Developing Applications with <b-frame/>

A Tutorial

Peter Belkner (peter.belkner@it-fws.de)
Markus Dahm (markus.dahm@it-fws.de)

Version 0.2
Dec 16, 2003

1 Introduction

<b-frame/> (http://www.b-frame.org/) is a framework generator designed to enable rapid prototyping and development of J2EE applications. The main property of <b-frame/> it comparison to other tools is that it works *declaratively*. The only thing the developer has to do is to *describe* her application in an XML file. Starting from that point, all the gory details such as creating the database, writing hundreds of Java classes, tag libraries, JSPs, deployment descriptors, compiling the Java classes and writing an ANT script to assemble everything is performed by <b-frame/>. Once the developer has finished describing his application the only thing left to do is to issue two commands, one for creating the database and the other one for deploying the application to the application server. Afterwards you can immediately view the application using your favorite web browser.

By default the applications created with <b-frame/> are full blown J2EE Web applications (known as Model II or MVC architecture). They consist of a JSPs and Tag libraries implementing the view, a Servlet as the controller communicating with EJB Session Beans implementing the transactional business logic. This version of <b-frame/> introduces the possibility of configuring the application for different approaches. For instance, one can now use entity beans for data storage instead of Session Beans (used as so called “Data Access Objects”). This enables the developer to switch between different implementation strategies on the fly. The generated code may be configured by custom code supplied by the developer.

1.1 Tutorial

This tutorial is designed as a step by step guide through the <b-frame/> development process using a simple sample application taken from university’s life. (We apologize however that the model rather reflects a German university than one in the English tradition.)

Note: It is assumed that you have some basic knowledge about developing J2EE applications with Servlets, JSPs and Enterprise Java Beans. We also assume that you already have installed a recent JDK, that you either use the complete <b-frame/> distribution (including a stripped JBoss 3.2.2, Ant and the <b-frame/> examples) or have installed and configured the aforementioned tools somewhere else.
The example here uses the Hypersonic database integrated into JBoss by default. For real world applications however, you will want to use a different database. How to deal with other databases, e.g., MySQL or Oracle, is described in chapter 10 at the end of this tutorial.

2 The Sample Application

Consider a simple application for planning and maintaining courses held at a university as shown in figure 1.

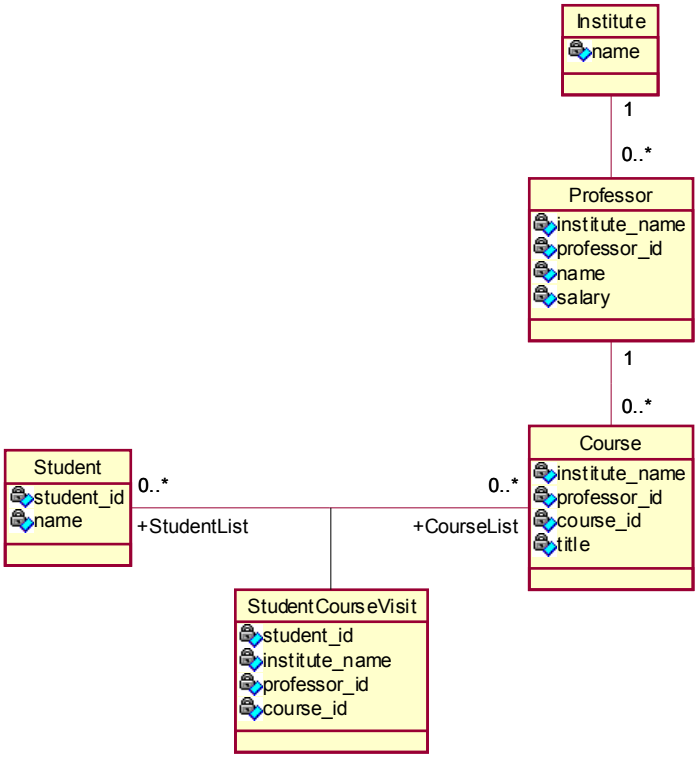


Figure 1 Planning a university's courses.

The application consists of the following business classes:

- A **Student** visits several *Courses*.
- An **Institute** is formed by several *Professors*.
- A **Professor** belongs to exactly one *Institute* and may offer several *Courses*.
- A **Course** is held by exactly one *Professor* and may be visited by several *Students*.

Based on these business classes we will create two *tasks*:

- The **Student Task** is for maintaining the university's *Students*. It will enable the user to relate *Students* with *Courses*. Before it is possible to assign a *Student* to a *Course* however it will of course be necessary to first create the *Course* using the *Institute Task*.
- The **Institute Task** is for maintaining the university's *Institutes*, their *Professors* and the *Courses* held by the *Professors*. It will enable the user to list and maintain all *Students* visiting a particular *Course*. Before it is possible to assign a *Student* to a *Course* it will of course be necessary to first create the *Student* using the *Student Task*.

Following an incremental step-by-step guide we will show how to create the *Student* and the *Institute Applications* using <b-frame/>

I. The *Student Application* without the possibility to assign a *Student* to a *Course*, i.e., it is only possible to list and maintain the *Student* business class.
II. The *Institute Application* without *Professors* and *Courses*.
III. The *Institute Application* with *Professors* but without *Courses*.
IV. The *Institute Application* with *Professors*, *Institutes* and *Courses*, but without the possibility to assign *Students* to the *Courses*.
V. The complete *Student* and *Institute Applications*.

But before we can start you should make sure that your environment is set up properly.

3 Setting Up the Environment

Before you can start to develop an application using <b-frame/> you should check some properties used for the build process first. There are two important folders: First, the <b-frame/> base directory named `b-frame`, you should not have to touch the contents of this directory. Second, there is the example's directory `b-frame-examples`. We will use this as the start point for our applications, you may use it as a template for your own future projects.

Since we're using the Ant tool to control the build process the important properties are expected in a file `"build.properties"` located in the `b-frame-example` directory. <b-frame/> provides a template `"build_example.properties"` which you should copy to `"build.properties"` before editing.

```
# Where is JBoss and which server configuration to use
jboss.home=../JBoss
jboss.conf=default

# Default configuration: Use JBoss-builtin hsqldb database
# on file system
database=hsqldb
ds.database=default
ds.name=DefaultDS
jdbc.jar=${jboss.home}/server/${jboss.conf}/lib/hsqldb.jar
jdbc.driver=org.hsqldb.jdbcDriver
jdbc.host=localhost
jdbc.port=1701
jdbc.url=jdbc.url=jdbc:hsqldb:${jboss.home}/server/$
{jboss.conf}/data/hypersonic/localDB
jdbc.userid=sa
jdbc.password=
```

Listing 1: A sample `"build.properties"`.

If you've chosen the standard distribution you may leave all values as they are.

Note the following:

- **jboss.home** must point to the root of your JBoss (version 3.2.x) installation.
- **jboss.conf** chooses the JBoss configuration (usually “default”).
- **database** currently selects among *hsqldb*, *mysql*, or *oracle*. Some of the following properties depend on the chosen database.
- **ds.database** is the name of the database your application will use. Hypersonic will physically implement the database by creating files in `${jboss.home}/server/${jboss.conf}/data/hypersonic/`. If you want to drop the database, shut down JBoss and delete those files (`*.data`).
- **ds.name** should provide the name of the data source operating on that database. To register your own data sources with JBoss see chapter 10.
- **jdbc.jar** must point to the JAR file containing the JDBC driver.
- **jdbc.driver** must contain the fully qualified name of the Java class implementing the JDBC driver.
- **jdbc.host** must contain the host on which your database server is running. In case of the Hypersonic database integrated with JBoss this is the host on which JBoss is running. However, by default Hypersonic is configured to be accessed via a file path, not a TCP connection.
- **jdbc.port** must contain the port your database server is listening on. The default Hypersonic database integrated with JBoss listens on port 1701, if you change default configuration.
- **jdbc.url** must contain the connection URL for JDBC. In the above example it is constructed from other properties, so probably there’s no need to change it explicitly.
- **jdbc.userid** contains the database user. For Hypersonic it seems always to be *sa*.
- **jdc.password** contains the database user’s password. For Hypersonic no password is needed.

Next you will habe to copy the file `"setpath_example.bat"` to `"setpath.bat"` (or `"setpath_example.sh"` to `"setpath.sh"`, depending on your operating system). It contains the paths to the <b-frame/> and Ant installation directories and is used by `"build.bat"`, or `"build.sh"`, respectively.
Before continuing you should make sure that your database (e.g. the Hypersonic database integrated into JBoss) as well as your application server (the current <b-frame/> version only supports JBoss) are up and running.

Now you can start developing your first <b-frame/> application.

4 Step One: Just Students

This section describes how to create a simple *task* just to list and maintain *Students*. The task will not allow for assigning *Students* to *Courses*. On our way to that goal we will see

- how to set up a project,
- how to create the database,
- how to create the application, and
- how to cleanup the project.

