NEF Sample Library Application

Tutorial

Palmer Business Park
4550 Forbes Blvd, Suite 320
Lanham, MD 20706

(301) 879-3321

http://www.netspective.com
info@netspective.com

Copyright

Disclaimer and limitation of liability

Customer Support

Customer support is available through e-mail via support@netspective.com

Library Application Tutorial

Table of Contents

1. Objectives

The Library Application is a project meant to get you familiar with more features of NEF by creating a complete library management application.

2. Assumptions

For the purpose of this tutorial, we will be assuming that you installed the following:

- Java Developer's Kit (JDK) 1.2, 1.3 or 1.4 See Section 2.1, “Downloading Java Developer's Kit (JDK)”
- An Application Server (servlet container) supporting the Servlet 2.2 or higher specification See Section 2.2, “Application Server (Servlet Container)”

We assume that you are using the default port `8080` for your web server. If you chose different values for the installation path and the port number, you should substitute the paths and the URLs in our example with your values as needed. This tutorial also assumes your familiarity with XML, SQL, Java, Servlets and JDBC.

2.1. Downloading Java Developer's Kit (JDK)

Since NEFS comprises Java libraries, a fundamental requirement to develop applications with it is a Java SDK (the full SDK is required, the JRE will not be enough). You can obtain Sun’s official Java SDK from its Java web site at http://java.sun.com/j2se/1.4/download.html. This is a link to the Java 1.4 SDK but Java 1.2 and 1.3 will also work.

2.2. Application Server (Servlet Container)

Since Sparx works with standard J2EE application servers, a Servlet container is required if you're going to use Sparx. Both Axiom and Commons work in web-based or non-web-based applications but Sparx is a web application development library so an application server with a Servlet 2.2 or better container is necessary. Sparx-based applications have been tested on the following application servers:

- Apache Tomcat[2] (free)
- Caucho Resin[3] (free for development, commercial license required for deployment)
- BEA WebLogic[4] (commercial)
- IBM WebSphere[5] (commercial)
- ORACLE Application Server[6] (commercial)
- Macromedia JRun[7] (commercial)

Note

We recommend the Caucho Resin[8] application server if you're not familiar with other Servlet containers or if you're new to Java/J2EE application servers. It's an easy to install, easy to use, and fast Servlet container with advanced features that rival other more expensive application servers such as WebLogic and WebSphere. Resin is free for development use but requires a paid license before putting your application into production use. Rest assured though that all Sparx-based applications you write, even on Resin, will remain app-server neutral.

[2] http://jakarta.apache.org/tomcat
[3] http://www.caucho.com
[4] http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/server
[5] http://www.ibm.com/websphere
[6] http://www.oracle.com/appserver/
[7] http://www.macromedia.com/software/jrun/
[8] http://www.caucho.com

3. Setting up the Starter Application

For creation of a new NEF based application, you will need to download the Starter Application. The starter application is just an empty application that contains the minimal set of files required for NEF web applications. It doesn't do anything particularly useful but you can use this sample as your template for the new project.

3.1. Downloading the Starter Application

You can download the NEF Starter Application file from http://www.netspective.com/corp/downloads/frameworks/samples

Important

Depending upon the operating system and browser you're using, the downloaded file may be saved as a zip file (`nefs-starter-empty.war.zip`). In that case, rename the file as `nefs-starter-empty.war` after it is downloaded successfully.

3.2. Setting up the Starter Application (Using Application Server)

3.2.1. Auto Deploying the Starter Application (Using Application Server)

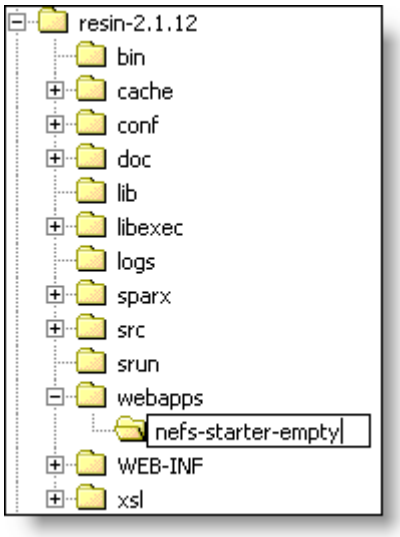
Copy the downloaded `nefs-starter-empty.war` file to the `webapps` folder of your servlet container (application server) and run (or restart) the application server. This creates an Application Directory Structure (see Section 4.1, “The NEF Application Directory Structure”), containing the necessary NEF files and sub-folders, under the `webapps` folder of the application server.

3.2.2. Setting up the Starter Application Manually

Some servlet containers (application servers) may not auto deploy the war files. In such cases you need to manually set up your starter application using the following steps:

1. Create a new folder in the `webapps` folder of your application server. Change this folder's name to `nefs-starter-empty`

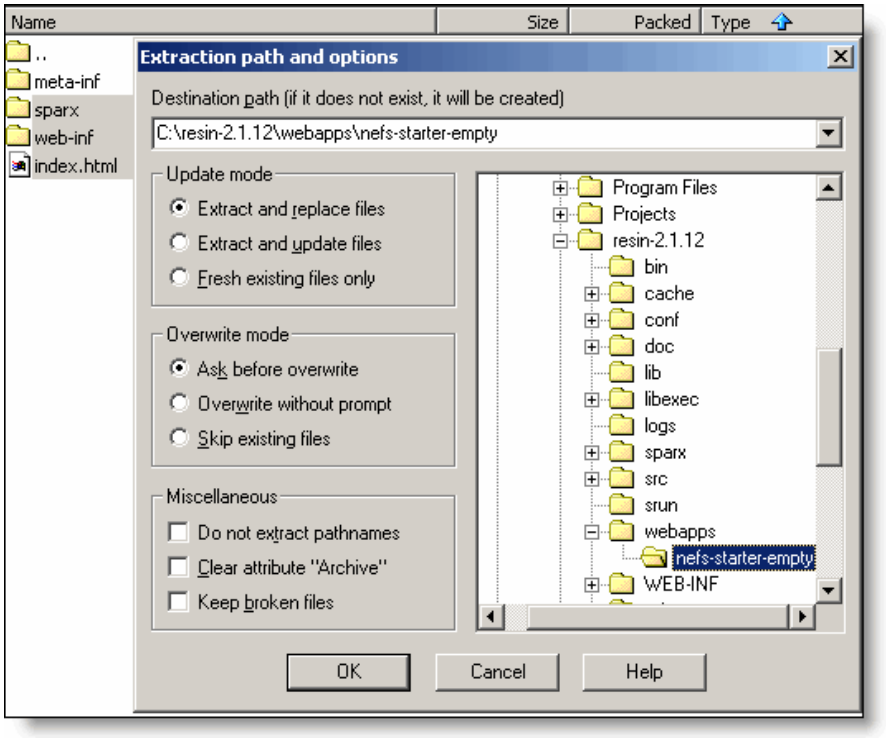
   Figure 1. Creating a New Folder for NEFS Starter Application

2. Extract the contents of `nefs-starter-empty.war` file into this newly created folder. You may use any ZIP file extraction utility, such as WinZip[10] or WinRAR[11], for this purpose.

[10] http://www.winzip.com/downwz.htm
[11] http://www.rarsoft.com/download.htm

Figure 2. Extracting nefs-starter-app.war File



This creates an Application Directory Structure (see Section 4.1, “The NEF Application Directory Structure”), containing the necessary NEF files and sub-folders, within the `nefs-starter-empty` folder.

3.3. Testing the Starter Application in a Browser

Use a web browser to access the root of the starter application using the URL of the form http://*host*:*port*/nefs-starter-empty. If everything worked as it should, you will see the Starter Application Welcome Page.

3.4. Verifying the Console

Use a browser to access the Console of the Starter Application. This will ensure not just the proper configuration of the application but also its proper configuration in relax to Sparx. In a web browser, we can go to the following URL: http://*host*:*port*/nefs-starter-empty/console. If everything is working, you will see the application Console login screen.

Congratulations! You now have an empty application upon which you can build. You can log in to your application's Console. The Console's default User Id is `'console'` and the default Password is `'console'`(each without quotes). Unless otherwise specified, that is the User Id and Password combination you should use if the Console prompts you to login.

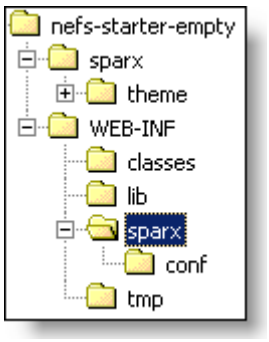
4. Key Concepts

This section outlines some of the important, global concepts that you should be familiar with before embarking on developing your own applications.

4.1. The NEF Application Directory Structure

Every NEF application shares the benefit of a standard directory structure. To see the structure of the empty application (one with only the basic files required for all applications), view the Project | Project Folders tab of the Starter Application using the URL of the form http://*host*:*port*/*appName*/console The appName will be `nefs-starter-empty` in this case.

Figure 3. NEF Standard Project Directory Structure



APP_ROOT — The root directory (in this case `nefs-starter-empty`) contains all the browser-accessible files for the application. This is commonly referred to as the *Document Root* for a website because it is the root directory visible to web browsers. It also contains a private directory, called `WEB-INF`, for the application to store NEF and Java servlet related files (it's called private because none of its contents will ever be served to end users). As already mentioned, all files in the application's root directory are accessible through a web browser. All subdirectories in the application root other than `WEB-INF` will also be directly accessible through a browser. Therefore, if you put an `index.jsp` file in this directory, you should be able to access it using a URL of the form http://*host*:*port*/*appName*/index.jsp.

