryDataProvider<Person> provider = createPersonProvider();
provider.addRestriction("p.department.id = 4");

//alternatively, we could use
QueryDataset<Person> dataset = new QueryDataset<Person>(createPersonProvider());
dataset.getProvider().addRestriction("p.department.id = 4");
```

It is often more convenient to specify limitations using parameters and DataValve has a number of different mechanisms with which to specify parameterized restrictions. First we'll look at adding constant parameters. We can add the restriction using the keyword `:param` where the parameterized value is meant to go and pass the value with the restriction. We can also add the restriction with the name of the parameter and set the parameter on the data provider separately. In the following example, the first two syntaxes rename the parameter and put the values in the internal parameter map. The last syntax keeps the parameter name you passed in and is useful if you anticipate changing that parameter since you know by what name it can be set.

Example 2.3. Restricting results with constant parameters

```
QueryDataProvider<Person> provider = createPersonProvider();

//add parameter using :param keyword in restriction
provider.addRestriction("p.department.id = :param",4);
provider.addRestriction("p.firstName = :param", "John");

//add parameter manually (takes 2 lines of code)
provider.addRestriction("or p.lastName = :lastNameParam");
provider.addParameter("lastNameParam", "Smith");
```

In many cases, we don't want to include restrictions where the parameter value we are using is null such as search forms where the user has left values blank. In these cases, any restriction with null parameter should not be included. In this example, we only add the first and last name restrictions if the values are set.

Example 2.4. Restricting with variable parameters

```
QueryDataProvider<Person> provider = createPersonProvider();
provider.addRestriction("p.firstName = :param",searchCriteria.firstName);
provider.addRestriction("p.lastName = :param",searchCriteria.lastName);
```

These are really just helper methods so you don't have to do your own "if value != null then add parameter" code for each line. There is also an overloaded version of this function specifically for strings which will check not only for null values but for a length of zero.

Many times you want to evaluate one value to see if is null but use another value for the actual parameter to query with. We have a method for that too.

Example 2.5. Alternate value parameters

```
QueryDataProvider<Person> provider = createPersonProvider();

provider.addRestriction("p.firstName like :param",
    searchCriteria.firstName,searchCriteria.firstName+"%");

provider.addRestriction("p.lastName like :param",
    searchCriteria.lastName,searchCriteria.lastName+"%");
```

We check whether the first/last name value is null and if so we ignore the restriction, but otherwise we add the restriction and we use the value with a wildcard added as the parameter value.

This may seem a fairly broad set of ways to parameterize the your queries, but is only the first half of DataValve's parameter definition mechanisms with the second being the use of parameter resolvers.

2.3. Parameter resolvers

The `ParameterResolver` interface can be used to implement objects that can be attached to data providers at run time to resolve parameters. When the query is executed the parameters are evaluated as the query is built so any restrictions with `null` parameters can be excluded. Different parameter resolvers can be used depending on the environment i.e. using an EL parameter resolver for EL environments.

Different mechanisms can be used for resolving values such as reflection on an object or just returning parameter values based on the parameter name.

Example 2.6. Parameter Resolving with Reflection

```
QueryDataProvider<Person> provider = createPersonProvider();
provider.addRestriction("p.firstName like :firstName");
provider.addRestriction("or p.lastName like :lastName");
provider.addParameterResolver(new ReflectionParameterResolver(searchCriteria));
```

We can also bake the parameter resolver into the criteria object directly by making it implement the parameter resolver interface. This kind of resolver executes faster than trying to resolve EL expressions or use reflection.

Example 2.7. Built-in parameter resolution

```
public class SearchCriteria implements ParameterResolver {

    //return values with wildcards
    boolean resolveParameter(ParameterizedDataProvider<? extends Object> dataset,
        Parameter parameter) {
        if (":firstName".equals(parameter.getName())) {
            return firstName == null ? null : firstName+"%";
        }
        if (":lastName".equals(parameter.getName())) {
            return lastName == null ? null : lastName+"%";
        }
        return null;
    }

    boolean acceptParameter(String name) {
        return name.startsWith(":");
    }
}
...
...
...
QueryDataProvider<Person> provider = createPersonProvider();
provider.addRestriction("p.firstName = :firstName");
provider.addRestriction("or p.lastName = :lastName");
provider.addParameterResolver(searchCriteria);
```

The Seam data provider classes use an EL based parameter resolver so you can use it the same way you would with a Seam `EntityQuery`.

Example 2.8. Using the Seam data provider

```
@Name("myPostsQuery")
public class MyPostsQueryBean extends SeamJpaQueryDataset{
    public MyPostsQueryBean() {
        setSelectStatement("select p from Posts p left join fetch p.comments");
        setCountStatement("select count(p) from Posts p");
        addRestriction("p.author.id = #{loggedInUser.id}");
    }
}
```

2.3.1. Using your own parameter prefix

You may decide you want to use a custom pattern for parameters in query providers restrictions. By default `:` and `#{...}` are supported for the default and EL parameter patterns. If you decide to have a reflection parameter type starting with `$` you will need to implement your own `ParameterParser` or you can extend the `RegexParameterParser` and override the `getRegex()` method to include `$` signs when parsing. This then needs to be provided to the `DataQueryBuilder` implementation which can be done by overriding the `createDataQueryBuilder` method in the query data provider so you can set your parameter parser on the query builder.

Example 2.9. Using our own parameter parser.

