sipXconfig Search Integration

this is work in progress

Using search

sipXconfig UI (starting in 3.1) has ability to easily locate various configured items (users, phones, groups, conferences etc.)
thanks to Lucene based indexing you can pretty much look for everything

This is sipXconfig attempt to jump on "search not sort" bandwagon. sipXconfig does support grouping and sorting if you happen to be search averse but a lot of administration activities rely on locating affected items quickly and we want to support it too.
For example changing PIN for a user you type user name, or alias and you quickly end up with link to user edit page. Also I hope that global search will encourage people to use description fields - since you can now quickly search their content.

The names of the items on the search result pages are hyperlinks: click on them and you'll be taken to edit page for a found item.

Tip If you want to locate an item quicky use the Description field of an item to enter something that you'll later search for.

Search-able items

The following items are currently indexed (and thus search-able) by sipXconfig:

- users and users' groups
- phones and phones' group
- hunt groups
- gateways
- dial plan rules
- call park orbits
- auto attendants

Query syntax

In most cases you can search just by entering a word or a number, but sipXconfig exposes rich Lucene search syntax:

- ```
  gre*
  ```

  will find all items that have words starting with greg
- ```
  name
  greg
  ```

  : will only search in name fields
- ```
  greg and not (description
  office)
  ```

  : will find all items named greg that do not have office in the description etc.

In order to use global search enter your query in the Search box, which is located to the right of navigation bar press Enter. You will be presented with the search result page that displays the number of items found, name and a short description of each item. Usually the results may comprise multiple types of items. You may see users, phones and hunt groups.

Adding search support

5 minutes minimum

In order to be search-able your beans need to inherit from BeanWithId class (if you cannot do it there is a way around it but it requires a lot of work) and it has to have an Edit page so that user has something to click on once your bean is found.

- add the class of your object to the list of indexed classes in DefaultBeanAdaptor
- add the information about the edit page of your bean to EnumEditPageProvider - you need a page name and the name of the id property which should represent the id of the object that page will load to edit
- add the user readable name for you component class in SearchPage.properties - if you do not do that your elements will be displayed on the result page as "Item"
- run the tests
  ```

Before you try searching for your objects remember to recreate index.

You do not really have to read or understand the following sections, but it may help if you have problems.

sipXconfig search implementation

Lucene is a general purpose library for indexing and searching. This description presents one of the many possible ways it can be used to index database accessed through Hibernate. The implementation was inspired by this code poet blog entry.

sipXconfig is using Lucene to implement its search functionality. Lucene is based on the concept of search-able index. In sipXconfig index is updated every time when items are added to database and every time it is modified. Every indexed entity (user, phone, hunt group etc.) is a "document" in the Lucene understanding of this word. The document is identified by its Java class and hibernate/database identifier: which allows us to load the object from the database when Lucene returns search results.

Maintaining index

Lucene index files are kept in {prefix}/var/sipxdata/tmp/index directory. If you delete this directory index will be automatically recreated. You should use:

```
sipxconfig.sh --database drop-index
```

command to delete the index.

In order to modify Lucene index sipXconig employs Hibernate interceptor architecture to monitor database operations and modify index if necessary. At the moment there are two Indexers implemented. Bulk Indexer is only called in demand by IndexManager when there is a need to create or completely rebuild the index (this usually happens if index has been removed). Fast Indexer is called on most of the database modifications operation - it is used by IndexingInterceptor responsible for monitoring Hibernate database operations. By now it should be clear that any modifications that are by-passing hibernate will not be reflected in the index. This may happen in testing environment when we inject data directly to the database. If it somehow happens in production environment the simple solution is to recreate the index.

There reason for having 2 indexers is that Bulk Indexer does not have to worry about removing anything from the index (it is only used to create a new index) and it can keep Lucene index opened for a longer time (instead of opening and closing it when indexing each and every item). On the other hand Fast Indexer is called relatively often: its job is to modify the index for a single item.

Both indexers defer most of the work to BeanAdaptor object. BeanAdaptor is responsible for creating a Lucene "document" for a bean that is being created, modified or deleted. Indexers are responsible for adding (or removing) such document in Lucene index.

At the moment we only have a single BeanAdaptor called DefaultBeanAdaptor. If you want to add support for indexing yet another bean this is the class that needs to be modified. Among other things BeanAdaptor is called to determine if a given Java class should be indexed - hibernate would call interceptor whenever any of the hibernate entities is updated. We only want to index relatively small subset of the all possible entities. DefaultBeanIndexer just looks up the name of the class (and the names of this class super classes] on the hard-coded list to verify if the bean should be indexed.

The other part of the BeanAdaptor job is to create a Lucene document from bean data. Documents are simple structures that keep parsed values of the bean fields. DefaultBeanAdaptor is responsible for making a decision which fields of the bean should be the part of Lucene document and if the field should be just indexed or indexed and stored in the index. The difference between an indexed field and a stored field is that you can get the value of the stored field directly from the index, whereas to retrieve the value of the field that was only indexed and not stored you'll need to get the bean from the database.

This is the simplified description of how the DefaultBeanAdaptor behaves:

It checks the value of the all fields for the bean:

- if value is a String it will be indexed, none other fields are indexed (or stored)
    - if the name of the field is on the FIELDS list it is added to the index as a separate field/column - which means that you can sort by it, and use its name in the query explicitly (for example one can search for all the phones using serial number 001122334455 using query serialNumber:001122334455)
    - otherwise the field is still indexed as a part of "all" field - it's value however is not stored in the indexed meaning that you can use it in search but you cannot display it on search result page without actually loading a found object from a database

The aliases are treated slightly differently - they are indexed when their owner is indexed so that when we look for an alias we actually find an object owner.

The names of the fields are the property names, and not the SQL database column names.

Displaying search results

The searching is an easy part. SearchManager implements several functions that you can call to retrieve search results. Most of the functions take a jakarta collections compatible Transformer parameter. If transformer is provided, it'll be used to convert each search result (hit). Typically you would use transformer to load the objects from the database, however searching is noticeably faster if you do not have to load the beans.

The difficult part of the search result presentation is locating the Edit page that corresponds to the found object. I could not come up with any smart mechanism to implement it. I settled on SearchPage deferring the work of identifying the edit page to EditPageProvider. The default implementation looks up the Edit Page name by class name of the found object (super classes are also taken into account). Since edit page needs to know which bean it has been called upon to edit, you have to tell page provider which property represents bean id.