APP_ROOT/resources — If present this directory tree contains all of the application's shared files that need

	to be served to end users of your applications. Web browser resources that your application needs such as images and scripts are placed here and will be served to your end users by their browser.
APP_ROOT/sparx	This directory tree contains all of the Sparx shared files that need to be served to end users of your applications. Web browser resources that Sparx needs such as style sheets, JavaScript sources, images, and Console files are placed here and will be served to your end users by their browser. You should not modify files in this directory because it does not contain any programmer-modifiable files.
APP_ROOT/resources/sparx	This directory tree (which is not present in the starter application or the diagram above) contains optional Sparx shared files and resources that usually belong in *APP_ROOT*/sparx but are being overridden by your application. For example, if you have your own stylesheets or images that need to replace something in Sparx, they would be placed in this directory. Because the *APP_ROOT*/sparx directory contents should never be modified, the *APP_ROOT*/resources/sparx directory gives you the opportunity to override Sparx resources without worrying about files being overwritten when Sparx is upgrade.
APP_ROOT/WEB-INF	The WEB-INF directory is required by the J2EE Servlet Specification. It contains all files private to the application, meaning none of the files in this directory will be accessible to an end-user's web-browser (except through the Netspective Console which optionally allows secure browsing of source files in WEB-INF). The *APP_ROOT*/WEB-INF/web.xml file configures your application for your J2EE Servlet container and you should refer to your application server's documentation for how to configure the contents of that file.
WEB-INF/classes	This directory, which is a part of the J2EE Servlet Specification, holds all the custom Java source code written for the application. After the application is built, each Java source file in this directory contains a corresponding compiled version in the same location as the source. All Java classes in WEB-INF/classes are automatically included in the classpath of the application. Therefore, if you have declared a dialog (in the project.xml file) to have a custom Java handler for complete or partial dialog processing, the Java source and compiled versions should be located somewhere in this directory structure. Any auxiliary Java classes that you might need should also be placed here. By default, you should place all of your Java classes in the directory WEB-INF/classes/app (or another appropriate subdirectory) because certain application servers will not work with Java classes that are not in a package.
WEB-INF/classes/auto	Although this directory is not found in the starter package, it is automatically created by NEF when it generates classes for use by your application. It is called auto because the classes in there are auto-generated and should not be modified.
WEB-INF/lib	This directory, which is a part of the J2EE Servlet Specification, holds all the Java Archive (JAR) files needed by your application. These include not only JAR files needed for Sparx but also extra JAR files needed by your own Java classes.
WEB-INF/sparx	Sparx uses the WEB-INF/sparx directory to store its project component descriptors. There is usually at least one project.xml file and may contain subdirectories if you wish to split up your application component declarations. The *APP_ROOT/WEB-INF/sparx/project.xml* is the file that drives all of the Sparx functionality in your application.
WEB-INF/sparx/conf	This directory contains sample web.xml configuration files for different application servers like WebLogic, WebSphere, Resin and Tomcat. It also contains Ant build files for compiling your application's classes.

5. Functionality

The NEF Library Application is an application meant to be used for a library of books. It allows you to track the

books within a library and maintain the loan records for these library books. Assets, borrowers and loan information can be added and edited using the various management interfaces provided by the Library Application.

6. Design

6.1. Application Design

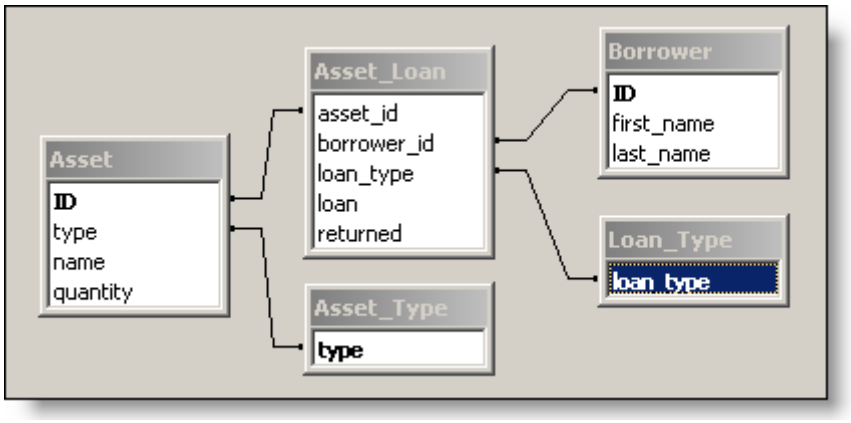
The Library Application is designed around the basic Sparx components. It will use static SQL and associated reports to help you track the assets, borrowers and loan information. The library contains certain assets (books, journals, etc.) which the borrowers can loan for a specified period.

The data storage of choice is the Java-based embedded database that is included in the Sparx Starter Application: HypersonicSQL™[1]. All the application components are included in the `project.xml` file for the Library Application.

6.2. Database Design

The Library Application stores the information about library assets, borrowers and loans. Each borrower can loan an asset belonging to the library. There are different types of assets belonging to the library (e.g. software, periodical). Each asset can be loaned either for a *short term* or *long term.* This is represented by the Loan Type.

Figure 4. Basic E-R Diagram for Library Application Database



The figure shows the entity-relationship diagram for the data we will be using. The database for the Library Application will be designed to store each entity (and its attributes) in a separate table. As with the application design, the database design will become clearer when it is implemented later in this tutorial.

7. Renaming the Starter Application

You can now build your Books Application upon the Starter Application's directory structure. Rename the Starter Application's root folder (`nefs-starter-empty`) to your application's name. This tutorial uses `nefs-sample-library` as the root folder name for the Library Application.

8. The NEF Project File (project.xml)

8.1. Dissecting the project.xml

[1]To learn more about HypersonicSQL, please go to http://hsqldb.sourceforge.net/ ™

Example 1. Project File of Starter Application

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

<project xmlns:xdm="http://www.netspective.org/Framework/Commons/XMLDataModel"> ❶

    <xdm:include resource="com/netspective/commons/conf/commons.xml"/> ❷
    <xdm:include resource="com/netspective/axiom/conf/axiom.xml"/> ❸
    <xdm:include resource="com/netspective/sparx/conf/sparx.xml"/> ❹
    <xdm:include resource="com/netspective/sparx/conf/console.xml"/> ❺

    <!-- Your application tags go here. --> ❻

    <xdm:include file="your/own/file.xml"/> ❼

    <!-- Your other application tags go here. -->
</project>
```

❶ The root tag is called `project` and should use the provided xdm namespace.
❷ Include the Netspective Commons default component declarations. It uses the resource attribute so it will be located by searching the classpath and will usually find the file in the JAR file and directly read it from there.
❸ Include the Netspective Axiom default component declarations and factory registrations. It uses the resource attribute so it will be located by searching the classpath and will usually find the file in the JAR file and directly read it from there.
❹ Include the Netspective Sparx default component declarations and factory registrations. It uses the resource attribute so it will be located by searching the classpath and will usually find the file in the JAR file and directly read it from there.
❺ Include the Netspective Enterprise Console servlet declarations and application components. If you are turning off the Console in your applications you may leave this line out.
❻ This is the location where your component declarations will be done. Unless otherwise specified, all the components are declared right under the `project` tag.
❼ This line demonstrates how you can include your own XML files using the file attribute. In this example, because the file is not absolute it will be treated as relative to the calling file. The `xdm:include` tag may be included anywhere in the file and simply takes items from the included file and places them into the calling file while parsing.

9. Creating the Data Layer

With the empty (Starter) application successfully created and running, it is time to work on the backbone of the Library Application: the database.

9.1. Setting up the Data Source

To set up the Library Application database, you need to have a database connection (data source) pointing towards your database. This is accomplished by using the `connection-provider` tag in the Project File (`project.xml`).

Example 2. Setting up the Data Source for Library Application

```
<project xmlns:xdm="http://www.netspective.org/Framework/Commons/XMLDataModel">
...
<connection-provider