4.1 The Project's Setup

First you have to think about a name for your application, it should be unique among the names of all applications, e.g. *university1*. For an application's setup you have to create a project folder inside b-frame's `app` sub folder with the name you've just invented for your application. Inside the project's folder you will need to create a folder `xml` which in turn should contain two sub folders, one named `db` and the other one named `app`. Just do the following:

1. Create a folder *<b-frame-examples>*`/university1`
2. Create a folder *<b-frame-examples>*`/university1/xml`
3. Create a folder *<b-frame-examples>*`/university1/xml/db`
4. Create a folder *<b-frame-examples>*`/university1/xml/app`

The sub folder `db` is for holding the description of the application's database, whereas the sub folder `app` is for holding the description of the application's tasks. The following section describes how to create the database.

4.1.1 The Database

In <b-frame/> the database is defined by means of an XML-file called the database descriptor. It resides inside the *b-frame-examples*`/university1/xml/db` folder and currently must be named `"tables.xml"`. The following listing shows the database descriptor for the simple *Student Task*.

```
<?xml version="1.0"?>

<bf-db:tables xmlns:bf-db="http://www.b-frame.org/b-
frame/db">

<bf-db:table name="student">
  <bf-db:primary-key name="student_pk">
    <bf-db:attribute name="id" type="integer"/>
  </bf-db:primary-key>
  <bf-db:attribute name="name" type="string" length="32"/>
</bf-db:table>

</bf-db:tables>
```

Listing 2: The database descriptor `"tables.xml"`.

Up to now the only table defined by the database descriptor is the *Student* table. The *Student* table consists of two attributes, *id* and name where the table's primary key is *id*. Steps discussed below will add more tables to the database descriptor.

Note the following:

- All tags are taken from the *name space* **http://www.b-frame.org/b-frame/db**, which is reserved for the <b-frame/> database descriptor. Typically the name space is abbreviated by the **prefix bf-db**.
- **<bf-db:tables>** is the root tag of the database descriptor. It allows for several nested `<bf-db:table>` tags.
- **<bf-db:table>** is a tag to describe a single table of a (relational) database. A `<bf-db:table>` tag
  - must provide the attribute **name** whose value matches the table name of the database in use,
  - must contain one `<bf-db:primary-key>` tag,
  - may contain one or more `<bf-db:foreign-key>` tags, and
  - may contain one or more `<bf-db:attribute>` tags.
- **<bf-db:primary-key>** is a tag for defining a table's primary key. It always has to be nested inside a `<bf-db:table>` tag. A `<bf-db:primary-key>` tag
  - must provide the attribute **name** with a value usable as the name of a database constraint (e.g., the table's name with the suffix "_pk"),
  - must contain at least one (the current <b-frame/> version requires exactly one) `<bf-db:attribute>` tag, and
  - may contain several `<bf-db:foreign-key>` tags.
- **<bf-db:foreign-key>** is a tag for defining a table's foreign key. It may be nested inside a `<bf-db:table>` tag or a `<bf-db:primary-key>` tag. A `<bf-db:foreign-key>` tag
  - must provide the attribute **name** with a value usable as the name of a database constraint (often, but not all times, the table's name and the referenced table's name concatenated by the infix "_ref_" is a good idea),
  - must contain one or more `<bf-db:attribute>` tags.
- **<bf-db:attribute>** is a tag for defining a table's attribute. It may be nested inside a `<bf-db:table>` tag, a `<bf-db:primary-table>` tag, or a `<bf-db:foreign-key>` tag. The `<bf-db:attribute>` doesn't allow for nested tags.

One additional note to the above: All potential global database identifiers, i.e., table names as well as names of primary and foreign keys, should be unique.

Before continuing with the next section you should create the database descriptor as described above and create the database schema. This is done by executing the aforementioned `build.bat/build.sh` command (in the following we will only refer to `build.bat`, for Linux/Unix the commands work accordingly). These scripts implicitly will call Ant targets, thus the command syntax is the same as with pure Ant.

To create the database schema you can use the `cre-schema` target. Note that you will have provide your application's name, i.e., *university1,* as an Ant property (altern-

atively you can override the default in `build.properties`). Assuming your working directory is <b-frame/>'s example installation, issue the following command:

```
D:\b-frame-examples>build -Dapp.name=university1 cre-schema
Buildfile: build.xml
...
      [sql] 2 of 2 SQL statements executed successfully
```

The *Student* table should now exist in the database. The next section describes how to create a task to maintain this table.

4.1.2 The Task

A <b-frame/> application is defined by means of an XML-file called the application descriptor. It resides inside the *b-frame-examples*/university1/xml/app folder and should have the same name as the application, i.e., "university1.xml". The following listing shows the application descriptor for the simple *Student Application*.

```
<bf-app:application
    xmlns:bf-app="http://www.b-frame.org/b-frame/app"
    name="university1">

     <bf-app:task table="student"/>

</bf-app:application>
```

Listing 3: The application descriptor "university1.xml".

<b-frame/> introduces the notion of a task, which is something some people might rather call a use case. A <b-frame/> task may be composed of several subtasks. Beside the information how to connect to a database the above application descriptor contains only one task, the *Student* task. Steps discussed below will add more tasks.

Note the following:

- All tags are taken from the name space **http://www.b-frame.org/b-frame/app**, which is reserved for the <b-frame/> application descriptor. Typically the name space is abbreviated by the **prefix bf-app**.
- **<bf-app:application>** is the root tag of the application descriptor. The `<bf-app:application>` tag
  - must contain at least one `<bf-app:task>` tag.
- **<bf-app:task>** is the most important tag of an application descriptor, it defines the application's structure. The `<bf-app:task>` tag allows for nesting several other `<bf-app:task>` tags, i.e., <b-frame/> tasks form a recursive data structure which is one of the most powerful concepts behind the scenes.

Before continuing you should create the application descriptor as described above, compile and deploy the application. Again, this is done by executing the build script parameterized with the application's name.

```
D:\b-frame-examples>build -Dapp.name=university1 deploy
Buildfile: build.xml
...
compile:
    [javac] Compiling 51 source files to ...

archive:
      [jar] Building jar: ...
      [war] Building war: ...
      [ear] Building ear: ...

deploy:
     [copy] Copying 1 file to ...
```

The next section describes how to test the *Student* task.

4.2 Testing the Task

To test the application you should provide some test data. Do the following:

- Create a folder `b-frame-examples/university1/sql`.
- Assuming you're database is Hypersonic, create a folder `b-frame-examples/university1/sql/hsqldb`.
- Inside the folder `b-frame-examples/university1/sql/hsqldb` create a file `"test_data.sql"` with a content like the following.

```
-- clean up
delete from student;

-- construct data
insert into student (id, name) values (1, 'Madonna');
insert into student (id, name) values (2, 'Bono');
insert into student (id, name) values (3, 'Mick Jagger');
```

Listing 4: Some sample test data `"test_data.sql"`.

To populate the database issue the following command:

```
D:\b-frame-examples>build -Dapp.name=university1 test-data
Buildfile: build.xml

test-data:
      [sql] Executing file: ...
      [sql] 4 of 4 SQL statements executed successfully
```

For testing the *Student Task* point your browser to http://localhost:8080/-university1/student. The URL is composed by the application's name, i.e., *university1*, and the task's name, i.e., *student*. After clicking the "Submit" button and selecting on Id 1 the following should appear:

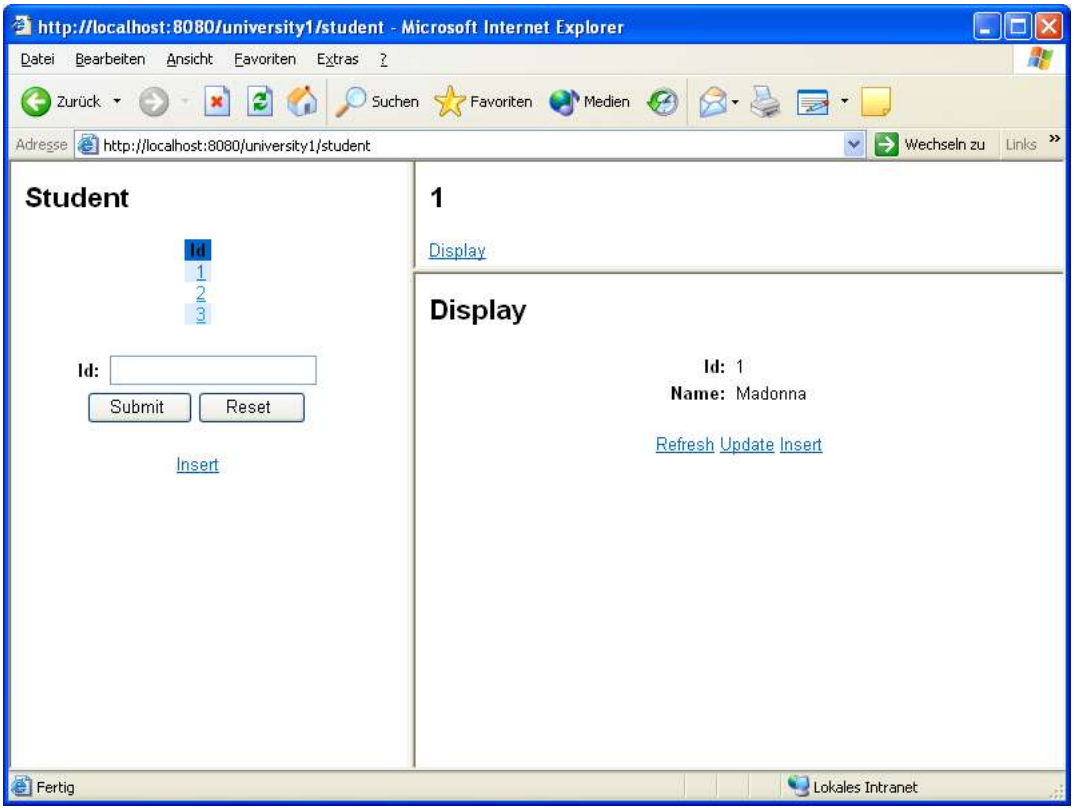


Figure 2: A first version of the *Student Task*.