```
public class MyCustomJpaDataProvider<T> extends JpaDataProvider<T> {

    @Override
    protected DataQueryBuilder createDataQueryBuilder() {
        DataQueryBuilder result = super.createDataQueryBuilder();
        result.setParameterParser(new MyCustomParameterParser());
        return result;
    }
}
```

2.4. Order in the house

`DataValve` also features key based ordering so rather than specify ordering as a set of fields on the dataset, you can specify ordering using a key that indicates how to order the results.

Example 2.10. Ordering on the data

```
QueryDataProvider<Person> provider = createPersonProvider();
provider.addRestriction("p.lastName = 'SMITH'");
provider.getOrderKeyMap().put("name", "p.lastName,p.firstName");
provider.getOrderKeyMap().put("age", "p.dateOfBirth");
provider.getOrderKeyMap().put("id", "p.id");
//now set the order
Paginator paginator = new DefaultPaginator();
paginator.setOrderKey("name"); //order by the persons name
paginator.setOrderAscending(false);

List<Person> orderedList = provider.fetchResults(paginator);
```

The order information on the `Paginator` consists of the `orderKey` attribute that indicates the order to use and the `orderAscending` attribute indicates whether it is meant to be ascending or descending. Both of these attributes are on the `Paginator` interface and are read when it is passed into the `fetchResults` method. Note that in this example, the ordering for the key "name" has two fields (last and first name) to sort on. Both these fields will be used to order and the framework takes care of ascending/descending syntax issues.

By abstracting the ordering mechanism we can use the same idea for defining ordering on non-database data providers. The key for example might map to a `Comparator` instance that can be used to sort the in-memory dataset.

2.5. Polymorphism

One of the best features here is that in order to switch from Hibernate to JPA, or another data driven implementation, all we need to change is our `createPersonProvider()` method. Following good principles, we have been coding to an interface and not an actual class type so we are not locked in to using a concrete class, but any class that implements the `QueryDataProvider` interface. Currently, there are `QueryDataProvider` implementations for SQL/JDBC, Hibernate and JPA. The JPA versions include both Ejbql and Native querying as do the Seam extensions for the JPA providers.

You could easily return a Hibernate data provider from the `createPersonProvider` without having to change any code since the mappings will be the same. At most you might have to change the base select/count statement or any complex where clauses. The same benefits apply if you are returning a `QueryDataset` instead of a provider.

If you are using some kind of dependency inject then you just need to change which bean is injected into the injection point. The injection point receives the same benefits of polymorphism due to the fact that it is an interface implemented by multiple providers or datasets.

2.6. Mapping SQL to objects

One problem so far is that our data providers always deal with objects natively because that is what Hibernate and JPA do. Our SQL based datasets are fetching data as a set of rows of values instead of actual objects so we still need some way to convert that raw data into an object that can be returned in our results.

This can be done in one of two ways but both involve a `java.sql.ResultSet` and building an instance of our model objects from the current row of data.

The first method is to subclass the `AbstractJdbcDataProvider` and override the `createObjectFromResultSet` method. This uses the template pattern to provide the missing functionality needed to convert SQL data into an actual object.

Example 2.11. Implementing the SQL to object mapping

```
public class PersonDataProvider extends
    AbstractJdbcDataProvider<Person> {
    public T createObjectFromResultSet(ResultSet resultSet) throws SQLException {
        return new Person(resultSet.getLong(1), resultSet.getString(2),
            resultSet.getString(3));
    }
}
```

Some people prefer to use a strategy pattern over a template pattern so we have an implementation for that also. The `ResultSetObjectMapper` interface provides a method to take a `javax.sql.ResultSet` and return an object for the current row in the results. The `JdbcDataProvider` class takes a reference to a `ResultSetObjectMapper` instance and uses it to convert the raw SQL rows to an object.

Example 2.12. SQL to object mapping using an external mapper.

```
public class PersonObjectMapper implements
    ResultSetObjectMapper<Person> {

    public Person createObjectFromResultSet(ResultSet resultSet)
        throws SQLException {
        return new Person(resultSet.getLong(1), resultSet.getString(1),
            resultSet.getString(2));
    }
}

public QueryDataProvider<Person> createDataProvider() {
    JdbcDataProvider<Person> prov = new JdbcDataProvider<Person>();
    prov.setResultSetObjectMapper(new PersonObjectMapper());
    return prov;
}
```

This kind of construction can also be taken care of by any dependency injection framework you are using.

Chapter 3. Implementing and testing a data provider

This chapter looks at creating a simple data provider, testing it and seeing the different ways we can access the data in it. Note that we implement our provider by subclassing `Object` and implementing the `DataProvider` interface directly. This was for the purpose of demonstration and while you can create data providers this way, a better option would be to subclass the `AbstractDataProvider` which implements these methods and also adds points for subclasses to extend the class for both the data provider implementor and the application developer using a subclass of the implementation.

3.1. Our first `DataProvider`

Let us start by seeing how the different parts and pieces connect by creating a new provider that lists the numbers from 1 to 100 and we will access it in different ways. We'll start by writing our data provider that implements the `DataProvider<Integer>` interface.

Example 3.1. `IntegerDataProvider.java`

```
public class IntegerDataProvider implements DataProvider<Integer>,
    Serializable {

    public Integer fetchResultCount() {
        return 100;
    }

    public List<Integer> fetchResults(Paginator paginator) {

        int start = paginator.getFirstResult();

        int end = paginator.includeAllResults() ? 100 :
            Math.min(100, start+paginator.getMaxRows().intValue()-1);

        List<Integer> results = new ArrayList<Integer>();
        System.out.println("Generating results from "+start+" to "+end);
        while (start <= end) {
            results.add(start++);
        }
        paginator.setNextAvailable(end < 100);
        return results;
    }
}
```

The `fetchResultCount()` method returns a constant value of 100 in this case because we are dealing with a fixed range of integers. Typically though, this will return the number of items in the complete dataset and is used to determine the number pages available.

The `fetchResults` methods takes a paginator and generates the list of integers to return to the user. The list is based on two things, the `firstResult` value that determines which page we are on and therefore where to start counting from, and the `maxRows` value that determines how many results to return. If `maxRows` is set to null then we are returning all rows, and we have the utility method `includeAllResults` to make it easier to check this. Once we have determined the start and end points to count from and to, we build the list of results, and before returning it, we set the flag for indicating whether there are more results available. This is the responsibility of the data provider because only it knows whether there are more results or not.

Next, we'll create a client for this provider as a simple class that just fetches the results and lists them.

Example 3.2. IntegerDataProvider.java

```
public class Main {  
  
    public static void main(String[] args) {  
        Paginator paginator = new DefaultPaginator(10);  
        DataProvider<Integer> provider = new IntegerDataProvider();  
  