class="com.netspective.axiom.connection.JakartaCommonsDbcpConnectionProvider"> ❶
 <data-source name="jdbc/default"> ❷
  <driver-class>org.hsqldb.jdbcDriver</driver-class> ❸
  <url>vs-expr:jdbc:hsqldb:${servlet-context-path:/WEB-INF/database/instance/db}</url>❹
  <user>sa</user> ❺
  <password></password> ❻
 </data-source>
</connection-provider>
```

❶ A `connection-provider` tag is used to declare the connection to your application's database.

Note

Note that data sources specified under this connection provider will be managed by the Jakarta Commons DBCP. If you wish to use JNDI, you simply have to specify the resource according to the server you're using.

❷ Each `connection-provider` tag may contain one or more `data-source` tags. The `data-source` tag is used to specify the data source for the application. Any data source called '*jdbc/default*' is automatically used as the default JDBC data source. That is why the name of the data source in the above example code is set to `"jdbc/default"`.

Note

If you wish to change the name of the default data source, you may specify it in `project.xml` using the `default-data-source` tag.

❸ The `driver-class` tag is used to provide the driver to be used for the specified database. Since the Library Application uses HSQL database, our sample code specifies the appropriate JDBC driver.

❹ The url is the JDBC URL used to connect to the database. The JDBC driver uses it to point to a specific database on a specific server. The URL has three parts which are separated by a colon `":"`. The first part is always `"jdbc"` and the second part is usually the name of the driver. In the example code, `hsqldb` is the name of the driver that is used to connect to your HypersonicSQL™ database. The third part is the name of the database.

It is important to note the `servlet-context-path` value source. Value sources allow dynamic data to be included in XML without creating a programming language inside XML. In the example code, the `servlet-context-path` value source creates the database named 'db' in `WEB-INF/database/instance` folder.

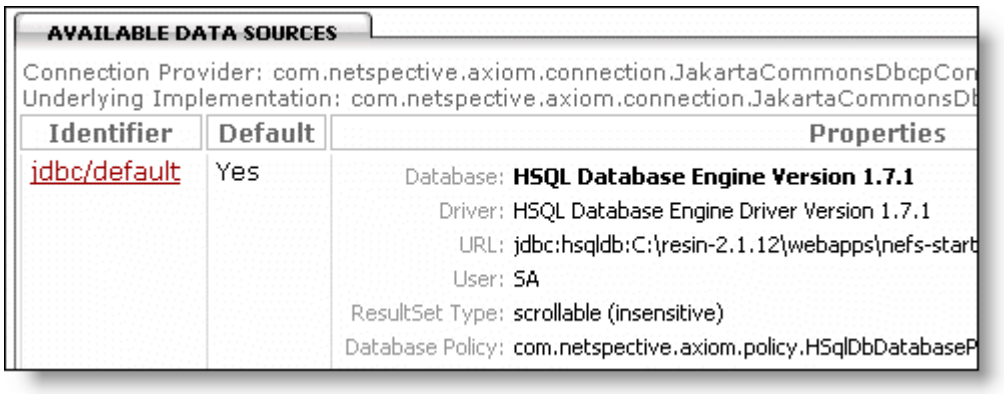
❺ The `user` tag defines a default user to log in to the database. The example code specifies '`sa`' which is the default user for System Administrator.

❻ The `password` tag is used to provide the password for the log in user. The default '`sa`' user has no password.

The above sample code declares a data source for the Library Application database.

9.1.1. Unit Testing the Data Source

You may test the data source by using Data Management | Data Sources section in the Console of your Library Application.



9.2. Creating the Schema

After analyzing the information that needs to be stored in the database and judging from the E-R diagram shown earlier, you can derive the database schema that is necessary for the Library Application. This schema consists of the following five tables:

- *Asset_Type*: used to store information about the different types of assets.

- *Asset*: stores the information about different library assets.
- *Borrower*: stores the information about the borrowers.
- *Loan_Type*: stores the allowed loan types.
- *Asset_Loan*: stores information about a loan.

The Asset_Type and Asset tables are 1:n related by the type (in Asset_Type) and type (in Asset) fields. The Asset and Asset_Loan tables are 1:n related by ID (in Asset) and asset_id (in Asset_Loan) fields. Similarly, Borrower and Asset_Loan tables are 1:n related by ID (in Borrower) and borrower_id (in Asset_Loan) fields. The Loan_Type and Asset_Loan tables are 1:n related by type (in Loan_Type) and loan_type (in Asset_Loan) fields.

Once entered as XML, this schema is available for platform-independent database access from your application.

Following is the code that creates the table types within Library Application schema:

Example 3. Creating table types for Library Application

```
<schema name="db">
  <xdm:include resource="com/netspective/axiom/conf/schema.xml"/>

   <table-type name="Entity"> ❶
    <column name="${owner.name.toLowerCase()} id" type="auto-inc" primary-key="yes" ❷
                                        descr="Unique identifier for ${owner.name}">
     <presentation>
      <field name="${column.name}" type="integer" caption="ID" primary-key-generated="yes"/>❸
     </presentation>
    </column>
   </table-type>

   <table-type name="Person" type="Entity"> ❹
    <column name="first name" type="text" size="64" descr="The person's first name"/> ❺
    <column name="last name" type="text" size="64" descr="The person's last name"/>
   </table-type>
```

❶ Defining a table type named `Entity`.

❷ The `column` element in the `table-type` elements creates actual columns derived from a particular `data-type`. The `column` elements will automatically maintain all type definitions and links to foreign keys. The name attribute represents the column name. Each column is usually named as a singular noun in all lower case with each word inside a name separated by underscores. Since this is a table template, the actual column name of the table that uses this table type will replace *${owner.name.toLowerCase()}* to form the actual column name.

The `type` attribute represents the name of data-type to inherit. All of the attributes and elements from the other data-type will be inherited and any attributes and elements defined in this data-type will override those values. The `primary-key` attribute specifies whether or not this column is a primary key.

❸ Defining the presentation for the column. This defines how the column data will be displayed.

❹ Inheriting a new table type from the `Entity` table type defined in steps 1 through 3.

❺ Specifying columns for the newly defined table type. The `Person` table type will consist of 3 columns - `person_id`, `first_name`, `last_name`.

Following is the code that creates the tables within Library Application schema:

Example 4. Creating tables for Library Application

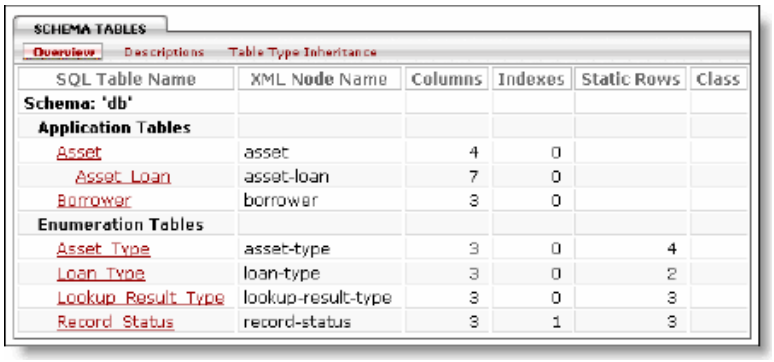
```
<table name="Asset Type" abbrev="atype" type="Enumeration"> ❶
 <enumerations>
  <enum>Other</enum>
  <enum>Software</enum>
  <enum>Periodical</enum>
  <enum>Book</enum>
 </enumerations>
</table>
```

```
<table name="Loan Type" abbrev="ltype" type="Enumeration"> ❷
 <enumerations>
  <enum>Short term</enum>
  <enum>Long term</enum>
 </enumerations>
</table>

<table name="Asset" abbrev="asset" type="Entity,Presentation"> ❸
 <column name="type" lookup-ref="Asset Type" required="yes" descr="The type of asset"/> ❹
 <column name="name" type="text" size="64" required="yes" descr="Name of the asset"/>
 <column name="quantity" type="integer" required="yes"
                         descr="Count of number of assets available to loan"/>
</table>
<table name="Borrower" type="Person,Presentation"> ❺
</table>


<table name="Asset Loan" abbrev="asloan" type="Entity,Presentation">
 <column name="asset id" parent-ref="Asset.asset id" ❻
         descr="The asset that was borrowed (a loan is owned by the Asset so
                it's a parent reference)">
   <presentation>
     <field name="${column.name}" type="select" caption="Asset" ❼
            choices="query:library.asset-names-for-select-field-choices"/>
   </presentation>
 </column>

 <column name="borrower id" lookup-ref="Borrower.borrower id" descr="The person ❽
                                                    that borrowed the asset">
   <presentation>
     <field name="${column.name}" type="select" caption="Borrower" ❾
            choices="query:library.borrower-names-for-select-field-choices"/>
   </presentation>
 </column>



 <column name="loan type" lookup-ref="Loan Type" descr="The type of loan"/> ❿
 <column name="loan" type="duration" required="yes" descr="The duration of the loan"/>
 <column name="returned" type="boolean" descr="Whether the asset has been returned or not"/>
</table>
```

❶ Defining an enumeration containing available asset types.
❷ Defining an enumeration containing available loan types.
❸ Defining `Asset` table using the `Entity` table type defined previously.
❹ Adding more columns to the `Asset` table. The `lookup-ref` attribute specifies a general foreign key relationship from this column which references the foreign field `type` in table `Asset_Type`. This creates a 1:N relationship between the `Asset_Type` and `Asset` tables.
❺ Defining `Borrower` table using the `Person` table type defined previously. No columns are being added.
❻ Defining the field which represents the asset to which the loan belongs. The `parent-ref` attribute specifies a parent/child foreign key relationship which indicates that `Asset` table is the parent of the `asset_id` column (creates a 1:N relationship between `Asset` table and the `asset_id` column).
❼ Defines presentation for the `asset_id` field. The available assets will be displayed as a select option. The `choices` attribute is used to fill the select with available assets. This definition uses the query `asset-names-for-select-field-choices` to obtain the assets.
❽ Defining the field which represents the borrower taking the loan. The `lookup-ref` attribute specifies a general foreign key relationship from this column which references the foreign field `borrower_id` in table `Borrower`. This creates a 1:N relationship between the `Borrower` and `Asset_Loan` tables.
❾ Defines presentation for the `borrower_id` field. The available borrowers will be displayed as a select option. The `choices` attribute is used to fill the select with available borrowers. This definition uses the query `borrower-names-for-select-field-choices` to obtain the borrower names.
❿ Defining the field which represents the loan type. The `lookup-ref` attribute specifies a general foreign key relationship from this column which references the foreign field `type`in table `Loan_Type`. This creates a 1:N relationship between the `Loan_Type` and `Asset_Loan` tables.

9.2.1. Unit Testing the Schema

You may view the newly defined schema by using Data Management | Schemas section in the Console of your Library Application.



SQL Table Name	XML Node Name	Columns	Indexes	Static Rows	Class
Schema: 'db'					
Application Tables					
Asset	asset	4	0		
Asset_Loan	asset-loan	7	0		
Borrower	borrower	3	0		
Enumeration Tables					
Asset_Type	asset-type	3	0	4	
Loan_Type	loan-type	3	0	2	
Lookup_Result_Type	lookup-result-type	3	0	3	
Record_Status	record-status	3	1	3	

There is a list of all the tables contained in the schema. It should list a total of 5 tables, of which the most important to you are the ones you explicitly created: `Asset`, `Borrower` and `Asset_Loan`. You can view the details for the schema tables from this section of the Console.

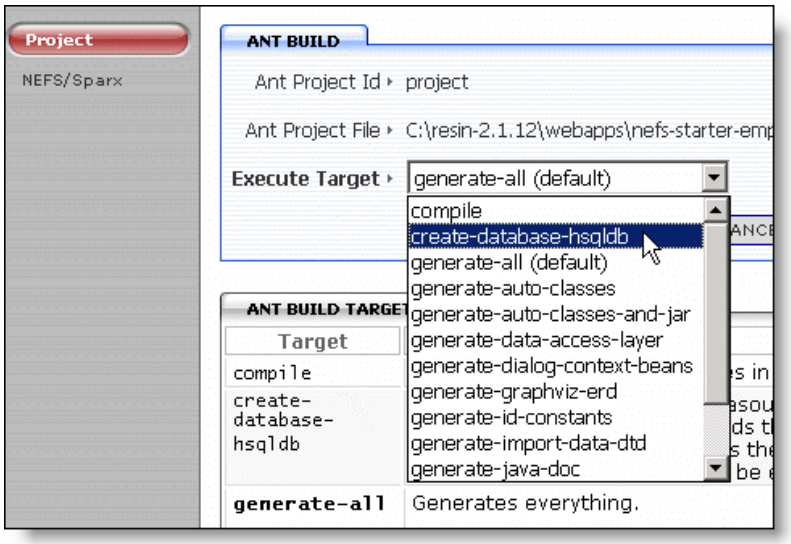
9.3. Generating Data Definition Language (DDL)

The DDL representation of your schema consists of the actual commands that you need to issue to a database to create the tables you specified in the schema and to populate them with any static data (such as the one stored in enumeration tables) if necessary. These commands are DBMS-specific.

9.3.1. Using the Ant Build in Console

To create the HSQL database and its DDL, you can use the Project | Ant Build section in the Console. In order to create the HSQL database, you must run the "`create-database-hsqldb`" target.

> **Note**
>
> Please note that you need the initial-and-test-data.xml file in order to create the HSQL database using the Ant Build Script. See Section 9.3.2, “Populating the HSQL Database with Test Data”



This erases the existing default datasource (Hypersonic database), generates the SQL DDL for the default schema, loads the SQL DDL (effectively creating the Hypersonic SQL database) and finally loads the 'starter' from XML files using Sparx import from XML feature. The Console displays different messages during the HSQL database creation (as show below):