4.3 Cleaning Up the Project

Dropping the database schema is a common task during development with <b-frame/>. To achieve this issue the following command:

```
D:\b-frame-examples>build -Dapp.name=university1 drop-schema
```

If this seems to fail, i.e., with a message like "`2 of 9 SQL statements executed successfully`" try to run it again until all statements have been executed (This may happen when foreign key constraints are involved).

Important Note: Removing an application's tables from the database is a vital operation, you should do it each time before you add or remove a table to or from the database descriptor. The reason for that is obvious, only the unmodified database descriptor reflects the current database schema. If you change the database descriptor without first dropping the tables, the descriptor is not in sync with the database any longer. It is very likely that you will be faced with a lot of JDBC errors the next time you try to drop or create the schema. If you run into this (and probably this will happen some time) the way out is to drop all tables by hand using a database tool.
In case of the Hypersonic database integrated with JBoss do the following: Shut down JBoss and edit the file `${jboss.home}/server/${jboss.conf}/data/hypersonic/localDB.script`. Remove all `CREATE TABLE` and `INSERT` statements that belong to your application.

Sometimes you may find it useful to clean up your project too. This typically occurs when you want to trigger a complete rebuild. In that case, issue the following command:

```
D:\b-frame-examples>build -Dapp.name=university1 clean
Buildfile: build.xml

undeploy:
   [delete] Deleting:
C:\jboss\server\default\deploy\university1.ear

clean:
   [delete] Deleting directory D:\b-
frame\app\university1\build
```

The next chapter describes how to complete the *Student* and *Institute Tasks*.

Note: Before continuing you should drop the database schema as described above, otherwise you probably will run into trouble as explained above.

5 Step Two: Just Institutes

This step introduces the *Institute Task* which is very similar to the *Student Task*. In order to avoid disturbances between the intermediate steps, each step will create a its own project. Thus do the following:

- Copy the complete folder *b-frame-examples*/university1 to *b-frame-examples*/university2
- Rename the application descriptor *b-frame-examples*/university2/-xml/app/university1.xml to *b-frame-examples*/university2/-xml/app/university2.xml.
- For the <bf-app:application> tag on top of the application descriptor *b-frame-examples*/university2/xml/app/university2.xml change the value of the attribute *name* from *university1* to *universiy2*.

Now you're ready to add the *Institute* table to the database descriptor.

```
<?xml version="1.0"?>

<bf-db:tables xmlns:bf-db="http://www.b-frame.org/b-
frame/db">

<bf-db:table name="student">
  <bf-db:primary-key name="student_pk">
    <bf-db:attribute name="id" type="integer"/>
  </bf-db:primary-key>
  <bf-db:attribute name="name" type="string" length="32"/>
</bf-db:table>

<bf-db:table name="institute">
  <bf-db:primary-key name="institute_pk">
```

```
    <bf-db:attribute name="name" type="string"
length="32"/>
  </bf-db:primary-key>
</bf-db:table>

</bf-db:tables>
```

Listing 5: Database descriptor "tables.xml", version 2

This being done you probably remember from the previous step that it is now time to create the database schema (note that you have to provide the application name *university2*):

```
D:\b-frame-examples>build -Dapp.name=university2 cre-schema
Buildfile: build.xml
...

cre-schema:
      [sql] Executing file: ...
      [sql] 4 of 4 SQL statements executed successfully
```

Next you have to introduce the *Institute Task* to your application descriptor "university2.xml", which now should look like this:

```
<?xml version="1.0"?>

<bf-app:application
    xmlns:bf-app="http://www.b-frame.org/b-frame/app"
    name="university2">

     <bf-app:task table="student"/>
     <bf-app:task table="institute"/>

</bf-app:application>
```

Listing 6: The new application descriptor "university2.xml"

Finally you're ready to deploy the application:

```
D:\b-frame-examples>build -Dapp.name=university2 deploy
Buildfile: build.xml
...

compile:
    [javac] Compiling 93 source files to ...

archive:
      [jar] Building jar: ...
      [war] Building war: ...
      [ear] Building ear: ...

deploy:
     [copy] Copying 1 file to ...
```

Probably you want to test the new task. To have some data you should modify the file "b-frame-examples/university2/sql/hsqldb/test_data.sql":

- Add a statement for cleaning up the *Institute* table on top of the file.
- Add some statements for population the *Institute* table at the bottom of the file.

Remember the general rule that cleaning up should always done in the opposite order than constructing (if you build a tower, you start at the bottom, if you remove it, you start at the top). If you think about the order by which tables are populated or cleaned up, respectively, bear in mind, that in general (not at this time) there are dependencies introduced by foreign keys: A referenced table should be populated before the referencing table.

The following is an example:

```
-- clean up
delete from institute;
delete from student;

-- construct student data
insert into student (id, name) values (1, 'Madonna');
insert into student (id, name) values (2, 'Bono');
insert into student (id, name) values (3, 'Mick Jagger');

-- construct institute data
insert into institute (name) values ('Computer Science');
insert into institute (name) values ('Mathematics');
insert into institute (name) values ('Physics');
```

Listing 7: Some sample test data "test_data.sql".

Populate the database by issuing the following command:

```
D:\b-frame-examples>build -Dapp.name=university2 test-data
Buildfile: build.xml

test-data:
      [sql] Executing file: ...
      [sql] 8 of 8 SQL statements executed successfully
```

To verify that everything is fine, point your browser to http://localhost:8080/-university2/institute, click the “Submit” button, select *Computer Science* and you should see something like this:

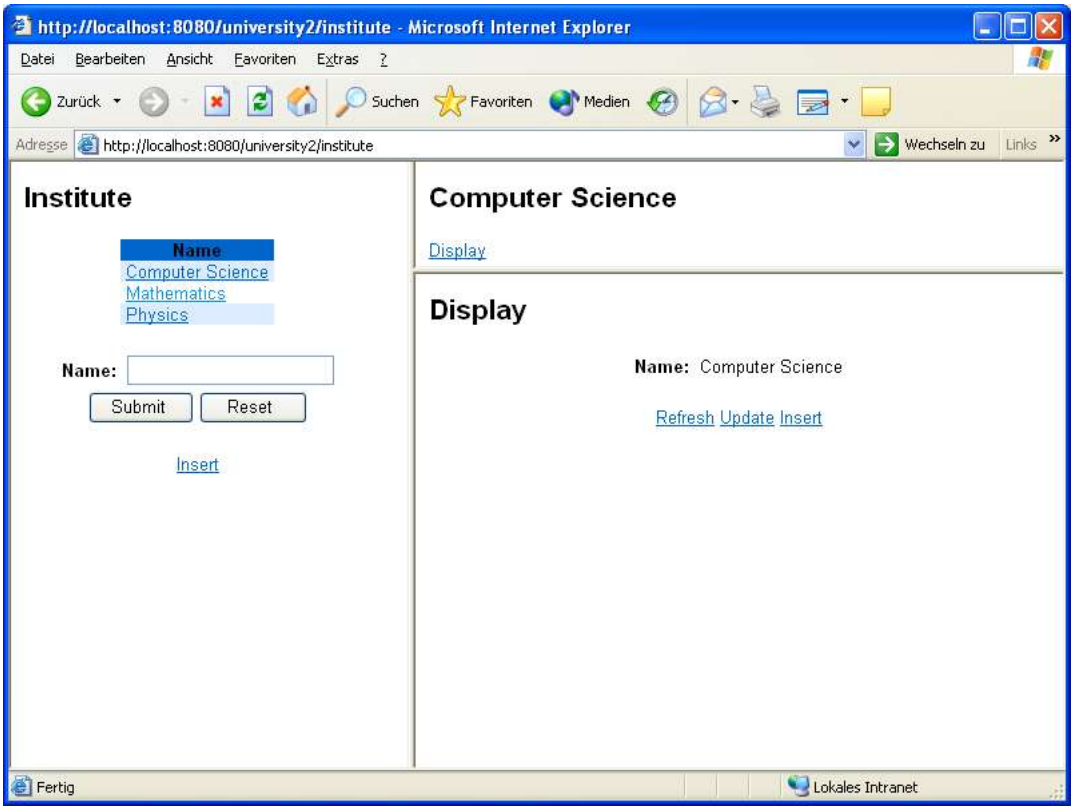


Figure 3 A first version of the *Institute Task*.