        List<Integer> results = provider.fetchResults(paginator);  
        showResults(results);  
    }  
  
    public static void showResults(List<Integer> results) {  
        System.out.println("Result Size = " + results.size());  
  
        for (Integer i : results) {  
            System.out.println("Value = " + i);  
        }  
    }  
}
```

In the main method, we create a new `DefaultPaginator` instance that implements the `Paginator` interface. We pass our initial page size into the constructor so we will start by returning 10 rows at a time. We then create an instance of our data provider, and call the `fetchResults` method to return the first page of results. We then pass the results to the `showResults` method which just displays the number of results. If we run this, we get the expected list of numbers from 0 to 9 with a page size of 10.

Figure 3.1. Execution results

```
Generating results from 0 to 9  
Result Size = 10  
Value = 0  
Value = 1  
Value = 2  
Value = 3  
Value = 4  
Value = 5  
Value = 6  
Value = 7  
Value = 8  
Value = 9
```

So everything looks fine, this is the result we expected so we must have a working dataset right? Well not so fast. Let's create a test case for it first. Since we are working with well defined interfaces we can define a test case that tests the interface and make sure our implementation is meeting the contract defined by the interface.

3.2. Testing our DataProvider

Create a new class (in the test packages if applicable) and create it as a subclass of `org.fluttercode.datavalve.testing.junit.AbstractObjectDatasetJUnitTest`

This class defines two methods which needs to be implemented in subclasses. The `getRowCount()` returns the number of rows expected in the dataset and the `buildObjectDataset` method returns an instance of an `ObjectDataset` for the provider we are testing. We can easily create one by creating an instance of a `org.fluttercode.datavalve.impl.Dataset` and pass it an instance of our `IntegerDataProvider` which the dataset will use to obtain data.

Example 3.3. IntegerDataProvider.java

```
public class IntegerDataProviderTest extends
    AbstractObjectDatasetJUnitTest<Integer> {

    @Override
    public ObjectDataset<Integer> buildObjectDataset() {
        DataProvider<Integer> provider = new IntegerDataProvider();
        return new Dataset<Integer>(provider);
    }

    @Override
    public int getRowCount() {
        return 100;
    }
}
```

If you run this test case now, you will see a lot of tests failing in code that not only looked ok, but worked fine when we ran it.

The problem is subtle but involves on the old gotcha there are 101 numbers from 0 to 100 inclusive. The solution is fairly simple, we just need to define more closely what our dataset does. We'll make it so that it does return 100 values in the range from 1 to 100, and we'll even add some test cases to ensure that it returns numbers in that range. First lets alter the `fetchResults` method in our `IntegerDataProvider` class.

Example 3.4. Revised fetchResults method in IntegerDataProvider.java

```
public List<Integer> fetchResults(Paginator paginator) {

    int start = paginator.getFirstResult()+1; //results start from 1

    int end = paginator.includeAllResults() ? 100 :
    Math.min(100, start+paginator.getMaxRows().intValue()-1);

    List<Integer> results = new ArrayList<Integer>();
    System.out.println("Generating results from "+start+" to "+(end));
    while (start <= end) {
        results.add(start++);
    }
    paginator.setNextAvailable(end < 100);
    return results;
}
```

If you run the tests now you should almost pass with flying colors even though we only made very small changes. The test that failed is the serialization test to determine if a dataset holding our data provider can be serialized. In this case, it cannot because we have not implemented the `Serializable` interface in the `DataProvider` class name and rerun the tests, it should pass with no problems. The serializable test is there as a helper because so many frameworks now depend on serializable components (Spring MVC, Wicket, CDI and Seam) and raises errors if there are serializable issues. If you do not need the dataset to be serializable or it cannot be made serializable, you can turn off this test by overriding the `includeSerializationTest()` and returning false.

The base test classes provide fairly good code coverage and can only get better over time as more tests are added. Even without writing any test cases yourself you are already off to a good start in determining how well your data provider code is written.

The `AbstractObjectDatasetJUnitTest` can only test how well the dataset lives up to its implementation of the `DataProvider` interface, it cannot test the quality of the results. For that, we need to add our own tests to our class which we'll do now for the constraint that the numbers should range from 1 to 100. We will test it in two ways, paginated and non-paginated.

Example 3.5. Additional tests in `IntegerDataProviderTest.java` to validate the results returned.

```
public void testResultsPaginated() {
    ObjectDataset<Integer> ds = buildObjectDataset();
    ds.setMaxRows(10);
    Integer expected = 1;
    for (Integer i : ds) {
        assertEquals(expected, i);
        expected = expected + 1;
    }
}

public void testResultsNonPaginated() {
    ObjectDataset<Integer> ds = buildObjectDataset();
    Integer expected = 1;
    for (Integer i : ds) {
        assertEquals(expected, i);
        expected = expected + 1;
    }
}
```

Again, running these tests should provide a clean set of test results.

3.3. Using our `DataProvider`

We can now be pretty confident that our dataset works as expected with dozens of test cases behind it. Now we can go back to our `Main` class and start seeing how we access the data provided by this class.

Data can be accessed from a provider in a number of ways, either paginated, non-paginated or using iterators paginated or non-paginated). In our original example, we simply listed the first 10 results. From the test code you might have seen a few more ways of accessing the data. First let's change our demo application to use an `ObjectDataset` instead of keeping separate `Paginator` and `DataProvider` references.

Example 3.6. Changing our demo application to use an ObjectDataset

```
public static void main(String[] args) {
    ObjectDataset<Integer> ds = new Dataset<Integer>(new IntegerDataProvider());
    ds.setMaxRows(5);

    List<Integer> results = ds.getResultList();
    showResults(results);
}
```

If you run this now, it should run the same as before except for the fact that we changed the page size to 5 so we only return 5 rows at a time for brevity. Let's add a second call to fetch the list of results and we'll see what happens.

Example 3.7. Changing our demo application to use an ObjectDataset

```
public static void main(String[] args) {
    DataProvider<Integer> provider = new IntegerDataProvider();
    ObjectDataset<Integer> ds = new Dataset<Integer>(provider);
    ds.setMaxRows(5);

    List<Integer> results = ds.getResultList();
    showResults(results);
    showResults(ds.getResultList());
}
```

Figure 3.2. Demo application results