```
Apache Ant version 1.5.3 compiled on April 16 2003

generate-sql-ddl:
    [sparx] mysql DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-I
    [sparx] ansi DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-IN
    [sparx] mssql DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-I
    [sparx] postgres DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WE
    [sparx] oracle DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-
    [sparx] hsqldb DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-

create-database-hsqldb:
     [echo] Hypersonic database name is 'db' and will be stored in C:/resin
    [sparx] hsqldb DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-
      [sql] Executing file: C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-
      [sql] 23 of 23 SQL statements executed successfully
    [sparx] BOOK_INFO: successful rows=102, unsuccessful rows=0, time=2093
```

Note

Please note that this target should be executed anytime the default schema is modified.

With this final step completed, you should be ready to add, update, delete and query data from the database using the Sparx Library. To do that, however, you need a user interface that will allow you to manipulate data as well as query what is stored in the database.

9.3.2. Populating the HSQL Database with Test Data

You will need some test data to be stored in the Library Application database. This will provide you with some initial data to test your application with. You can load this test data using the `WEB-INF/database/data/initial-and-test-data.xml` and `initial-and-test-data.xsl` files.

Important

The `initial-and-test-data.xml` file is necessary to create the HSQL database.

Example 5. Loading Initial Test Data into Asset and Borrower Tables

```
<?transform --xslt initial-and-test-data.xsl?> ❶


<!DOCTYPE dal SYSTEM "../defn/db-import.dtd"> ❷

<dal> ❸
    <generate-asset-records count="35"/> ❹

    <borrower last-name="Doe" first-name="John"/> ❺
    <borrower last-name="Doe" first-name="Jane"/>
</dal>
```

❶ The xdm-transform processing instruction tells Sparx to filter special tags through the XSLT before processing.
❷ The db-import.dtd is the DTD that is automatically created (by the Ant Build) based on the schema that is provided by the `schema` tag. The DTD is always called *`dbname`*-import.dtd, where *`dbname`* is the name specified in the `schema` tag.
❸ The root tag for the initial-and-test-data.xml is `dal` .
❹ Setting the value for the number of asset records to be generated automatically by the XSL.
❺ Adding the test record in the `Borrower` table.

You may optionally use XSL to automate the creation of a large number of test data.

Example 6. Using XSL to Generate Test Data

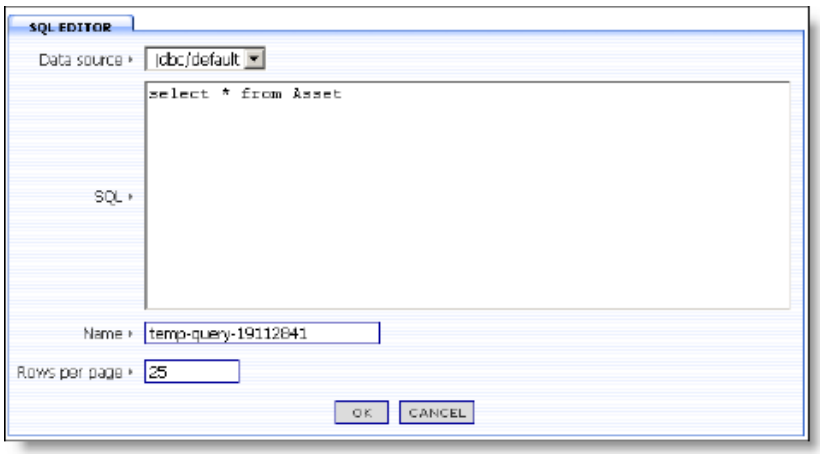
```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
     xmlns:dal="http://www.netspective.org/Framework/Axiom/DataAccessLayer"> ❶

    <xsl:output method="xml" indent="yes"/> ❷
```

```
    <xsl:template match="*"> ❸
        <xsl:copy>
            <xsl:copy-of select="attribute::*[. != '']"/>
            <xsl:apply-templates/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="generate-asset-records"> ❹
        <xsl:call-template name="iterate-one">
            <xsl:with-param name="x">0</xsl:with-param>
            <xsl:with-param name="count"><xsl:value-of select="@count"/></xsl:with-param>
        </xsl:call-template>
    </xsl:template>


    <xsl:template name="iterate-one"> ❺
        <xsl:param name="x"/>
        <xsl:param name="count"/>


        <asset> ❻
            <xsl:attribute name="name">Item <xsl:value-of select="$x"/></xsl:attribute>
            <xsl:attribute name="type">Book</xsl:attribute>
            <xsl:attribute name="quantity">5</xsl:attribute>
        </asset>


        <xsl:if test="$x &lt; $count"> ❼
          <xsl:call-template name="iterate-one">
            <xsl:with-param name="x" select="$x + 1"/>
            <xsl:with-param name="count"><xsl:value-of select="$count"/></xsl:with-param>
          </xsl:call-template>
        </xsl:if>
    </xsl:template>
</xsl:stylesheet>
```

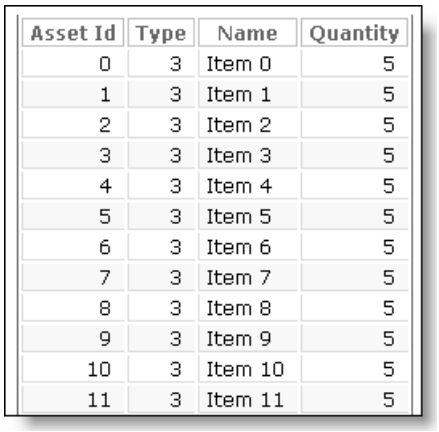
❶ Defining the Axiom Data Access Layer namespace prefix `dal`.
❷ Defines the output of the XSL (XML in this case).
❸ Defines the default template.
❹ The template that you will use in your XML file. This template receives, through the parameter `count`, the number of records to be added. It calls another custom template named "iterate-one". The value of `count` parameter is also sent to the called template.
❺ This is the iterate-one template which is called by the generate-asset-records template. It generates the values for all the fields of `Asset` table.
❻ The `Asset` XML records are generated using the value of parameter `x`, which changes on every iteration of the iterate-one template.
❼ This tag is responsible for recursively calling the iterate-one template. It checks the value of the parameter `x` and, if it is less than `count`, increases it by one to call the iterate-one template with this new value.

9.3.3. Unit Testing the HSQL Database

You may test the newly created HSQL database through Data Management | Data Sources | SQL Explorer option in the Console. Enter a SQL query for the Asset table in the SQL Editor.



To execute the test query, click the **OK** button. This displays the query result form (as shown below):



Asset Id	Type	Name	Quantity
0	3	Item 0	5
1	3	Item 1	5
2	3	Item 2	5
3	3	Item 3	5
4	3	Item 4	5
5	3	Item 5	5
6	3	Item 6	5
7	3	Item 7	5
8	3	Item 8	5
9	3	Item 9	5
10	3	Item 10	5
11	3	Item 11	5

9.4. Creating the Data Management Layer

9.4.1. Declaring Queries

The Library Application uses different queries to retrieve required information from the library database.

Example 7. Static query to get all the library assets