The next section explains how to add a nested task, the *Professor Task*, to the *Institute Task*. But before moving on, you should clean up the project and drop the database schema as described in section 4.3.

6 Step Three: Institutes and Professors

This step will add the *Professor Task* to the *Institute Task*. Before continuing you should duplicate the project *university2* to *university3* accordingly as described in the previous step.

First you have to add the *Professor* table to the database descriptor.

```
<bf-db:table name="professor">
  <bf-db:primary-key name="professor_pk">
    <bf-db:foreign-key name="prof_ref_inst" ref="insti-
tute">
      <bf-db:attribute name="institute_name" label="Insti-
tute Name"
          type="name"/>
    </bf-db:foreign-key>
    <bf-db:attribute name="id" label="Professor ID"
type="integer"/>
  </bf-db:primary-key>
  <bf-db:attribute name="name" label="Professor Name"
      type="string" length="32"/>
  <bf-db:attribute name="salary" label="Salary" type="in-
teger"/>
```

```
</bf-db:table>
```

Listing 8: The *Professor* table.

Note the following:

- The *Professor* table's primary key has a nested foreign key referencing the *Institute* table. The `<bf-db:foreign-key>` tag's attribute *ref* defines which table is referenced.
- A `<bf-db:foreign-key>` tag has to contain exactly the same number of `<bf-db:attribute>` tags as the `<bf-db:primary-key>` tag of the referenced table.
- The `<bf-db:attribute>` tags nested to the `<bf-db:foreign-key>` tag and to the referenced table's `<bf-db:primary-key>` tag, respectively, have to match each other pair-wise. The match is mediated by the `<bf-db:attribute>` tag nested to the `<bf-db:foreign-key>` tag. Its *type* attribute has to point to the *name* attribute of the corresponding `<bf-db:attribute>` tag nested to the referenced table's `<bf-db:primary-key>` tag. In the above example the attribute *institute_name* of the *Professor* table references the attribute *name* of the *Institute* table.
- If not provided explicitly by an <bf-db:attribute> tag's *label* attribute, labels are derived from the *name* attribute. Some of the *Professor* table's attribute define labels explicitly.

Provided you've added the *Professor* table to your database descriptor you may now create the database schema as discussed in previous steps. Next you have to introduce the *Professor Task* to your application descriptor. Because the *Professor Task* is a subtask of the *Institute Task* you should do so by modifying the *Institute Task* in the following way.

```
<bf-app:task table="institute">
  <bf-app:task table="professor"/>
</bf-app:task>
```

Listing 9: The *Professor Task* nested inside the *Institute Task*.

Done that you finally may deploy the application as described in previous steps.

To have some data for testing the *Professor* Task define it as usual by modifying the file `"test_data.sql"` accordingly. Add a command for cleaning up the *Professor* table at the top of that file and at its bottom some statements like the following for creating the data.

```
-- construct professor data
delete from professor;
insert into professor (institute_name, id, name, salary)
  values ('Computer Science', 1, 'Gosling', '6000');
insert into professor (institute_name, id, name, salary)
  values ('Computer Science', 2, 'Stroustrup', '6000');
insert into professor (institute_name, id, name, salary)
  values ('Computer Science', 3, 'Wirth', '6000');
insert into professor (institute_name, id, name, salary)
```

```
  values ('Mathematics', 1, 'Euklid', '6000');
insert into professor (institute_name, id, name, salary)
  values ('Mathematics', 2, 'Gauss', '6000');
insert into professor (institute_name, id, name, salary)
  values ('Physics', 1, 'Newton', '6000');
insert into professor (institute_name, id, name, salary)
  values ('Physics', 2, 'Maxwell', '6000');
insert into professor (institute_name, id, name, salary)
  values ('Physics', 3, 'Hawking', '6000');
```

Listing 10: Some sample *Professor* data.

Once you have

- populated the database as described in previous steps,
- pointed the browser to http://localhost:8080/university3/institute,
- clicked on the "Submit" button,
- selected the *Computer Science* institute,
- selected the *Professor* page,
- and finally selected the *Professor Id* 3,

you should see something like this:

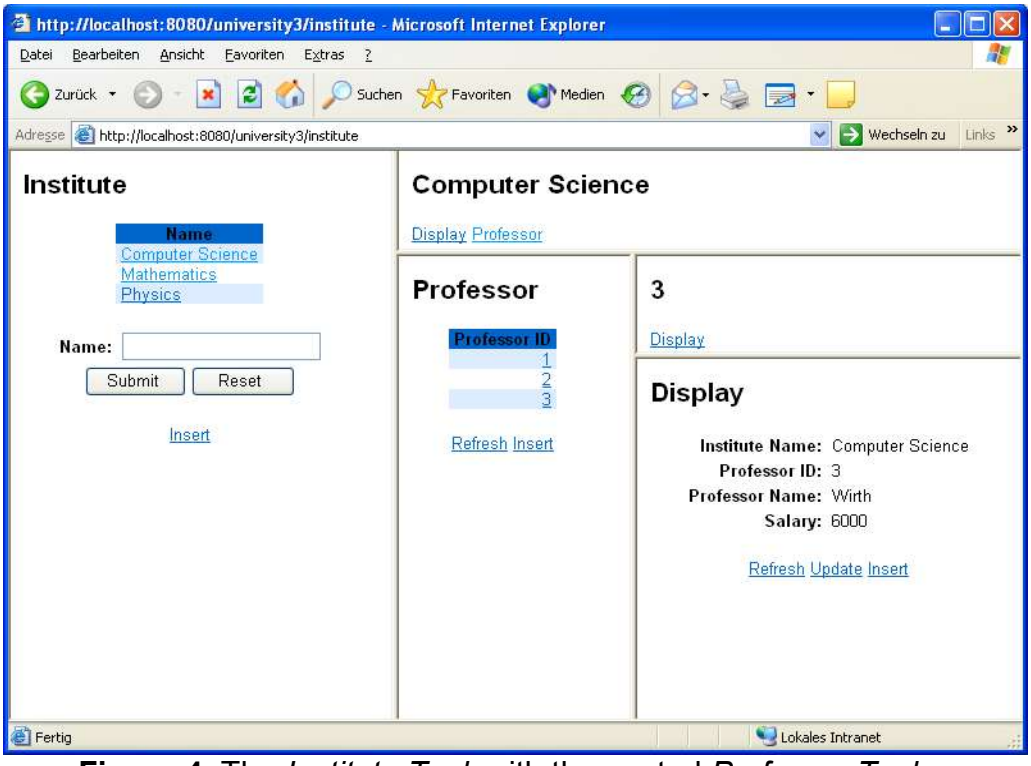


Figure 4: The *Institute Task* with the nested *Professor Task*

Note how the task's recursive structure is preserved by the mask.
The following section adds the *Course Task* to the *Professor Task*. Before continuing don't forget to clean up, especially the database schema.

7 Step Four: Institutes, Professors and Courses

Now its time to add the *Course Task* to the *university* application. Before doing so you should duplicate the project from *university3* to *university4*. The *Course* table to be added looks like this:

```
<bf-db:table name="course">
  <bf-db:primary-key name="course_pk">
    <bf-db:foreign-key name="course_ref_prof" ref="profess-
or">
      <bf-db:attribute name="institute_name" type="insti-
tute_name"/>
      <bf-db:attribute name="professor_id" type="id"/>
    </bf-db:foreign-key>
    <bf-db:attribute name="id" label="Course ID" type="in-
teger"/>
  </bf-db:primary-key>
  <bf-db:attribute name="title" label="Course Title"
      type="string" length="32"/>
</bf-db:table>
```

Listing 11: The *Course* table.

Note that the *Course* table's primary key has a nested foreign key pointing to the *Professor* table. Create the database schema in the usual way.
The application descriptor should be modified as follows to define the *Course Task* nested to the *Professor Task*.

```
<bf-app:task table="institute">
  <bf-app:task table="professor">
    <bf-app:task table="course"/>
  </bf-app:task>
</bf-app:task>
```

Listing 12: The *Course Task* nested to the *Professor Task*.