```
Generating results from 1 to 5
Result Size = 5
Value = 1
Value = 2
Value = 3
Value = 4
Value = 5
Result Size = 5
Value = 1
Value = 2
Value = 3
Value = 4
Value = 5
```

Unsurprisingly, we display the same list of numbers twice, but what is surprising is that we only generate the results the first time they are requested. The second time we call for the results, there is no message about generating results. This is because the `ObjectDataset` caches the results and only invalidates them when the pagination changes i.e. the first result or page size is changed). You can manually force the results to be refreshed by calling `invalidateResults()` on the dataset. Doing so after the first call would force the dataset to re-fetch the data on the second call.

This is one of the benefits of using a dataset, it can hold on to the results for you until you are done with them. Using separate `Paginator` and `DataProvider` instances can be useful in

more stateless applications, or frameworks like Wicket that uses a detachable model that keeps hold of state, but lets you detach unwanted or re-fetchable state.

3.3.1. I want it all

You can fetch all the results in the dataset by settings the `maxRows` to `null` . This means that when you call `getResults` you will get all the results from the `firstResult` index to the end. When you create a `ObjectDataset` by default, it is set up to fetch results from the first row and is not paginated so it will return all results from the beginning to the end.

Example 3.8. Returning all results

```
public static void main(String[] args) {
    DataProvider<Integer> provider = new IntegerDataProvider();
    ObjectDataset<Integer> ds = new Dataset<Integer>(provider);
    showResults(ds.getResultList());
}
```

Figure 3.3. Output from returning all results

```
Generating results from 1 to 100
Result Size = 100
Value = 1
Value = 2
Value = 3
Value = 4
Value = 5
Value = 6
Value = 7
...
...
...
Value = 98
Value = 99
Value = 100
```

3.3.2. Iteratively speaking

The dataset implements the `Iterable` interface which means that we can iterate over the dataset. If pagination is in use, it is performed transparently as you iterate through the dataset.

Example 3.9. Iterating through all results

```
public static void main(String[] args) {
    DataProvider<Integer> provider = new IntegerDataProvider();
    ObjectDataset<Integer> ds = new Dataset<Integer>(provider);
    ds.setMaxRows(5);
    for (Integer i : ds) {
        System.out.println("Iterating over value "+i);
    }
}
```

Figure 3.4. Output from iterating over a paginated dataset

```
Generating results from 1 to 5
Iterating over value 1
Iterating over value 2
Iterating over value 3
Iterating over value 4
Iterating over value 5
Generating results from 6 to 10
Iterating over value 6
Iterating over value 7
Iterating over value 8
Iterating over value 9
Iterating over value 10
Generating results from 11 to 15
Iterating over value 11
Iterating over value 12
...
...
...
Iterating over value 89
Iterating over value 90
Generating results from 91 to 95
Iterating over value 91
Iterating over value 92
Iterating over value 93
Iterating over value 94
Iterating over value 95
Generating results from 96 to 100
Iterating over value 96
Iterating over value 97
Iterating over value 98
Iterating over value 99
Iterating over value 100
```

This could be useful in cases where you want to group fetches into batches if fetches are expensive, or maybe even incorporate some multi-threading to let one thread fetch data and other threads to process result sets as they come in.

3.3.3. Paging datasets

Generally paging through a dataset is done using the next and previous which changes the subset of data you are looking at. Generally speaking, using the iterator is the best way to iterate over every item in the dataset since paging is generally used to move to a specific page as opposed to iterating through the data. Iterating manually is fraught with corner cases that could lead to a number of errors.

Example 3.10. Attempting manual iteration

```
//BROKEN CODE, DO NOT USE
while (ds.isNextAvailable()) {
    List<Integer> res = ds.getResultList();
    showResults(res);
    ds.next();
}
```

What happens here is that when we move to the last page using `next`, the `nextAvailable` property is set to `false` and we exit the loop before we display it. We could process the results after the loop but what happens when you forget to add the code at the end? The best demonstration of this is where the size of the result set is smaller than the page size, and when we fetch the first page of results, `nextAvailable` is `false` since this is the first and only page, and so we don't display the list.

In general, use the dataset `next/previous/first/last` methods to navigate the dataset from page to page and the `Iterator` implementation to go through the dataset item by item^{*}. The iterator implementation handles all the corner cases in a familiar and consistent manner.

Note that in the event of you reaching the end of the dataset and calling `next`, the active page will not change and you will receive the last result set back. Unless the dataset is empty, there will always be results returned in the result list, and if there are no results, an empty list should be returned. The results should never be `null`.

Note

^{*} We do use this kind of incorrect paging through data in some of the test cases, but only where we are not testing the number of results, but the actual result content, and we want to test the `first/next/previous/last` methods as opposed to the iterator implementation.

3.3.4. Manually fetching data

The `ObjectDataset` is a helper class that manages the `Paginator` and `DataProvider` interfaces. It is easy to use these two classes separately since there is no complex glue code implemented by the `ObjectDataset` except where it has provided enhanced features. Using two separately instantiated paginators and providers is straightforward :

Example 3.11. Manually creating provider and pagination instances and fetching results

```
DataProvider<Integer> provider = new IntegerDataProvider();
Paginator paginator = new SimplePaginator(10);
List<Integer> results = provider.fetchResults(paginator);
showResults(results);
```

Each call to `fetchResults` executes the process to fetch the data. This can mean running a database query or parsing a file. The results are not cached unlike the `ObjectDataset`.

One important thing to bear in mind with using these components manually is that the `DataProvider` implementation is responsible for setting the `nextAvailable` property on the `Paginator` instance passed in to the `fetchResults` method. This means that when you first create a paginator, the `isNextAvailable` value is `false` so you cannot create a new paginator instance and call the `next()` method since it, as well as other functions, depends on the `nextAvailable` flag.

One benefit of using these standalone components manually is that you can use multiple paginator instances to provide different views of the same dataset independent of each other. This can be useful when you want application scoped data with local pagination.

Example 3.12. Multiple views of the same data

```
DataProvider<Integer> provider = new IntegerDataProvider();
Paginator paginator1 = new DefaultPaginator(10);
Paginator paginator2 = new DefaultPaginator(15);
paginator1.setFirstResult(20);
paginator2.setFirstResult(30);

System.out.println("First result = "+paginator1.getFirstResult());