```
<queries package="library"> ❶

 <query name="get-all-assets"> ❷

   select asset id, name, Asset Type.caption, quantity, ❸
          (select count(*) from Asset Loan where Asset Loan.asset id = asset id),
          (select count(*) from Asset Loan where Asset Loan.asset id = asset id
                                                 and Asset Loan.returned = 0)
   from Asset, Asset Type
   where Asset.type = Asset Type.id
   order by name;



   <presentation> ❹
     <panel name="main" height="300">
       <frame heading="static:All available assets"/>
       <report>
         <actions>
           <action type="add" caption="Add Asset" redirect="page-id:/asset/add"/> ❺
           <action type="edit" redirect="page-id:/asset/edit?asset id=${0}"/> ❻
           <action type="delete" redirect="page-id:/asset/delete?asset id=${0}"/>
         </actions>
         <column heading="ID" format="plain" redirect="page-id:/asset/edit?asset id=${0}"/> ❼
         <column heading="Name" />
         <column heading="Type"/>
         <column heading="Quantity" align="right"/>
         <column heading="Total Loans" align="right"/>
         <column heading="Currently Out" align="right"/>
       </report>
     </panel>
   </presentation>
 </query>
```

❶ All the static queries in Axiom must belong to a statement package represented by `queries` XML tag. The statement package is identified by its name (`library` in the above example). You can define multiple packages within your application's project.xml file.

❷ The static query is defined (with or without bind parameters) by using the `query` tag. Each query is identified by its name (`get-all-assets` in the above example).

❸ The static SQL (with or without bind parameters) is declared under the query tag. The query in this example declares a join on the `Asset` and `Asset_Type` tables using the `type` (from `Asset`) and `id` (from `Asset_Type`) fields.

❹ Defining the presentation for displaying the query result. Each `query` tag contains a `presentation`tag associated with it. This tag defines presentation aspects of the query.

❺ You can declare a set of actions for your query's result. These actions provide a way to perform different functions on the displayed query result. Sparx has pre-defined action types for performing add, edit and delete operations on the selected record within your query result.

The action tag is used to define individual actions (`add` in this case). The `action` tag may specify a `redirect` attribute to automatically redirect to another page whenever it is chosen. The name of the redirect page is supplied using the `page-id` value source.

❻ Declaring `edit` action. You may also supply parameters and their values within the redirect URL (selected record's `id` field in this case). ${XXX} specifies a dynamic replacement. In the above example, *${0}* is used to indicate that it should be replaced with the value of the first column of current row.

❼ Every SQL report contains `column` tags that are used to customize the appearance of a particular column or, more accurately, a particular field. You may also specify a redirect page URL (`edit` page in this case).

Example 8. Static query to get all the loans

```
<query name="get-all-asset-loans"> ❶

  select asset_loan_id, Asset.name, last_name + ', ' + first_name, ❷
         Loan_Type.caption, loan_begin_date, loan_end_date, returned
  from Asset, Asset Loan, Loan Type, Borrower
  where Asset.asset_id = Asset_Loan.asset_id and loan_type = Loan_Type.id and
        Asset_Loan.borrower_id = Borrower.borrower_id
  order by loan begin date desc


  <presentation> ❸
    <panel name="main">
      <frame heading="static:Loans for Asset"/>
      <report>
        <actions>
         <action type="edit" redirect="page-id:/loan/edit?asset loan id=${0}"/> ❹
         <action type="delete" redirect="page-id:/loan/delete?asset loan id=${0}"/> ❺
        </actions>
        <column heading="ID" format="plain" ❻
                             redirect="page-id:/loan/edit?asset_loan_id=${0}"/>
        <column heading="Asset"/>
        <column heading="Borrower"/>


        <column heading="Type"/>
        <column heading="Begin"/>
        <column heading="End"/>
        <column heading="Returned"/>
      </report>
    </panel>
  </presentation>
</query>
```

❶ Defining query to retrieve 'all' the asset loans from the database.

❷ The query declares a join on the `Asset`, `Asset_Loan`, `Loan_Type` and `Borrower` tables.

❸ Defining the presentation for displaying the query result.

❹ Defining `edit` action for the asset loan. The `action` tag specifies a `redirect` attribute to automatically redirect to `edit` page.

❺ Defining `delete` action for the asset loan. The `action` tag specifies a `redirect` attribute to automatically redirect to `delete` page.

❻ Every SQL report contains `column` tags that are used to customize the appearance of a particular column or, more accurately, a particular field. You may also specify a redirect page URL (`edit` page in this case). The `format` attribute specifies the format (`plain` in the above example) to be used to display the column data.

Example 9. Static query to retrieve a specific loan

```
<query name="get-asset-loans"> ❶
```

```
  select
    asset loan id, last name + ', ' + first name, Loan Type.caption,
    loan_begin_date, loan_end_date, returned
  from
    Asset Loan, Loan Type, Borrower
  where
    loan_type = Loan_Type.id and
    Asset Loan.borrower id = Borrower.borrower id and
    Asset_Loan.asset_id = ? ❷


  order
    by loan begin date desc

  <params>
    <param value="request:asset_id"/> ❸
  </params>

  <presentation>
    <panel name="main">
      <frame heading="static:Loans for Asset"/>
      <report>


        <actions>
          <action type="add" caption="Add Loan"
            redirect="page-id:/loan/add?simple-expr:asset id=${request:asset id}"/> ❹
          <action type="edit" redirect="page-id:/loan/edit?asset_loan_id=${0}"/>
          <action type="delete" redirect="page-id:/loan/delete?asset_loan_id=${0}"/>
        </actions>
        <column heading="ID" format="plain"
                             redirect="page-id:/loan/edit?asset_loan_id=${0}"/>
        <column heading="Borrower"/>
        <column heading="Type"/>
        <column heading="Begin"/>
        <column heading="End"/>
        <column heading="Returned"/>
      </report>
    </panel>
  </presentation>
</query>
```

❶ Defining query to retrieve a specific loan from the database.
❷ The ID for the specific asset to be retrieved will be replaced with the selected value dynamically.
❸ Defining bound parameter for the query. This parameter represents the asset_id for the asset whose information is to be retrieved. The asset ID is retrieved from the request parameter named `asset_id`.
❹ Defining `add` action for the asset loan. The `action` tag specifies a `redirect` attribute to automatically redirect to `add` page. The asset ID is retrieved from request parameter named `asset_id`. This asset ID is in turn passed to the `add` page as the request parameter.

Example 10. Static query to retrieve all loans of a borrower

```
<query name="get-borrower-loans"> ❶

  select
    asset loan id, Asset.name, Loan Type.caption, loan begin date,
    loan end date, returned
  from
    Asset, Asset_Loan, Loan_Type
  where
    loan_type = Loan_Type.id and
    Asset Loan.asset id = Asset.asset id and
    Asset Loan.borrower id = ? ❷
  order by
    loan begin date desc

  <params>
    <param value="request:borrower id"/> ❸
  </params>

  <presentation>
    <panel name="main">
      <frame heading="static:Loans for Borrower"/>
```

```
      <report>
        <actions>
          <action type="edit" redirect="page-id:/loan/edit?asset_loan_id=${0}"/>
          <action type="delete" redirect="page-id:/loan/delete?asset_loan_id=${0}"/>
        </actions>
        <column heading="ID" format="plain"
                             redirect="page-id:/loan/edit?asset_loan_id=${0}"/>
        <column heading="Asset"/>
        <column heading="Type"/>
        <column heading="Begin"/>
        <column heading="End"/>
        <column heading="Returned"/>
      </report>
    </panel>
  </presentation>
</query>
```

❶ Defining query to retrieve all loans of a specific borrower.
❷ The ID for the specific borrower whose loans are to be retrieved. The value will be replaced with the selected value dynamically.
❸ Defining bound parameter for the query. This parameter represents the borrower_id for the borrower whose loan information is to be retrieved. The borrower ID is retrieved from the request parameter named `borrower_id`.

Example 11. Static query to retrieve all the borrowers

```
<query name="get-all-borrowers"> ❶

  select borrower id, last name, first name,
        (select count(*) from Asset_Loan where Asset_Loan.borrower_id = borrower_id),
        (select count(*) from Asset_Loan where Asset_Loan.borrower_id = borrower_id
         and Asset Loan.returned = 0)
  from Borrower
  order by last_name, first_name

  <presentation>
    <panel name="main">
      <frame heading="static:All available borrowers"/>
      <report>
        <actions>
          <action type="add" caption="Add Borrower" redirect="page-id:/borrower/add"/>
          <action type="edit" redirect="page-id:/borrower/edit?borrower_id=${0}"/> ❷
          <action type="delete" redirect="page-id:/borrower/delete?borrower id=${0}"/>
        </actions>


        <column heading="ID" format="plain"
                redirect="page-id:/borrower/edit?borrower_id=${0}"/> ❸
        <column heading="Last Name" />
        <column heading="First Name"/>
        <column heading="Borrowed" align="right"/>
        <column heading="Unreturned" align="right"/>
      </report>
    </panel>
  </presentation>
</query>
```

❶ Defining query to retrieve all the borrowers.
❷ Defining `edit` action for the borrowers. The `action` tag specifies a `redirect` attribute to automatically redirect to `edit` page.
❸ Specifying the redirect page URL (`edit` page in this case). The selected borrower's ID is also passed to the redirected page.

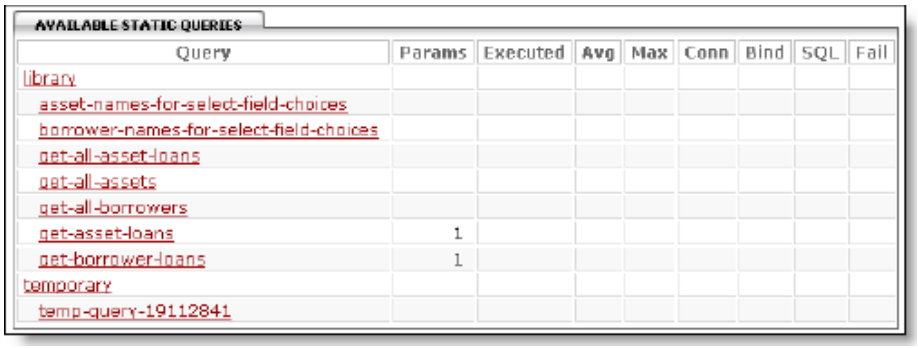
Example 12. Static queries to get all the assets and borrowers (to fill the select fields)