Now everything is ready to deploy the application in the usual way. For testing the *Course Task* first modify the file `"test_data.sql"` by adding a statement for cleaning up the *Course* table on top of that file and some statements like the following at its bottom.

```
-- construct course data
delete from course;
insert into course (institute_name, professor_id, id,
title)
  values ('Computer Science', 1, 1, 'Java');
insert into course (institute_name, professor_id, id,
title)
  values ('Computer Science', 2, 1, 'C++');
insert into course (institute_name, professor_id, id,
title)
  values ('Computer Science', 3, 1, 'Modula');
insert into course (institute_name, professor_id, id,
title)
```

```
  values ('Computer Science', 3, 2, 'Pascal');
insert into course (institute_name, professor_id, id,
title)
  values ('Computer Science', 3, 3, 'Oberon');
insert into course (institute_name, professor_id, id,
title)
  values ('Mathematics', 1, 1, 'Geometry');
insert into course (institute_name, professor_id, id,
title)
  values ('Mathematics', 2, 1, 'Analysis');
insert into course (institute_name, professor_id, id,
title)
  values ('Mathematics', 2, 2, 'Applied Mathematics');
insert into course (institute_name, professor_id, id,
title)
  values ('Physics', 1, 1, 'Mechanics');
insert into course (institute_name, professor_id, id,
title)
  values ('Physics', 1, 2, 'Gravity');
insert into course (institute_name, professor_id, id,
title)
  values ('Physics', 2, 1, 'Electrodynamics');
insert into course (institute_name, professor_id, id,
title)
  values ('Physics', 3, 1, 'Cosmology');
```

Listing 13: Some sample data to populate the *Course* table.

Provided the database schema is created, the application is deployed and the database is populated, pointing your browser to http://localhost:8080/university4/institute and navigating to the Oberon course held by Professor Wirth should reveal something like this:

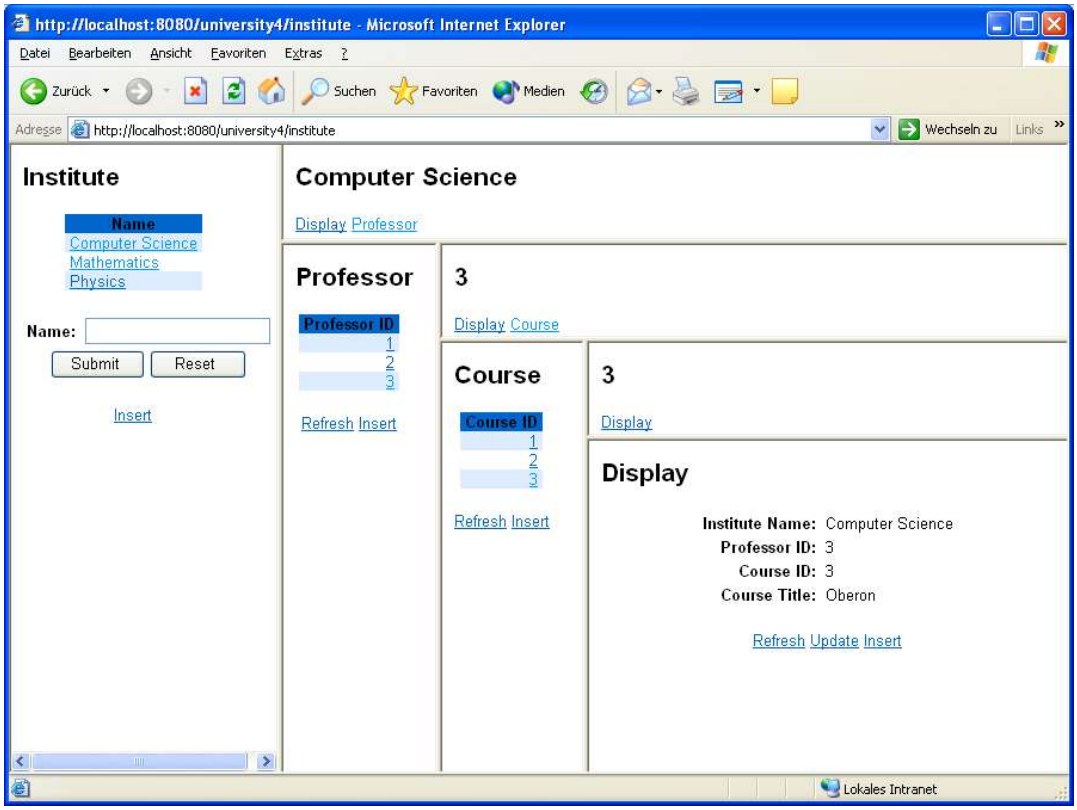


Figure 5: The *Institute Task* with the *Professor* and *Course Tasks* nested

The next step is the final one, it will modify both tasks, the *Student Task* as well as the *Institute Task*, to enable a user assigning *Students* to *Courses*. As usual you should clean up the database schema and the project before continuing.

8 Step Five: Students and Courses

So far we have two tasks both allowing a user to maintain independent hierarchical data structures. This final step will link the two hierarchies by means of the association class *StudentCourseVisit*.

```
<bf-db:table name="visit">
  <bf-db:primary-key name="visit_pk">
    <bf-db:foreign-key name="visit_ref_student"
ref="student">
      <bf-db:attribute name="student_id" type="id"/>
    </bf-db:foreign-key>
    <bf-db:foreign-key name="visit_ref_course"
ref="course">
      <bf-db:attribute name="institute_name"
type="institute_name"/>
      <bf-db:attribute name="professor_id"
type="professor_id"/>
      <bf-db:attribute name="course_id" type="id"/>
    </bf-db:foreign-key>
  </bf-db:primary-key>
</bf-db:table>
```

Listing 14: The *Visit* table.

As usual you should add the *Visit* table to the database descriptor and create the database schema.
From the application's point of view it should be possible,

- if looking at a *Student* to see all *Courses* he or she visits, and
- if looking at a *Course* to see all *Students* visiting it.

To achieve this you have to add a corresponding subtask to both tasks, the *Student* as well as the *Course Task*.

```
<bf-app:task table="student">
  <bf-app:task name="course_list" table="visit"/>
</bf-app:task>

<bf-app:task table="institute">
  <bf-app:task table="professor">
    <bf-app:task table="course">
      <bf-app:task name="student_list" table="visit"/>
    </bf-app:task>
  </bf-app:task>
</bf-app:task>
```

Listing 15: The *Course List* and *Student List Tasks*.

Note that because a task's name has to be unique within application scope, it's no longer possible to make use of <b-frame/>'s implicit assumption that a task's name is the same as the name of the table the task operates on. To allow more than one task operating on the same table, the tasks have to be distinguished by the *name* attribute of the `<bf-app:task>` tag.

Provided you've adapted the application descriptor you may deploy the application in the usual way. To have some sample data, add a command for deleting the Visit table on top and some statements like the following at the bottom of file `"test_data.sql"`.

```
-- construct visit data
delete from visit;
insert into visit (student_id, institute_name,
professor_id, course_id)
  values (1, 'Computer Science', 1, 1);
insert into visit (student_id, institute_name,
professor_id, course_id)
  values (1, 'Mathematics', 2, 1);
insert into visit (student_id, institute_name,
professor_id, course_id)
  values (1, 'Physics', 3, 1);
insert into visit (student_id, institute_name,
professor_id, course_id)
  values (2, 'Computer Science', 2, 1);
```

```
insert into visit (student_id, institute_name,
professor_id, course_id)
  values (2, 'Mathematics', 2, 2);
insert into visit (student_id, institute_name,
professor_id, course_id)
  values (2, 'Physics', 2, 1);
insert into visit (student_id, institute_name,
professor_id, course_id)
  values (3, 'Computer Science', 3, 1);
insert into visit (student_id, institute_name,
professor_id, course_id)
  values (3, 'Mathematics', 2, 1);
insert into visit (student_id, institute_name,
professor_id, course_id)
  values (3, 'Physics', 1, 1);
```

Listing 16: Some sample data to populate the *Course* table.

That being done populate the database as described in previous steps.

You may now see the *Course List Task* as well as *Student List Task* if you navigate to them starting with from *Student Task* or the *Institute Task*, respectively. The following figure shows the Course *List Task* nested to the *Student Task*.