List<Integer> results1 = provider.fetchResults(paginator1);
List<Integer> results2 = provider.fetchResults(paginator2);
System.out.println("Result Set 1");
showResults(results1);
System.out.println("Result Set 2");
showResults(results2);
```

Here we obtain two different sets of data from a single data provider. This can let you re-use data providers that obtain data from expensive data sources.

Chapter 4. The DataValve architecture

Now we've covered these fundamentals of using DataValve we can look at the different modules that DataValve includes. We will then look at the different subclasses of the `DataProvider` interface and implementations so we create datasets that can attach to different datasources.

4.1. List of Module

DataValve contains a number of modules, and is split into the main core and dataset modules and the more framework specific modules.

Table 4.1. List of Modules

DataValve-Core	Core module with the API interfaces and also includes in-memory and file based data providers. In general this contains providers and classes that are not dependent on external frameworks and also utility classes.
DataValve-Dataset	Contains the core classes for datasets as well as datasets for the
DataValve-Hibernate	Query data providers and datasets for the Hibernate framework
DataValve-JPA	Query data providers and datasets for JPA
DataValve-SQL	Query data providers and datasets for using SQL
DataValve-Faces	Provides features for use with Java Server Faces. This includes an EL parameter resolver, and JSF components for Pagination and column ordering links.
DataValve-Seam	Extensions for use with Seam, includes a Seam Jpa and Native datasets as well as a Seam parameter resolver that uses the EL resolvers included as part of Seam. It also includes an adapter for substituting a DataValve dataset for a Seam EntityQuery dataset. It implements most features except for sorting.
DataValve-Test	Includes classes for base test cases for testing data provider implementations. You should only need this if you are writing your own Data provider implementations. At the moment, it also includes the <code>TestDataFactory</code> class that is used to generated data. For that reason, this module is also used in most of the demo applications for generating data.

4.2. Data provider types

Most subclasses of the `DataProvider` are mirrored in some form of `ObjectDataset` subclass which acts as a thick client wrapper around the data provider and internally uses an instance of that particular subclass.

4.2.1. ParameterizedDataProvider

The first subclass of the `DataProvider` interface is the `ParameterizedDataProvider` that introduces methods for using dynamic parameters to provide parameterization features on datasets that were non-sql based. For example, in a data provider that features file listings, you could have a directory parameter that indicates the directory to list files from.

4.2.2. StatementDataProvider

The `StatementDataProvider` extends the `ParameterizedDataProvider` and adds the ability to define string based statements for fetching and counting the number of rows of data. Since this extends the `ParameterizedDataProvider` we can let the user define statements that have parameters within them. This obviously lends itself to easily providing the bulk of features of a `Sql` or `Ejbql` or `Hql` data provider that does not use ordering or dynamic restrictions.

4.2.3. QueryDataProvider

The `QueryDataProvider` extends the `StatementDataProvider` and adds the ability to define restrictions and ordering. This class is really intended for targeting data stores that have a SQL-like query language which can be parameterized. Typically, this is how you should reference query based data providers, and use the corresponding `QueryDataset` when referencing the `Dataset`.

4.2.4. AbstractQlDataProvider

This class provides the hooks for implementing the data fetch using a query language such as `Sql` or `Ejbql`. It does this by using an intermediate `DataQuery` object that contains the state of the query in an implementation-neutral manner. Subclasses then use the `DataQuery` to query the database using their specific data access mechanism.

4.2.5. AbstractQueryDataProvider

This class extends the `AbstractQlDataProvider` and implements the `QueryDataProvider` interface. It handles the creation of the `DataQuery` using a `DataQueryBuilder` instance. This builder takes a statement and restrictions and resolves the parameters and generates the final `DataQuery`. This query is then executed in the subclasses using the particular data access mechanism specific to that implementation.

4.3. ObjectDataset Types

`ObjectDataset` instances can help when using the data provider in a stateful manner by caching the current set of results. The basic `DataProviderDataset` class takes a basic `DataProvider` implementation as its data provider. However, this can lead to code requiring a lot of casting for a data provider that extends `DataProvider` and has custom attributes or methods, including extensions like `QueryDataProviders`.

Example 4.1. Poorly typed provider and dataset usage

```

QueryDataProvider<Person> provider = new JpaQueryProvider<Person>();
DataProviderDataset<Person> dataset = new DataProviderDataset<Person>(provider);

//
// Won't work, getProvider returns a DataProvider,
// not a query data provider. DataProvider doesn't
// have the query data provider methods on there.
//
dataset.getProvider().addRestriction("p.id = :param",300);

// Casting required
((JpaQueryProvider<Person>) dataset.getProvider()).addRestriction(
    "p.id = :param",300);

```

Rather than force the user to keep type casting the provider, there is a generic implementation of the dataset that takes the type of data provider it uses as a parameter. This lets you call the `getProvider()` in a type safe manner.

Example 4.2. Strongly typed data provider and dataset usage

```

//declare variables
QueryDataProvider<Person> provider;
Dataset<Person,QueryDataProvider<Person>> dataset;

provider = new JpaQueryProvider<Person>();
dataset = new Dataset<Person,QueryDataProvider<Person>>(provider);
//we can access the provider attribute as a QueryDataProvider
dataset.getProvider().init(Person.class, "p");
dataset.getProvider().addRestriction("p.id = #{personSearch.id}");
dataset.getProvider().addParameterResolver(new MyParameterResolver());

```

This is further extended by creating implementations for the different type of query interfaces we use. In this example, rather than create a generic `Dataset` with the provider class type as a generic type parameter, there is a `QueryDataset` type that extends the `Dataset` and sets the provider type parameter to `QueryDataProvider<T>`.

Example 4.3. Type safety with the QueryDataProvider

```

QueryDataProvider<Person> provider = new JpaQueryProvider<Person>();
QueryDataset<Person> dataset = new QueryDataset<Person>(provider);
dataset.getProvider().init(Person.class, "p");
dataset.getProvider().addRestriction("p.id = #{person.id}");

```

We still maintain polymorphism because the `QueryDataset` implementation can take any data provider that implements the `QueryDataProvider` such as the JPA, Hibernate or SQL providers.

This is the preferred way of referencing datasets, using pre-built typed versions since it allows you to change implementations much easier as you are not tied to a specific database access mechanism in the declaration. It also deters developers from using specific types and accidentally locking themselves in

Example 4.4. Examples of polymorphic provider references

```

QueryDataProvider<Person> provider = new JpaQueryProvider<Person>();