```
<query name="asset-names-for-select-field-choices"> ❶
  select name, asset id
  from Asset
</query>

<query name="borrower-names-for-select-field-choices"> ❷
  select (last_name + ', ' + first_name) as "Name", borrower_id
  from Borrower
</query>
```

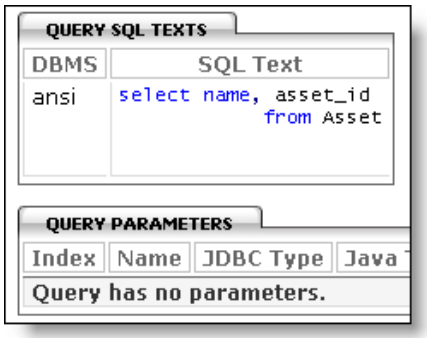
❶ Defining the query to retrieve all the assets. This information is used to fill the select fields displaying the Assets.
❷ Defining the query to retrieve all the borrowers. This information is used to fill the select fields displaying the Borrowers.

9.4.2. Unit Testing a Static SQL

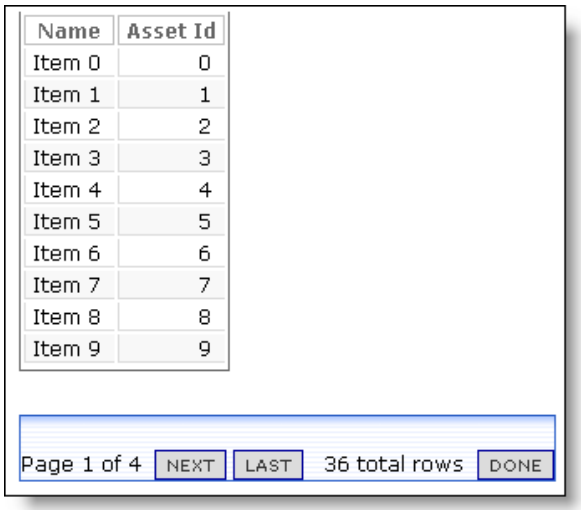
To test your newly defined static query, go to the Data Management | Static Queries section in the Console. This displays all the Static SQL statements defined in your `project.xml` file.



Click on the `asset-names-for-select-field-choices` query to see the SQL statement.



Click on the Unit Test option in the left menu bar. The unit test page is displayed containing a form/dialog for specifying the number of record rows to be displayed per page. By default, the number of records displayed per page is 10 records per page. Enter the new value in this field if you want to change the number of records displayed per page. Click the **OK** button. The query is executed and its result is displayed (as shown below):



10. Creating the Presentation Layer

The presentation layer of your Library Application comprises the following pages:

- Home Page
- Assets Page
- Borrowers Page
- Loans Page
- Console Page
- NEFS Sample Apps Home Page

10.1. Creating the Home Page

Your Library Application Home page will display list of all the assets and the borrowers. These lists will be displayed in two separate panels. Both the lists will also have Add, Edit and Delete options.

Sparx handles the pages that an end user sees and the transfer of control from one page to another using URIs and URLs. This is called `navigation`. Once declared using XML, Sparx can automatically manage the visual and operational end-user control of the navigation from one area of your application to another. You simply define the rules for what happens when a user visits a page and Sparx takes care of the rest.

Following is the XML declaration for your Library Application navigation:

Example 13. Creating "Home" Page

```
<navigation-tree name="app" default="yes"> ❶
  <page name="home" default="yes" caption="Home"> ❷
    <panels style="horizontal"> ❸

      <panel type="command" ❹
             command="query,library.get-all-assets,-,-,-,record-manager-compressed"/>


      <panel type="command" ❺
             command="query,library.get-all-borrowers,-,-,-,record-manager-compressed"/>
    </panels>
  </page>
</navigation-tree>
```

❶ The `navigation-tree` tag starts out the definition of the navigation tree. Each tree has a `name` (`app` in this case). The `name` attribute is required and must be unique across all navigation trees. The `caption` attribute is a value source and is always required. A tree may be marked with `default=yes` if it is to be the default tree. Which tree is actually used by the application may be specified as a server parameter or chosen dynamically at runtime based on some processing rules.

❷ The `page` tag begins a definition of a single page and appears under the `navigation-tree` tag. The `name` attribute is required and must be unique within the navigation tree in which it is defined. A page may be marked with `default=yes` if it is to be the default page. Which page is actually used by the application may be specified as a servlet parameter or chosen dynamically at runtime based on some processing rules.

❸ The `panels` tag starts a definition of a page body and appears under the `page` tag. Its contents are handled by Sparx by laying out pre-defined panels similar to the way portals lay out their content. The `panels` tag ends up as an instance of the `com.netspective.sparx.panel.HtmlLayoutPanel` class. The `style` attribute specifies how the panels will be arranged. In the above example, the panels will be arranged horizontally.

❹ Declaring the panel containing the list of all assets. The value of `type` attribute is set to `'command'` which specifies that the panels will be displaying the result of a command's execution.

The `command` attribute calls `execute()` on an instance of the `com.netspective.sparx.command.Command` interface and includes the content of the execution as the content of the page. The command attribute in this example executes the query `get-all-asset` from the query package named `library` (see Section 9.4.1, “Declaring Queries”). It also specifies one of the pre-defined report skins (`record-manager-compressed` in this case) for the displayed report.

❺ Declaring the panel containing the list of all borrowers. The value of `type` attribute is again set to `'command'`. The command attribute in this example executes the query `get-all-borrowers` from the query package named `library` (see Section 9.4.1, “Declaring Queries”). The `record-manager-compressed` skin is used for displaying the report.

10.2. Creating the Assets Page

The next step is to create a page for managing library assets. The Assets management page will contains links to pages for viewing, adding, editing and deleting the assets. Sparx allows creation of nested pages. i.e. pages that contain further pages. The following XML declaration creates the Assets management page:

Example 14. Creating "Assets" Page

```
<page name="asset" caption="Assets" dialog-next-action-url="page-id:/asset/view"> ❶
  <page name="view" caption="View Assets" ❷
        command="query,library.get-all-assets,-,-,-,record-manager-compressed"/>
  <page name="add" caption="Add Asset" command="dialog,schema.db.Asset,add"/> ❸
  <page name="edit" caption="Edit Asset"
        require-request-param="asset_id" retain-params="asset_id"> ❹
  <missing-params-body> ❺
      Please choose an asset to edit from the &lt;a href='view'&gt; assets
      list&lt;/a&gt;.
    </missing-params-body>


    <panels style="horizontal"> ❻
      <panel type="command" command="dialog,schema.db.Asset,edit"/> ❼
      <panel type="command" ❽
        command="query,library.get-asset-loans,-,-,-,record-manager-compressed"/>
    </panels>
  </page>


  <page name="delete" caption="Delete Asset" command="dialog,schema.db.Asset,delete" ❾
        require-request-param="asset id" retain-params="asset id">
    <missing-params-body> ❿
      Please choose an asset to delete from the &lt;a href='view'&gt; assets
      list&lt;/a&gt;.
    </missing-params-body>
  </page>
</page>
```

❶ The `page` tag in the example code defines Library Application's `Assets` page. The name of the page is set to `asset` and the caption for the page is `'Assets'`. The `dialog-next-action-url` attribute sets the next action URL (to be used instead of a next action provider) for this particular page. This example sets the '`View Assets`' page as the page to be called after execution of an action on any page within this page tree.

❷ Declaring the '`View Assets`' page within the Assets page tree. The command attribute executes the query get-all-assets to display a list of all the assets of the library.

❸ Declaring the '`Add Assets`' page within the Assets page tree. The `command` attribute displays and executes a dialog box for Asset table using the "`add`" perspective.

❹ Declaring the `Edit Asset` page. The `require-request-parameter` attribute specifies `asset_id` as the required request parameter. The `asset_id` parameter contains unique identifier for the asset record that is being edited. The `retain-params` attribute specifies that the "asset_id" parameter's value will be sent to another page (tab) when that tab/page is clicked.

❺ The `missing-params-body` tag is used to optionally produce automatic error message when required request parameter (`'asset_id'` in this case) is not provided.

❻ Declaring the panels for `Edit Asset` page. The panels will be arranged horizontally.

❼ The panels is used to display and execute a dialog box for `Asset` table using the "`edit`" perspective.

❽ Declaring the panel containing the list of all loans associated with the selected asset. The value of `type` attribute is again set to `'command'`. The command attribute in this example executes the query `get-asset-loans`. The `record-manager-compressed` skin is used for displaying the report.

❾ Declaring the `Delete Asset` page. The `require-request-parameter` attribute specifies `asset_id` as the required request parameter. The `asset_id` parameter contains unique identifier for the asset record that is being edited. The `retain-params` attribute specifies that the "asset_id" parameter's value will be sent to another page (tab) when that tab/page is clicked. The `command` attribute displays and executes a dialog box for Asset table using the "`delete`" perspective.

❿ The `missing-params-body` tag is used to optionally produce automatic error message when required request parameter `('asset_id'` in this case) is not provided.

10.3. Creating the Borrowers Page

The next step is to create a page for managing the borrowers. The Borrowers management page will contains links to pages for viewing, adding, editing and deleting the borrowers. The following XML declaration creates the Borrowers management page:

Example 15. Creating "Borrowers" Page