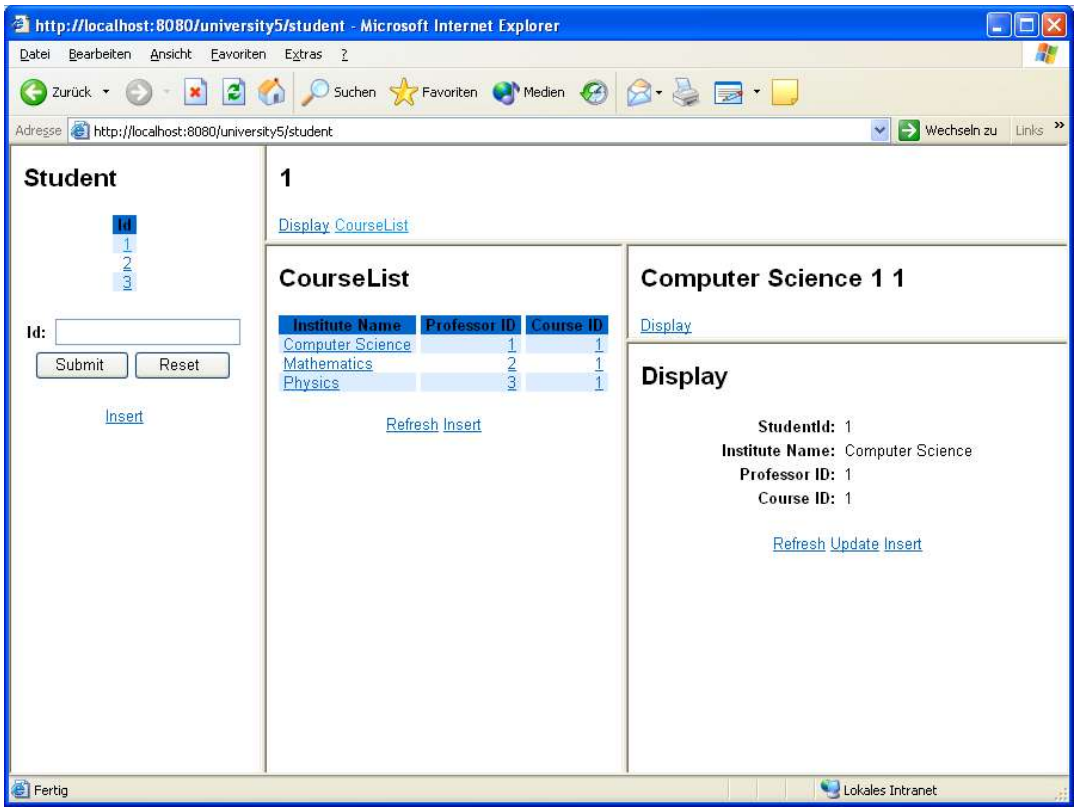


Figure 6: The *Student Task* with the *Course List Task* nested.

As far as the structure is considered, both tasks, the *Student Task* as well as the *Institute Task*, are now complete. Now we will customize the tasks to make them more user friendly.

9 Customizing Tasks

9.1 Customizing a Group

Looking at the *Student Task* it is not very comfortable to see only a list of numbers, who can remember them all? It would be better to also have the names in the list. Fortunately <b-frame/> offers a possibility to customize the list as well as all other field groups.

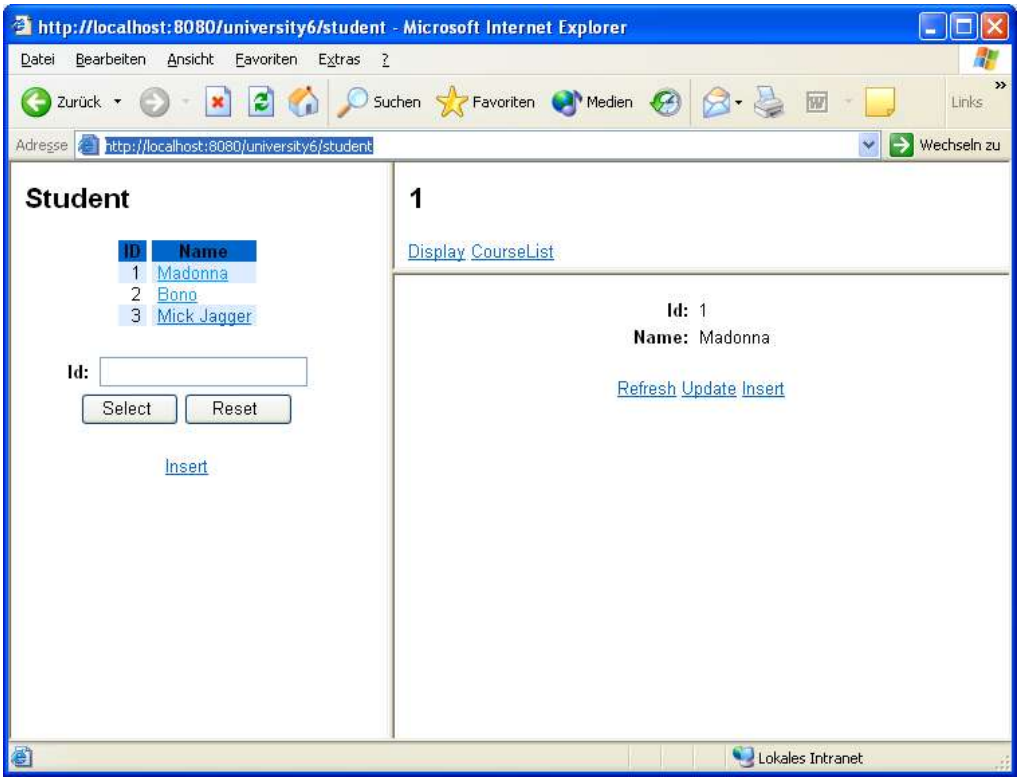
```
<bf-app:task table="student">
  <bf-app:group type="list">
    <bf-app:field name="id" label="ID"/>
    <bf-app:field name="name" label="Name" link="true"/>
  </bf-app:group>
  <bf-app:task name="course_list" table="visit"/>
</bf-app:task>
```

Listing 17: The *Student Task* with customized select list.

Note the following:

- One or more `<bf-app:group>` tags may be nested to a `<bf-app:task>` tag for customizing a group of fields.
- The group is identified by the *type* attribute of tag `<bf-app:group>` defining the mask which should be customized. For the *type* attribute the following values are allowed:
  - `"list"`,
  - `"display"`,
  - `"insert"`, and
  - `"update"`.
- A group of a given *type* can occur only one time.
- One or more `<bf-app:field>` tags may be nested to a `<bf:-app:group>` tag. The `<bf-app:field>` tag allows the following attributes:
  - The *name* attribute is mandatory, it has to match an attribute of the table corresponding to the task.
  - The *label* attribute is optional.
  - The *link* attribute is optional. It is only useful for a group of type `"list"`. If set to `"true"`, this field will have a link.

After adapting the application descriptor and deploying the application, it should look like this:



Listing 18: The *Student Task* with customized select list.

The following section describes how to customize a task to have other than primary key fields for user input to the select operation.

9.2 Customizing the Query

By default, the query of a top level task allows user input for the primary key fields. This is a good default solution, but in most cases it will be necessary to define the fields for user input. To enable this the tags `<bf-app:query>` and `<bf-app:infix>` may be used as shown in the following listing.

```
<bf-app:task table="student">
  <bf-app:query>
    <bf-app:infix name="and">
      <bf-app:field name="id" label="ID"/>
      <bf-app:field name="name" label="Name"/>
    </bf-app:infix>
  </bf-app:query>
  <bf-app:group type="list">
    <bf-app:field name="id" label="ID"/>
    <bf-app:field name="name" label="Name" link="true"/>
  </bf-app:group>
  <bf-app:task name="course_list" table="visit"/>
</bf-app:task>
```

Listing 19: The *Student Task* with customized query.

The tables *Course* and *Professor* are joined to the *Visit* table for displaying the course's title and the professor's name. Note that the fields *course.id* and *profess-*

or.id are given an alias because each field of a group needs to have a unique identifier. When thinking about identifiers, bear in mind that each group is extended by <b-frame/> to include all primary key attributes (but only the defined attributes occur on a mask).

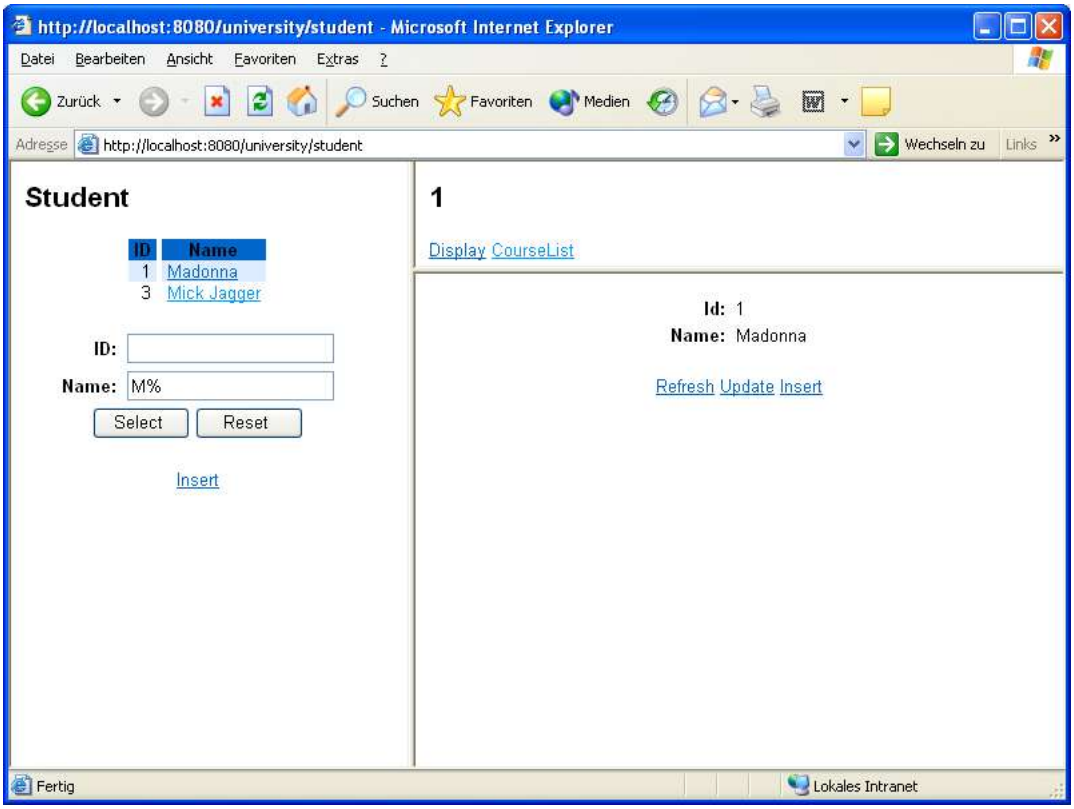


Figure 7: The *Student Task* with customized query.