//this is verbose, and hard to read
Dataset<Person,QueryDataProvider<Person>> ds = new
    Dataset<Person,QueryDataProvider<Person>>();

//this is bad code as you are locked in to the JpaQueryProvider type
JpaDataset<Person> ds = new JpaDataset<Person>(provider);

//this is double bad as you are locked in to the JpaQueryProvider type and it
//is verbose
Dataset<Person,JpaQueryProvider<Person>> ds = new
    Dataset<Person,JpaQueryProvider<Person>>();

//this is usable if you don't want access to features of the QueryDataProvider,
//It also allows a large degree of polymorphism since you can use any provider
ObjectDataset<Person> ds = new Dataset<Person>(provider);

//this is short, type safe, and polymorphic as far as QueryDataProviders go
QueryDataset<Person> ds = new QueryDataset<Person>(provider);

```

4.4. ObjectDataset usage

You might find yourself using the stateful `ObjectDataset` most of the time. Even in web applications where you might have a stateless page, when rendering the page, the method to return the query results might be called a number of times as it is rendered. If you used a data provider to obtain the results, they would have to be regenerated on each call (unless you manually held the results somewhere). Using a dataset to hold the results will make the results stateful, even if only for the duration of the web request. This same feature could be used in other stateless web environments. There is little overhead in using an `ObjectDataset` implementation instead of separate provider and paginator instances.

If you use Seam, Spring web flow, CDI or other frameworks that allow state to last beyond the current request, or even just putting it in the session, you can extend the life of the query over multiple requests which will save you having to re-query the data next time it is needed.

4.5. Choosing a dataset type

In general, the rules for choosing a dataset type are fairly simple. In factory or construction code, where you construct and configure the data provider you can be as specific in the class type as you want. However, it is best to refer to it in the client code as far down the hierarchy as possible. Typically, for things like search forms, you might use a factory to construct a `JpaDataProvider` and set the `entityManager` attribute and return it as a `QueryDataset`. In your client code, you can also refer to it as a `QueryDataset` and still gain access to the general features of the query (restrictions, sorting, pagination, and even select/count statements). You won't however be able to access the entity manager attributes. If you later decide to switch to hibernate or sql, even for just this one query, you just need to change the type specific construction code in the factory. The client code, as long as it is referencing it as a `QueryDataset` can stay the same.

If you are using few features in the client code, you could return the dataset as a basic `DataProviderDataset` which limits the functions you can use on it such as no access to

`QueryDataProvider` related features. However, it does make it much more polymorphic, you can swap out a `QueryDataProvider` based data provider for something else, in fact anything that implements the `DataProvider` interface. You could even switch to a provider that gets its data from a text file.

4.6. Extending `DataProvider` implementations

When creating a new data provider implementation, you are most likely to want to subclass from the `AbstractDataProvider` class rather than start from scratch implementing the `DataProvider` interface. To implement your own data provider you can override the following methods.

4.6.1. `void doPreFetch()`

This method lets you alter the state of the provide prior to fetching either the result count or the actual results.

4.6.2. `Integer doFetchResultCount()`

Returns the actual number of rows in the dataset available to the user. This takes into account any filtering or restrictions that might be applicable.

4.6.3. `List<T> doFetchResultCount()`

Returns the actual results from the source dataset to the user. This takes into account any filtering or restrictions that might be applicable.

4.6.4. `Integer doPostFetchResultCount(Integer resultCount)`

This method is called after the result count has been fetched and the result from this method is the value returned to the user. By default this method just returns the count value passed in.

4.6.5. `List<T> doPostFetchResultCount(List<T> results, Paginator paginator)`

This method is called after the results have been fetched and they are passed into this method along with the pagination info used to fetch the results. This method allows for any post-processing of the results. This method returns the results that are then passed back to the user. By default this method just returns the list of results passed in with no changes.

4.6.6. Extending data providers

You would normally only need to override the `doFetchResults` and the `doFetchResultCount` methods to implement the provider adequately.

Application developers will find more use in the pre and possibly the post fetch methods. In most cases, developers will use the `doPreFetch()` to perform any last minute configuration of the provider before fetching the results. This may mean altering restrictions based on application state, or even altering parameter values.

Beyond that, there shouldn't be many more ways developers need to extend the provider classes between the use of composition and extension points that can be overridden. The goal is to make it open for extension, closed for modification. If you feel there are other ways we can improve this, or other extension points need adding, get in touch and we'll see what can be done.

4.7. Extending datasets for custom providers

When you create a new data provider implementation it is also good practice to implement a dataset wrapper for it so your users do not have to implement their own, or keep using the generic versions. Doing so is very simple, just create a new dataset class that extends the `Dataset` and uses your own custom provider type as the type parameter.

Example 4.5. Writing datasets for custom providers

```
class MyCustomProvider<T> extends AbstractQueryDataProvider<T> {
    ...
    ...
    //this is specific to this implementation
    public void setWidgetConnection(WidgetConnection connection) {
        this.connection = conn
    }
}

class MyCustomProviderDataset<T>
    extends Dataset<T,MyCustomProvider<T>> {

    public MyCustomProviderDataset() {
        super();
    }

    public MyCustomProviderDataset(MyCustomProvider<T> provider) {
        super(provider);
    }
}
```

The dataset name should consist of the provider class name with the word dataset appended on the end to make it easy to remember. By including this class, it is easier to use datasets for custom data providers, especially for custom attributes on your new custom provider.

Example 4.6. Using your new data provider dataset

```
MyCustomProvider<Person> provider = new MyCustomProvider<Person>();
MyCustomProviderDataset<Person> ds = new MyCustomProviderDataset<Person>(provider);
ds.getProvider().setWidgetConnection(ConnectionFactory.createConnection());
```

Note that while this provides accessibility to implementation specific functions, it does make your code more type specific and tightly bound. This has the effect of losing the ability to use polymorphism and making your code more tightly coupled to specific classes. To avoid this problem, only use more concrete class types in factory methods and return a more generalized provider or dataset type from the factory to the main application as discussed in the previous section.

Example 4.7. Using polymorphism to avoid type lock-in

```
QueryProvider<Person> buildProvider() {
    MyCustomProvider<Person> result = new MyCustomProvider<Person>();
    result.setWidgetConnection(xyz);
    return result;
}

