**Project Overview**

The idea of ValueListService is simple.

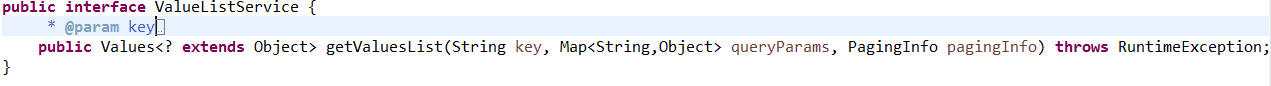
1. Configure in XML SQL queries to run.
2. Send HTTP request and get query results in JSON.

The *valuelistservice.jar* provides a flexible framework to support queries to filter, sort, and page through large result sets.

The full project is available at: <https://github.com/ryanp102694/valuelistservice>

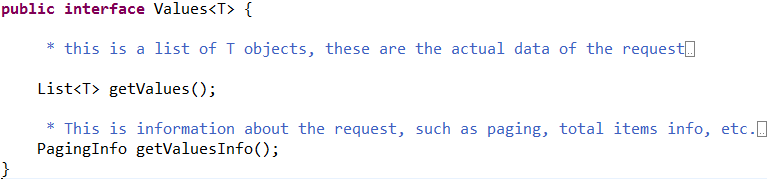
**Interface Overview**

The **ValueListService** interface is very simple, it contains one method to get back a **Values** object which holds results.



The intention of these parameters is the following.

1. **key:** Used to identify a query which the ValueListService is capable of running.
2. **queryParams:** Parameters used to by the query specified by **key.**
3. **pagingInfo:** An object which encapsulates requested page, and number of items per page for specified query. (This can be left null if paging is not necessary)

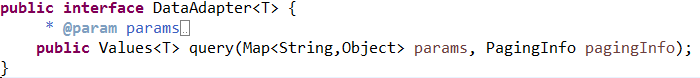
The method returns **Values<? extends Object>**. Let’s break that down.

As you can see, a Values is nothing more than a List and paging information, which tells the client the total number of items, page, and items per page.

**Using Default Implementations**

The **DefaultValuesListService<T>** is a simple implementation of **ValueListService**. The default service takes advantage of another interface, **DataAdapter<T>**. Essentially the default service has a map of adapters, which are accessible by the query key mentioned earlier. The adapters are actually what run the queries, and return the appropriate type from the service.

Let’s take a look at **DataAdapter<T>.**



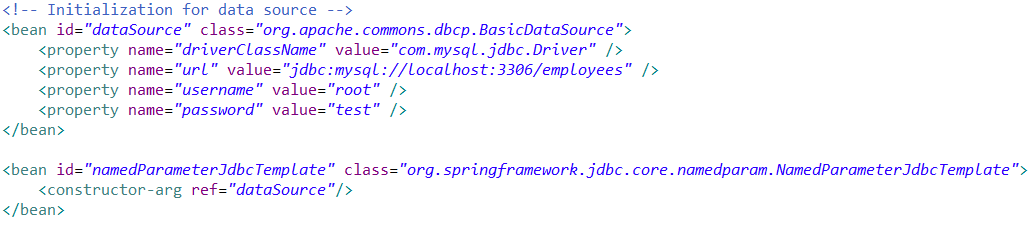
Here we can see that the adapter returns the same type as the service does, in fact, the method is almost exactly the same as the service method, except that the **key** parameter is removed.

The intention of a **DataAdapter<T>** implementation is to be able to allow different queries to return different types, as well as different queries to connect to different data sources. For example, we could have a *FileSystemDataAdapter*, *JdbcDataAdapter*, or even *MongoDBDataAdapter*.

In fact, valueslistservice.jar includes a JDBC adapter, **DefaultJdbcDataAdapter**. Let’s see how we can easily configure this in Spring to read data from a MySQL database.

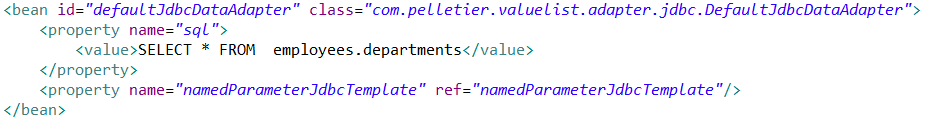
1. **Configure a DataSource and NamedParameterJdbcTemplate.**

It is important to use NamedParameterJdbcTemplate for reasons we will see later.



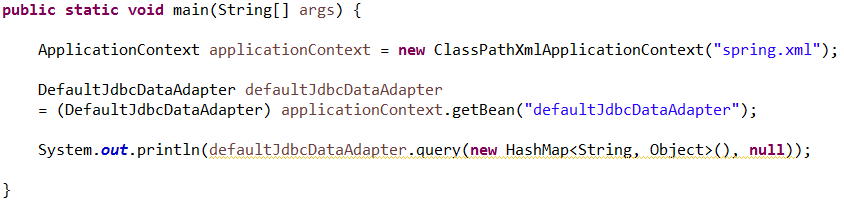
1. **Configure DefaultJdbcAdapter**

Here I will be querying a sample MySQL table for employee department names.



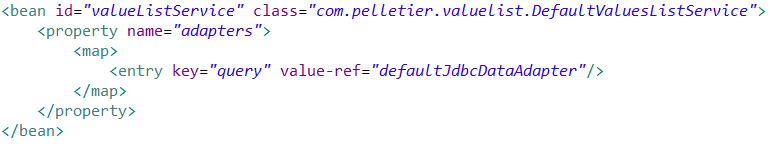
1. **Call standalone adapter.**

You can use adapters apart from the service if you wish, in the following example program, we see our adapter query method producing the expected results. By default the results are of the type List<Map<String,Object>> due to Spring’s **ColumnMapRowMapper**. Optionally, a **RowMapper**<T> can be injected into the adapter to return a different type.