```
<page name="borrower" caption="Borrowers" dialog-next-action-url="page-id:/borrower/view"> ❶
  <page name="view" caption="View Borrowers" ❷
        command="query,library.get-all-borrowers,-,-,-,record-manager-compressed"/
  <page name="add" caption="Add Borrower" command="dialog,schema.db.borrower,add"/> ❸
  <page name="edit" caption="Edit Borrower" command="dialog,schema.db.borrower,edit" ❹
        require-request-param="borrower id" retain-params="borrower id">
    <missing-params-body> ❺
      Please choose a borrower to edit from the &lt;a href='view'&gt; borrowers
      list&lt;/a&gt;.
    </missing-params-body>
    <panels style="horizontal"> ❻
      <panel type="command" command="dialog,schema.db.Borrower,edit"/> ❼
      <panel type="command"
          command="query,library.get-borrower-loans,-,-,-,record-manager-compressed"/> ❽
    </panels>
  </page>


  <page name="delete" caption="Delete Borrower" command="dialog,schema.db.borrower,delete"
        require-request-param="borrower id" retain-params="borrower id"> ❾
    <missing-params-body> ❿
      Please choose a borrower to delete from the &lt;a href='view'&gt; borrowers
      list&lt;/a&gt;.
    </missing-params-body>
  </page>
</page>
```

❶ The `page` tag defines Library Application's `Borrowers` page. The name of the page is set to `borrower` and the caption for the page is `'Borrowers'`. The `dialog-next-action-url` attribute sets the '`View Borrowers`' page as the page to be called after execution of an action on any page within this page tree.

❷ Declaring the '`View Borrowers`' page within the `Borrowers` page tree. The command attribute executes the query `get-all-borrowers` to display a list of all the borrowers.

❸ Declaring the '`Add Borrowers`' page within the Borrowers page tree. The `command` attribute displays and executes a dialog box for Borrower table using the "`add`" perspective.

❹ Declaring the `Edit Borrower` page. The `require-request-parameter` attribute specifies `borrower_id` as the required request parameter. The `borrower_id` parameter contains unique identifier for the borrower record that is being edited. The `retain-params` attribute specifies that the "borrower_id" parameter's value will be sent to another page (tab) when that tab/page is clicked.

❺ The `missing-params-body` tag is used to optionally produce automatic error message when required request parameter `('borrower_id'` in this case) is not provided.

❻ Declaring the panels for `Edit Borrower` page. The panels will be arranged horizontally.

❼ The panels is used to display and execute a dialog box for Borrower table using the "`edit`" perspective.

❽ Declaring the panel containing the list of all loans taken by the selected borrower. The command attribute in this example executes the query `get-borrower-loans`. The `record-manager-compressed` skin is used for displaying the report.

❾ Declaring the `Delete Borrower` page. The `require-request-parameter` attribute specifies `borrower_id` as the required request parameter. The `borrower_id` parameter contains unique identifier for the borrower record that is being edited. The `retain-params` attribute specifies that the "borrower_id" parameter's value will be sent to another page (tab) when that tab/page is clicked. The `command` attribute displays and executes a dialog box for Borrower table using the "`delete`" perspective.

❿ The `missing-params-body` tag is used to optionally produce automatic error message when required request parameter (`'borrower_id'` in this case) is not provided.

10.4. Creating the Loans Page

The next step is to create a page for managing the loans. The Loans management page will contains links to pages for viewing, adding, editing and deleting the loans. The following XML declaration creates the Loans management page:

Example 16. Creating "Loans" Page

```
<page name="loan" caption="Loans" dialog-next-action-url="page-id:/loan/view"> ❶
  <page name="view" caption="View Loans" ❷
        command="query,library.get-all-asset-loans,-,-,-,record-manager-compressed"/>
  <page name="add" caption="Add Loan" command="dialog,schema.db.Asset Loan,add"/> ❸


  <page name="edit" caption="Edit Loan" command="dialog,schema.db.Asset Loan,edit" ❹
        require-request-param="asset loan id" retain-params="asset loan id">
    <missing-params-body> ❺
      Please choose an asset and select a loan from the &lt;a
      href='../asset/view'&gt;asset list&lt;/a&gt;.
    </missing-params-body>

  </page>
  <page name="delete" caption="Delete loan" command="dialog,schema.db.Asset Loan,delete" ❻
        require-request-param="asset loan id" retain-params="asset loan id">


    <missing-params-body> ❼
      Please choose an asset and select a loan from the
      &lt;a href='../asset/view'&gt;asset list&lt;/a&gt;.
    </missing-params-body>
  </page>
</page>
```

❶ The `page` tag defines Library Application's `Loans` page. The name of the page is set to `loan` and the caption for the page is `'Loans'`. The `dialog-next-action-url` attribute sets the '`View Loans`' page as the page to be called after execution of an action on any page within this page tree.

❷ Declaring the '`View Loans`' page within the `Loans` page tree. The command attribute executes the query `get-all-asset-loans` to display a list of all the loans.

❸ Declaring the '`Add Loans`' page within the Loans page tree. The `command` attribute displays and executes a dialog box for Asset_Loan table using the "`add`" perspective.

❹ Declaring the `Edit Loan` page. The `require-request-parameter` attribute specifies `asset_loan_id` as the required request parameter. The `asset_loan_id` parameter contains unique identifier for the asset loan record that is being edited. The `retain-params` attribute specifies that the "`asset_loan_id`" parameter's value will be sent to another page (tab) when that tab/page is clicked.

❺ The `missing-params-body` tag is used to optionally produce automatic error message when required request parameter (`'asset_loan_id'` in this case) is not provided.

❻ Declaring the `Delete Loan` page. The `require-request-parameter` attribute specifies `asset_loan_id` as the required request parameter. The `asset_loan_id` parameter contains unique identifier for the asset loan record that is being edited. The `retain-params` attribute specifies that the "`asset_loan_id`" parameter's value will be sent to another page (tab) when that tab/page is clicked. The `command` attribute displays and executes a dialog box for Asset_Loan table using the "`delete`" perspective.

❼ The `missing-params-body` tag is used to optionally produce automatic error message when required request parameter (`'asset_loan_id'` in this case) is not provided.

10.5. Creating the Console Page

Like every other Sparx application, your Library Application also has an associated Console. You may provide the access to this Console using the following XML declaration:

Example 17. Linking to the Library Application Console

```
<navigation-tree name="app" default="yes">
 <page name="console" caption="Console" redirect="servlet-context-uri:/console"/> (1)
</navigation-tree>
```

(1) The `page` tag declares `Console` page for the `Library Application`. The `redirect` attribute is set (using the `servlet-context-uri` value source) to automatically redirect to the application Console.

10.6. Creating the Sample Apps Home Page

As a last step to the creation of your Library Application, you will add a Sample Apps Home page using the following XML declaration:

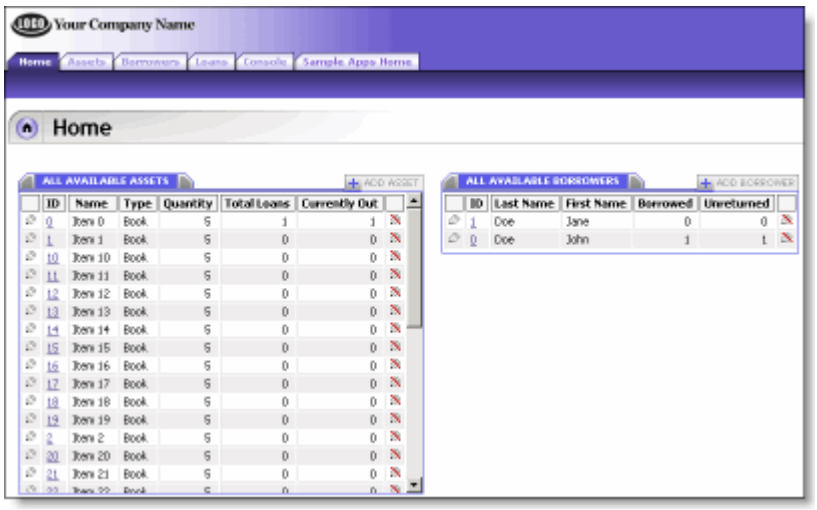
Example 18. Creating Sample Apps Home Page for Library Application