Note that the current version does not allow to have fields from joined tables as user input for the select operation, as described in the following section for groups of type *list* and *display*.

9.3 Joining Tables

By default, the fields of a group are the attributes of the corresponding table. However, sometimes it is necessary to display information from related tables as well. Currently <b-frame/> allows joining tables using foreign keys in list and display groups as illustrated by the following listing.

```
<bf-app:task table="student">
  <bf-app:query>
    <bf-app:infix name="and">
      <bf-app:field name="id" label="ID"/>
      <bf-app:field name="name" label="Name"/>
    </bf-app:infix>
  </bf-app:query>
  <bf-app:group type="list">
    <bf-app:field name="id" label="ID"/>
    <bf-app:field name="name" label="Name" link="true"/>
  </bf-app:group>
```

```
  <bf-app:task name="course_list" table="visit">
    <bf-app:group type="list">
      <bf-app:field name="institute_name"
label="Institute"/>
      <bf-app:field table="professor" name="id"
          alias="professor_test_id" label=""/>
      <bf-app:field table="professor" name="name"
label="Professor"/>
      <bf-app:field table="course" name="id"
          alias="course_test_id" label=""/>
      <bf-app:field table="course" name="title"
          label="Course" link="true"/>
      <bf-app:join table="course" on="visit_ref_course">
        <bf-app:join table="professor"
on="course_ref_prof"/>
      </bf-app:join>
    </bf-app:group>
  </bf-app:task>
</bf-app:task>
```

Listing 20: The *Course List Task* with fields joined to the list.

The example joins fields from the *Course* and *Professor* tables to the *Visit* table using the foreign keys *visit_ref_course* and *course_ref_prof*, respectively. The resulting mask looks as follows.

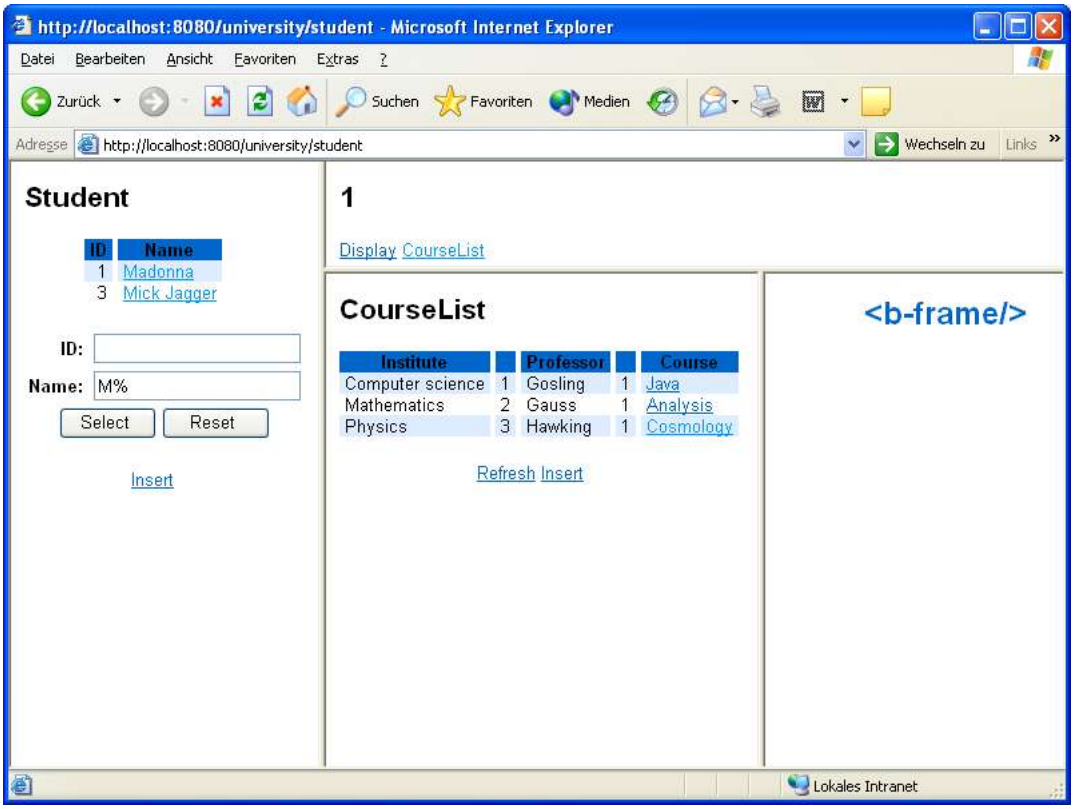


Figure 8: The *Course List Task* with fields joined to the list.

Note that the current <b-frame/> version allows joining only for groups of type *list* and *display*. The following chapter discusses how to use databases other than Hypersonic.

10 Database configuration

All information with respect to an application's database and its data source has to be provided in the properties file `"build.properties"`. What kind of information is necessary depends on the database and is explained in detail below. For creating the database and the data source the following `Ant` targets are available:

- **cre-user** creates a database (MySQL) or a database user (Oracle). Editing SQL scripts beforehand is not needed any longer. For Hypersonic the target is meaningless.
- **drop-user** drops a database (MySQL) or a database user (Oracle). Editing SQL scripts beforehand is not needed any longer. For Hypersonic the target is meaningless.

However, use these targets with care. It is probably better to use the proper administration tool provided by your database (e.g., MysqlCC for MySQL).

10.1 Installing a data source in JBoss

To install a new data source choose one of the templates in the `${jboss.home}/docs/examples/jca` directory and copy it to `${jboss.home}/server/${jboss.conf}/deploy` directory. For instance, copy `mysql-ds.xml` and edit it to reflect your local settings:

```
<datasources>
  <local-tx-datasource>
    <jndi-name>UniversityDS</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/university</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name></user-name>
    <password></password>
  </local-tx-datasource>
</datasources>
```

Listing 21: MySQL data source descriptor.

Note that the `<jndi-name>` tag contains the symbolic name for the data source, and that the `<connection-url>` tag ends with the database name.
Before you start JBoss however, you'll have to make the JDBC driver known to it. This can be accomplished by simply copying the JAR file containing the driver to `${jboss.home}/server/${jboss.conf}/lib`.

The next sections discuss which properties should be provided in the `"build.properties"` for the MySQL and the Oracle databases. This chapter's final section describes how to deal with the appropriate Ant commands.

10.2 MySQL

This section describes how to use MySQL instead of the Hypersonic database. Assuming you're application shall use a database named *university* and a data source named *UniversityDS*, your properties file "build.properties" should look like this:

```
# Where JBoss is located and which server configuration to
use
jboss.home=../JBoss
jboss.conf=default

# Alternate configuration: MySQL
database=mysql
ds.database=university
ds.name=UniversityDS
jdbc.jar=C:/mysql/mysql-jdbc.jar
jdbc.driver=com.mysql.jdbc.Driver
jdbc.host=localhost
jdbc.url=jdbc:mysql://${jdbc.host}/${ds.database}
jdbc.userid=
jdbc.password=
jdbc.system-url=jdbc:mysql://${jdbc.host}/university
```

Listing 22: "build.properties" customized for using the MySQL database.

10.3 Oracle

This section describes how to use Oracle instead of the Hypersonic database. Assuming you're application shall use a database named *university* and a data source named *UniversityDS*, your properties file "build.properties" should look like this:

```
# Alternate configuration: Oracle
database=oracle
ds.name=UniversityDS
jdbc.jar=C:/oracle/ora81/jdbc/lib/classes12.zip
jdbc.driver=oracle.jdbc.driver.OracleDriver
jdbc.host=localhost
jdbc.port=1521
jdbc.instance=pb
jdbc.url=jdbc:oracle:thin:@${jdbc.host}:${jdbc.port}:
${jdbc.instance}
jdbc.system-userid=system
jdbc.system-password=manager
jdbc.userid=university
jdbc.password=university
```

Listing 23: "build.properties" customized for the Oracle database.