1. **Inject adapter into default ValueListService.**

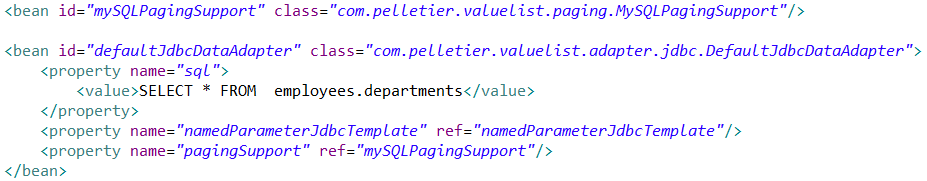
Ultimately, we will want to use the service, rather than the standalone adapter. The **DefaultValueListService** can be configured with the following XML.



Additional adapters can be injected into the map, which are able to be accessed by key in the actual service.

**Paging**

Some result sets will be extremely large. The user may want only the first 1000 results, but they likely want to know the total number of results that match their query. By passing a **PagingInfo** object to our service, the adapter will correctly paginate our query. Since we are using MySQL, we inject **MySQLPagingSupport** into our adapter. When finished, the adapter configuration will look like this:



Now, we can create a **PagingInfo** object, set the *numberPerPage* property, and the *page* property, and pass it to the service. The result will only show the correct information for the selected page.

**Complex Queries**

By default, the **DefaultJdbcDataAdapter** will use a **VelocityQueryParameterMapper** to pre-process any of the SQL that is run by Spring on the database. This allows the user to write the SQL as a velocity template, knowing that Velocity will attempt to map the parameters to the SQL.

Let’s write a query that takes advantage of velocity, as well as the **NamedParameterJdbcTemplate**.



Suppose we pass in a map with our parameters which includes (“dept\_name”, “Customer Service”).

Two things will happen before the SQL is actually run.

1. Velocity will process the SQL, transforming it into:

*SELECT \* FROM employees.departments WHERE dept\_name = :dept\_name*

1. The Spring template will insert the value of the parameter in place of <:dept\_name> The final query becomes:

*SELECT \* FROM employees.departments WHERE dept\_name = “Customer Service”*

**Using Velocity templating allows the defined queries to be flexible and allow filtering and sorting based on the existence of certain parameters.**

**Further Extension**

If necessary, classes can be written which build upon the existing interfaces. For example, if a user wishes to not use Velocity templating. They could create a class which implements **QueryParameterMapper**, and inject that into the adapter. Or suppose Microsoft SQL Server is being used. It may be necessary to implement the **SQLPagingSupport** interface to support paging.

**Using ValueListService in a Web Service**

The original purpose of this project was to get JSON results. An example of this being done is in a Spring MVC project with Jackson JSON processing.

An example service can be found here: <https://github.com/ryanp102694/valuelistservice_webservice>

Let’s take a look at our **ValueListServiceController**, which is really the only code in the project.

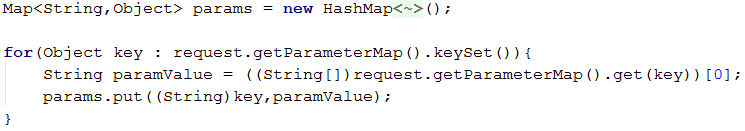


Basically, this is what is going on here.

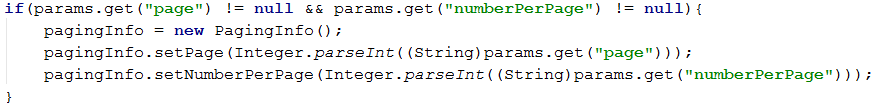
1. The **ValueListService** is injected into the controller using the @Autowired annotation. The service is defined in a Spring xml file.



1. Parameters are taken from the request and put into a parameter map.



1. A **PagingInfo** object is constructed from specific parameters *page* and *numberPerPage*.



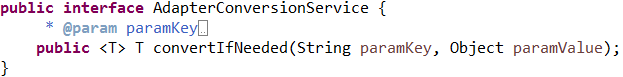
1. **ValueListService** is called, getting the Values. The Values are then marshalled to JSON by the Jackson library, before being sent in the response body.



When using **ValueListService** in a web service, a problem arises. **Query parameters in an HttpServletRequest are Strings!** What does this mean when we need to pass a date in a format like “1994-10-27”? After all, Spring needs a java.util.Date to run the correct SQL query.

**Configuring the AdapterConversionService**

The **DefaultJdbcDataAdapter** has a property **AdapterConversionService**. When configured, the adapter will convert parameters from one type to another, based on the adapter requirements.



By default, the adapter uses **ParameterConversionService**, which extends Spring **DefaultConversionService**. **ParameterConversionService** has a Map<String,Class>, as well as a list of **Converter** objects. When the **ParameterConversionService** calls its implemented method, it checks to see if the name of the parameter is in its map, if it is, it will use a **Converter** from its registry to attempt to convert the parameter value to the target class type.

Let’s see an example of how to configure this. Here is the configuration for a **ParameterConversionService** bean. 

Here we can see that we have set up query parameter maps with keys “birth\_date” or “hire\_date” to have their values converted to java.util.Date. We have also implemented Spring’s **Converter<S,T>** interface in the **DefaultStringToDateConverter** class. This converter is then added to the conversion service’s list of converters. This bean will be injected into the adapter’s adapterConversionService property.

**Note:** Spring’s MapFactoryBean and ListFactoryBean can be set up as parent-child beans, so that their lists and maps inherit values. This is useful if a global type conversion map or converters list must be configured.

Our adapter can now convert parameters from one type to another in order to run its query.