QueryDataset<Person> buildDataset() {
    MyCustomProvider<Person> provider = buildProvider();
    MyCustomProviderDataset<Person> ds = new MyCustomProviderDataset<Person>();
    ds.getProvider().setWidgetConnection(ConnectionFactory.newConnection);
    return ds;
}

...
...
//assign the dataset using the generic dataset
QueryProvider<Person> provider = buildProvider();
QueryDataset<Person> ds = new QueryDataset<Person>(provider);
ds.getProvider().addRestriction("p.id = :param",id);

//you can even assign custom datasets the the generic query dataset type
QueryDataset<Person> ds2 = buildDataset();
```

Chapter 5. Non-database providers

There are a number different non-database sources you can choose to fetch data from and usually you have no common way to get that data into your application. DataValve provides the ability to fetch contents from these sources and transform them to pojos. It does this by abstracting the concepts of fetching data to a level that is higher than retrieving it from the database. As a result, there are a number of different implementations that fetch data without a database while letting you benefit from using the same interfaces to control the data access.

5.1. In-memory based datastores

The simplest form of non-database datastore is one which just keeps the a list of objects in memory and the whole or partial set is returned to the user when requested. An in-memory data store subclasses the `InMemoryDataProvider` class and implements the abstract method to return the backing data. The backing data is the complete list of in-memory results that the data provider is paginating. Here is a data provider that contains a list of persons

Example 5.1. In-memory person data provider.

```
public class InMemoryPersonProvider extends InMemoryDataProvider<Person> {

    private List<Person> backingList;
    private static final int PERSON_COUNT = 100;

    @Override
    protected List<Person> fetchBackingData() {
        //generate some random test data
        if (backingList == null) {
            backingList = new ArrayList<>(PERSON_COUNT);
            for (int i = 0; i < PERSON_COUNT; i++) {
                Person p = new Person(new Long(i + 1), TestDataFactory
                    .getFirstName(), TestDataFactory.getLastName());
                backingList.add(p);
            }
        }
        return backingList;
    }
}
```

We can now iterate through the data from this provider as though it were a database driven provider. `InMemorydataProvider` result sets can also be sorted by the use of `Comparator` implementations. Like the database driven providers, the in memory providers have an order key except the key is mapped to a comparator instance which is used to order the data in the list.

Example 5.2. Sorted in-memory person data provider.

```

public class InMemoryPersonProvider extends InMemoryDataProvider<Person> {

    private List<Person> backingList;
    private static final int PERSON_COUNT = 100;

    private class PersonIdComparator implements Comparator<Person>, Serializable {

        public int compare(Person o1, Person o2) {
            return (int) (o1.getId() - o2.getId());
        }
    }

    private class PersonNameComparator implements Comparator<Person>,
        Serializable {

        public int compare(Person o1, Person o2) {
            int val = o1.getLastName().compareToIgnoreCase(o2.getLastName());
            //if last names match, then try comparing by the first name
            if (val == 0) {
                val = o1.getFirstName().compareToIgnoreCase(o2.getFirstName());
            }
            return val;
        }
    }

    public InMemoryPersonProvider() {
        //add the order keys by default in the constructor
        getOrderKeyMap().put("id", new PersonIdComparator());
        getOrderKeyMap().put("name", new PersonNameComparator());
    }

    @Override
    protected List<Person> fetchBackingData() {
        //generate some random test data
        if (backingList == null) {
            backingList = new ArrayList<>(PERSON_COUNT);
            for (int i = 0; i < PERSON_COUNT; i++) {
                Person p = new Person(new Long(i + 1), TestDataFactory
                    .getFirstName(), TestDataFactory.getLastName());
                backingList.add(p);
            }
        }
        return backingList;
    }
}

```

To test this, we can subclass the `AbstractObjectDatasetJUnitTest` to create new test cases for this class. We add in additional tests for testing that the ordering is working correctly.

Example 5.3. Testing the InMemoryPersonProvider

```

public class InMemoryPersonProviderTest
    extends AbstractObjectDatasetJUnitTest<Person> {

    @Override
    public ObjectDataset<Person> buildObjectDataset() {
        InMemoryPersonProvider provider = new InMemoryPersonProvider();
        return new Dataset<Person>(provider);
    }

    @Override
    public int getDataRowCount() {
        return 100;
    }

    public void testIdOrdering() {
        ObjectDataset<Person> ds = buildObjectDataset();
        ds.setOrderKey("id");
        ds.setOrderAscending(false);
        ds.setMaxRows(10);
        //check descending
        Long id = null;

        for (Person p : ds) {
            if (id != null) {
                assertTrue("Next Id is not less than the last", p.getId() < id);
            }
            id = p.getId();
        }

        //now reverse the ordering
        ds.setOrderAscending(true);
        id = null;
        for (Person p : ds) {
            if (id != null) {
                assertTrue("Next Id is not greater than the last", p.getId() > id);
            }
            id = p.getId();
        }
    }
}

```

In this test case, we toggle the ordering to ensure that the ordering is working as we switch from one order to the next. One caveat with ordering in memory providers is that the ordering takes place on the backing data list in the data provider. Therefore, you can't directly have multiple paginators that fetch independent views of the data since they would all have to share the same sort order. You could do this if there was no ordering or they used the same order. However to get around this you could use an `InMemoryAdapterProvider`.

5.2. InMemoryAdapterProvider

The `InMemoryAdapterProvider` wraps an existing `DataProvider` (any kind, not just the in-memory kind) from which it sucks up all the data and puts it in its own in-memory list. This can

be used to either make a data provider a fast in-memory provider, or create an independent view of an in-memory dataset with independent sorting.

While the `InMemoryAdapterProvider` keeps a separate list of the data elements, it does not keep its own copy of the actual data which is shared with the original provider. This lets you use a single copy of the all data, but lets you have independent lists of the data. Obviously, care must be taken when using this on a database since it will load all the data from the source data provider into memory.

Example 5.4. Multiple views using an in-memory adapter.