10.4 Ant Targets

This section gives examples for how to deal with the `Ant` commands for creating or removing a database. The examples are taken from the *university1* application. If you need to create the application's database, e.g. *university*, issue the following command:

```
D:\b-frame-examples>build -Dapp.name=university1 cre-user
Buildfile: build.xml

cre-user:
    [style] Processing ...
    [style] Loading stylesheet ...
      [sql] Executing file: ...
      [sql] 1 of 1 SQL statements executed successfully
```

As you may have guessed, dropping the database is possible in the following way:

```
D:\b-frame-examples>ant -Dapp.name=university1 drop-user
Buildfile: build.xml

drop-user:
      [sql] Executing file: ...
      [sql] 1 of 1 SQL statements executed successfully
```

11 <b-frame/> architecture

The <b-frame/> architecture is influenced by the so called „Business Delegate" design pattern in many places. The client (here: the web server) calls the business logic via facades (a pattern again). The implementation classes behind the abstract interfaces are created by generic "Factories" (you guessed it: a pattern). The developer may choose among several variants, for example whether the data access is performed through direct JDBC statements in the facades or by delegation the access to Entity Beans. You may also decide whether Entity and Session Beans should be accessed via remote (RMI) or local method calls. The former makes sense if web server and application server are supposed to run on different hosts. Otherwise the local view is much more efficient.
The following figure 9 shows the three most import configuration possibilities with <b-frame/>:

1. By default a JDBC-DB is created which is used by the Servlet to directly access the database via JDBC calls. This is simple, but transactional properties have to maintained by this business delegate itself.
2. The second possibility is to use the EJB approach with a Session Bean facade that delegates requests to DAO business delegate ("Data Access Object"). This is more complex but allows to use the transactional services offered by the application server.
3. In the third case the business logic is mapped to Entity Beans (EJB) and thus to an EJB business delegate is used. The access to the Entity and Session Beans my be either a remote or a local call where the latter is more efficient and thus the default.

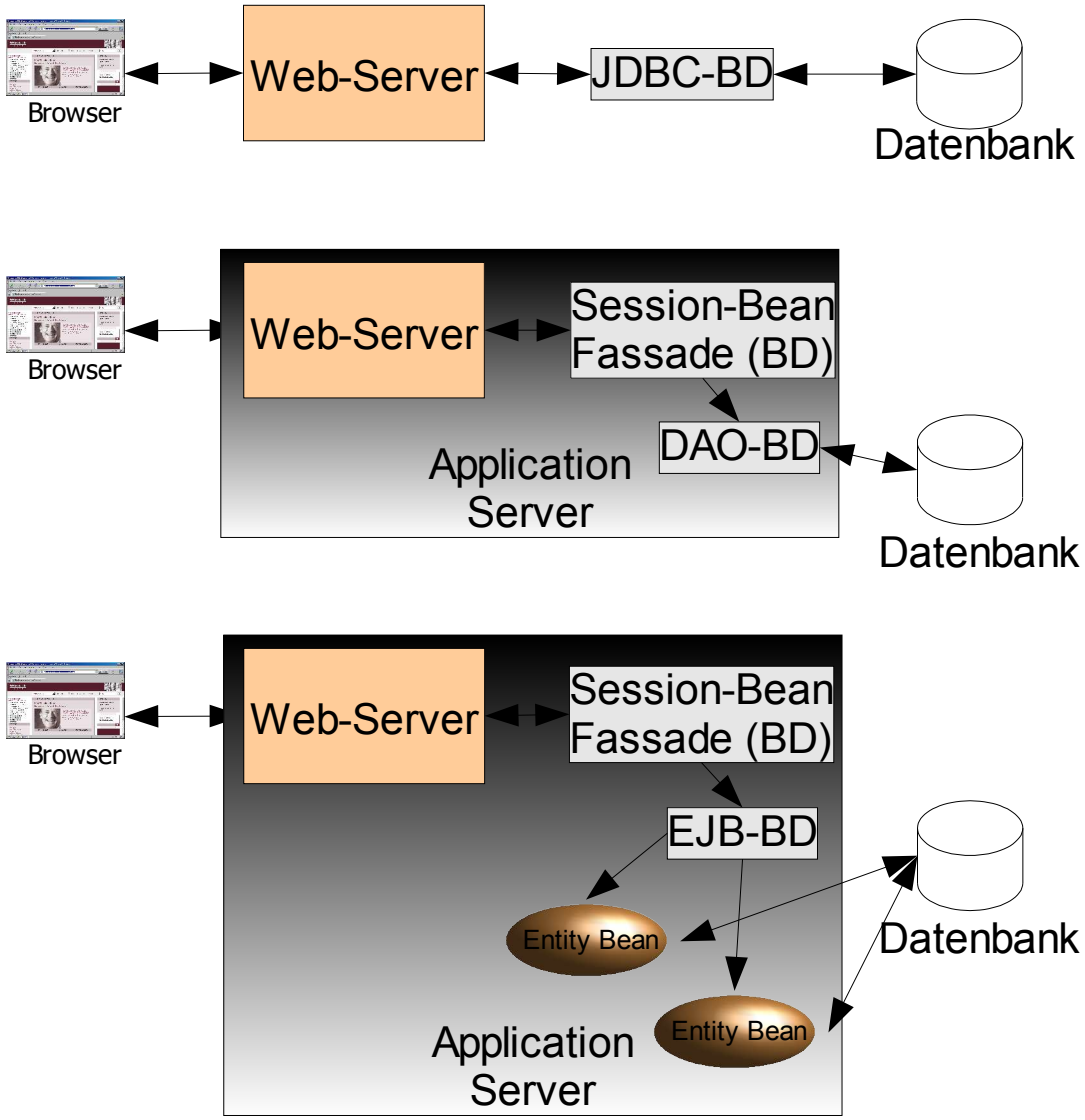


Figure 9: <b-frame/> architecture

The creation of the respective adapter is controlled by factories instantiating the proper class at run time. Which implementation class is to be used is determined during the build process, i.e., by configuration of the application descriptor (see section 11.2).

11.1 How to augment business logic

In order to extend or adapt the default business logic it is possible to override the used factories. To do so you just have to supply a Java class whose fully qualified name is referenced in the application descriptor. Currently, the source file must be located in the project's sub folder `"src"`, e.g., in `"bframe-examples/university/src"`. The class must implement the generated interface. It is also possible to extend the generated factory class and override just those methods which are of interest. A very simple example is just to add a `"println"` statement.

```
package user.foo;
import java.util.List;

import university.student.common.CourseListBD;
import university.student.common.StudentBD;
import university.student.common.StudentDefaultFactory;
import university.student.common.StudentFactory;
import university.student.common.StudentSelectVO;
import university.student.common.StudentUncommittedDsBD;

public class MyStudentFactory extends StudentDefaultFactory
     implements StudentFactory {
  public MyStudentFactory() {
    System.out.println(getClass().getName() +
                       "::MyStudentFactory()");
  }

  public StudentBD createStudentBD() {
    System.out.println(getClass().getName() +
                       "::createStudentBD()");
    return super.createStudentBD();
  }
}
```

Listing 24: Custom factory class for creating student BDs.

11.2 Configuration of the application descriptor

The application descriptor, e.g., "university.xml" may contain a single tag configuring the way the application will be generated.

```
<bf-app:application
     xmlns:bf-app="http://www.b-frame.org/b-frame/app"
     name="university">

     <bf-app:ejb type="local" entity="true"/>
     ...
</bf-app:application>
```

Listing 25: Application descriptor configuration.

The <bf-app:ejb> has two parameters: *type* determines whether the access to the facade should be implemented via a remote or a local call. The second parameter *entity* determines whether to use Entity Beans or Data Access Objects (which is the default).

To override the used factory class, you just have to set an additional parameter *factory* for the already introduced <bf-app:task> tag:

```
<bf-app:task table="student"
     factory="user.foo.MyStudentFactory">
     ...
</bf-app:task>
```

Listing 26: Factory configuration.

12 Summary

Having worked through the all steps of the example it should have become clear that developing an application with <b-frame/> goes like this:

1. Adapt the file "build.properties" according to your needs.
2. Invent a name for your application.
3. Inside <b-frame-examples/>'s create a project folder with the application's name (or create a new directory containing "build.properties" and everything).
4. Inside the project folder's sub folder "xml/db" edit the database descriptor "tables.xml".
5. Create the database schema ("build cre-schema").
6. Inside the project folder's sub folder "xml/app" edit the application descriptor.
7. Deploy the application ("build deploy").
8. Test the application.
9. If the test is OK:
   - You're done
10. Otherwise:
    - Drop the database schema (See section 4.3).
    - Optionally cleanup.
    - GOTO 4

Probably you've discovered some restrictions, weaknesses or limitations of the current <b-frame/> version, among them:

- Attributes may only have a few types: string, integer, number.
- The user input is not verified.
- The client using nested frames is slow.
- Mozilla allows stacking frames into each other only a few times.
- Primary keys may consist of one or more foreign keys (that's OK) but exactly one additional attribute, and that's a real weakness.

These issues will be addressed in the near future and a lot more features will be available. If you have ideas how to improve <b-frame/> or like to participate on the <b-frame/> project, please send mail to *info@b-frame.org*.