```
InMemoryPersonProvider provider = new InMemoryPersonProvider();
InMemoryAdapterProvider<Person> providerAdapter =
    new InMemoryAdapterProvider<Person>(provider);

ObjectDataset<Person> ds = new Dataset<Person>(provider);
ObjectDataset<Person> ds2 = new Dataset<Person>(providerAdapter);

//shared data, independent ordering and paging
ds.setOrderKey("name");
ds2.setOrderKey("id");
```

The `InMemoryAdapterProvider` will automatically pick up the order key comparators if it wraps an `InMemoryDataProvider` which it does in this case.

5.3. IntegerDataProvider revisited

Here is our `IntegerDataProvider` reimplemented using the in-memory data provider. All we need to do is override the method to return the backing data and return our list of integers from 1 to 100.

Example 5.5. In-memory implementation of our Integer provider.

```
public class InMemoryIntegerProvider
    extends InMemoryDataProvider<Integer> implements Serializable {

    @Override
    protected List<Integer> fetchBackingData() {
        List<Integer> result = new ArrayList<Integer>();
        for (int i = 1; i <= 100; i++) {
            result.add(i);
        }
        return result;
    }
}
```

We can also easily test this using our `IntegerDataProviderTest` class that we used to test the integer data provider last time by subclassing it and overriding the `buildObjectDataset` method. This time we return an instance of our in-memory Integer data provider.

Example 5.6. Testing our in-memory Integer data provider.

```
public class InMemoryIntegerProviderTest extends IntegerDataProviderTest {

    @Override
    public ObjectDataset<Integer> buildObjectDataset() {
        DataProvider<Integer> provider = new InMemoryIntegerProvider();
        return new Dataset<Integer>(provider);
    }
}
```

In our previous implementation of the Integer data provider we neglected to include sorting, so this time we'll add sorting into this implementation. Since our integer in-memory data provider only has one comparator to order by (for comparing two integers), we could just override the `translateOrderKey` method to return the single comparator in our dataset. This saves us from making the user assign an order key value, and it will always be sorted by this comparator.

Example 5.7. Sorting the Integer data provider.

```
public class InMemoryIntegerProvider extends InMemoryDataProvider<Integer>
    implements Serializable {

    private transient Comparator<Integer> comparator = new Comparator<Integer>() {
        public int compare(Integer o1, Integer o2) {
            return o1-o2;
        }
    };

    @Override
    protected List<Integer> fetchBackingData() {
        List<Integer> result = new ArrayList<Integer>();
        for (int i = 1; i <=100; i++) {
            result.add(i);
        }
        return result;
    }

    @Override
    protected Comparator<Integer> translateOrderKey(String key) {
        return comparator;
    }
}
```

Note also that in this case, the comparator field is marked `transient` since it is not serializable.

5.4. File Based Data Providers

One common need is to load a set of from one or more files which can be either text based, comma delimited, xml or some proprietary format. DataValve provides the same interface to access these data sources, and therefore use the same client code to access them. At the root of file based data providers, we have the `AbstractFileBasedProvider` which takes a filename in its constructor and creates the `File` instance. This is subclassed into different implementations.

5.4.1. TextFileProvider

The `TextFileProvider` is the base class for data providers using text files as the data source. It treats each line in the file as a row of data and calls the abstract method `createObjectFromLine(String line)` to generate an instance of the object data for each row in the file. This is further extended in the `CommandDelimitedProvider` which assumes that each row of text is a row of comma delimited data. For each row of data, the provider splits up the data elements and passed them as a string array to the abstract method `createObjectFromColumns(String[] columns)`. Subclasses need to implement this method to return the specific type of entity that this data provider returns.

Chapter 6. Odds and Ends

This sections covers any additional info not included anywhere else.

6.1. IndexedDataProviderCache

The `IndexedDataProviderCache` provides random access to provider data with look ahead caching to optimize querying and take advantage of data coherence. The best example of this is in the swing table which is included in one of the examples. A swing table model knows the number of rows of data that exist (which could be thousands or millions) and lets the user scroll across the whole dataset. This means that the user can select any record from the available dataset at any point in time. In this event, it would be fetched from the data provider and returned to the user. However, the user will probably end up fetching the next x number of objects from the dataset as the next x number of rows in the table are displayed. For this reason, the cache not only fetches the requested row, but the next n number of rows where n is the batch size defined by an attribute on the cache. The `IndexedDataProviderCache` keeps hold of these items since they may well be reused when the table is repainted or the user starts to scroll backwards or forwards a few records at a time. This cache is limited in size, and records will start to be ejected on a least recently used basis. The `IndexedDataProviderCache` can be used to randomly browse thousands of records with no startup time and efficient memory use since it doesn't read all the data in at once. (The delay at the start is due to the creation of the database, not the loading of all the data).

6.2. Don't call record count

You should not avoid calling `fetchRecordCount` from within the data provider. The most likely place you will call it is to determine whether there are more results to be returned. This can be an expensive call to count all records or rows of data from the source that should be avoided if possible. In the database data provider included, we fetch an extra row to see if there are more results after the set of rows we want. This extra row is then taken off the final results that are returned to the user.

6.3. Seam Usage

With `DataValve` you can subclass either a `SeamJPaQueryDataset` or a `SeamJpaNativeDataset` so you can use either a `Ejbql` or `Native` query for fetching data. Also, there are adapter classes that can be used instead which adapts the interface to look more like an `EntityQuery` so you can just substitute one for the other.

6.4. DataValve and JSF

`DataValve-Faces` is a module for working with `DataValve` in JSF. It provides a `Expression Language (EL)` parameter resolver so you can use EL expressions directly in your queries. It also provides visual components for creating a sort link for clickable sortable columns and a simple paginator class.

For an example of using the `datavalve-faces` module in a servlet environment using CDI and JSF, take a look at the `cdi demo` application in the samples directory.

6.5. Executing code before a fetch

There are many times you may want to execute some code prior to executing a fetch (either record count, or actual results). For example, you may want to clear and re-add your restrictions

if you are manually adding them. The `doPreFetch()` in the `AbstractDataProvider` class allows you to execute code prior to a fetch.

Example 6.1. Setting up the query prior to fetching.

```
public class SomePersonSearchProvider extends HibernateProvider {

    private String firstName;
    private String lastName;

    @Override
    protected void doPreFetch() {
        getRestrictions().clear();
        addRestrictionStr("p.firstName like :param", firstName+"%", firstName);
        addRestrictionStr("p.lastName like :param", lastName+"%", lastName);
    }
}
```