```
<navigation-tree name="app" default="yes">
   <page name="sample-apps-home" caption="Sample Apps Home" (1)
                                 redirect="netspective-url:nefs-sample-apps-home"/>
</navigation-tree>
```

(1) The `page` tag declares the `Sample Apps Home` page. The `redirect` attribute specifies the URL of the Netspective `Sample Apps` page using the `netspective-url` value source.

10.7. Testing the Library Application

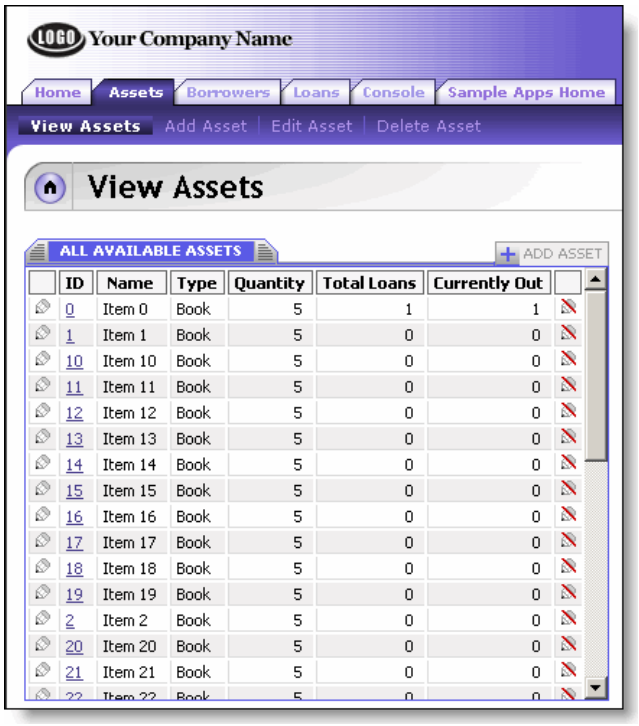
Congratulations! The Library Application is complete. Now you should open up your browser window and go to the Library Application's main page using the URL: http://*host*:*port*/*appName*. Assuming you are using localhost and port `8080` for your Resin server and `nefs-sample-library` as your application's name, the URL for Library Application Home Page becomes: http://localhost:8080/nefs-sample-library[13]



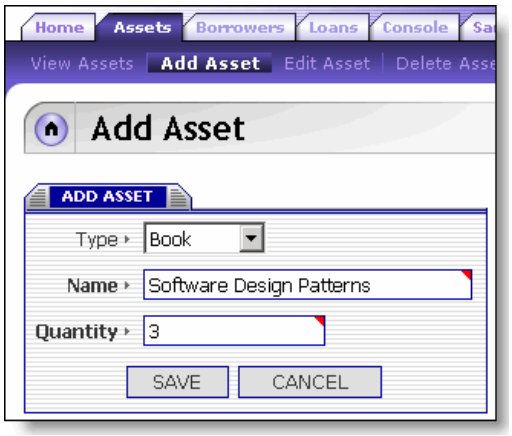
10.7.1. Assets Page

The Home page lists all the assets available in your Library Application database along with the available borrowers. Explore the Assets tab which contains View Assets, Add Asset, Edit Asset and Delete Asset pages. Use the View Assets page to view a list of all the available assets.

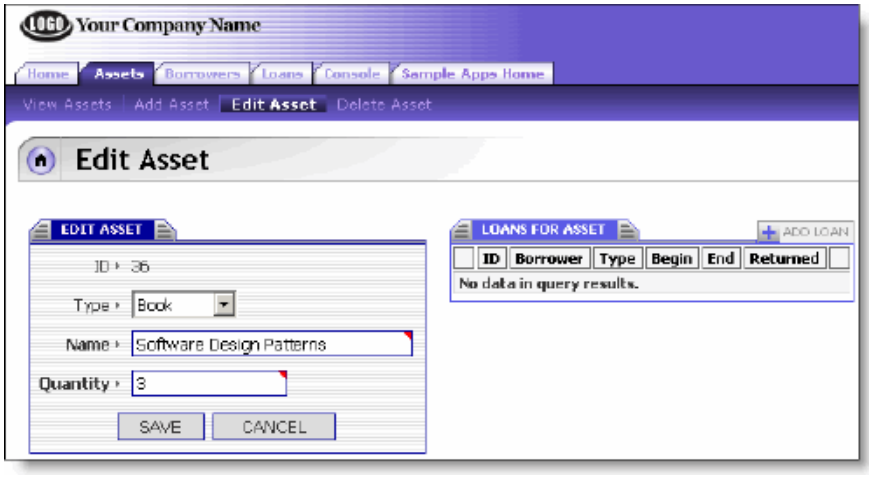
[13] http://localhost:8080/nefs-sample-books



Use the Add Asset page to add an asset.

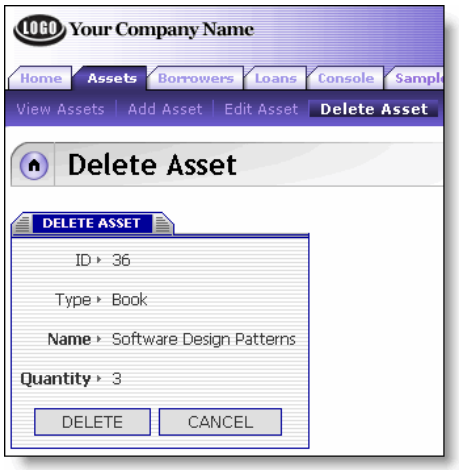


Enter the asset information and click the **Save** button. You can see the newly added asset in the list available on the Home or View Assets pages. You may also edit an existing asset's information by selecting it from the list of Assets using the Edit icon. This opens the Edit Assets page:



Try editing the details of the asset. Select the asset from the Asset List by clicking on its ID. The Edit Asset page is displayed containing the selected asset's information. The associated loan records are also shown in a separate panel on the same page.

Go to Home page and select the asset to delete. The Delete Asset page is displayed containing the selected asset's information. Click on the Delete button to delete the selected asset's record.

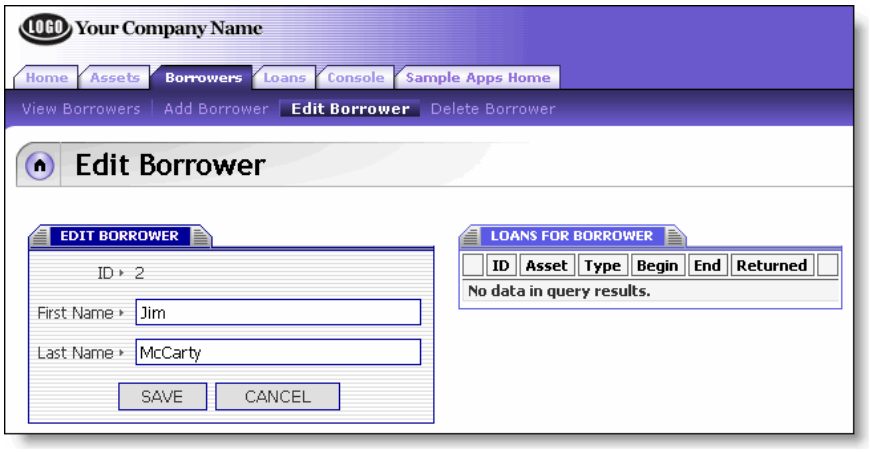


10.7.2. Borrowers Page

Explore the Borrowers tab which contains View Borrowers, Add Borrower, Edit Borrower and Delete Borrower pages. Use the View Borrowers page to view a list of all the available borrowers.

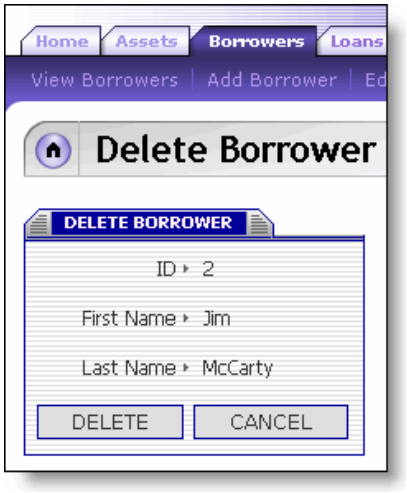
Use the Add Borrower page to add a borrower.

Enter the borrower information and click the **Save** button. You can see the newly added borrower in the list available on the Home or View Borrowers pages. You may also edit an existing borrower's information by selecting it from the list of Borrowers using the Edit icon. This opens the Edit Borrowers page:



Try editing the details of the borrower. Select the borrower from the Borrowers List by clicking on the borrower's ID. The Edit Borrower page is displayed containing the selected borrower's information. The associated loan records are also shown in a separate panel on the same page.

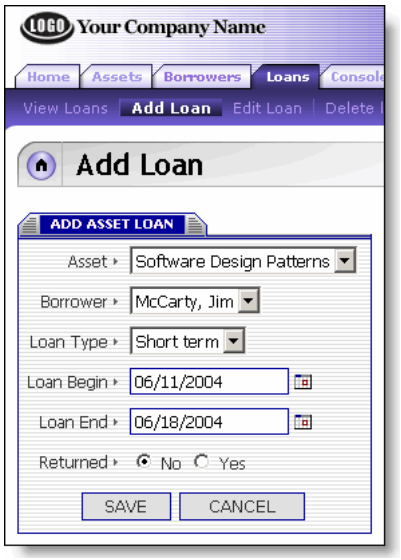
Go to Home page and select the borrower to delete. The Delete Borrower page is displayed containing the selected borrower's information. Click on the Delete button to delete the selected borrower's record.



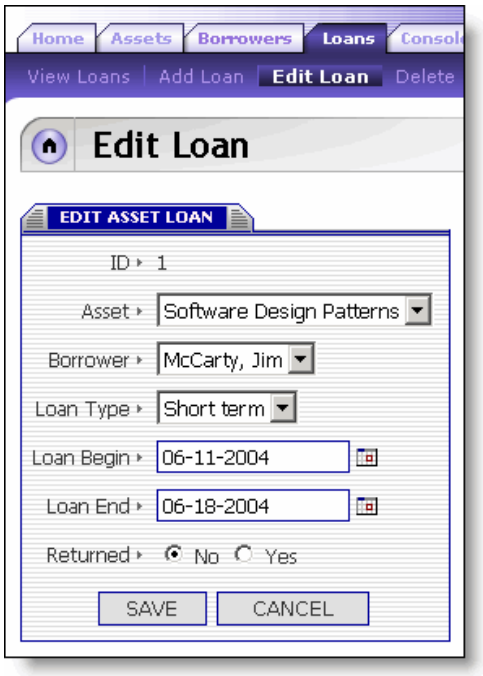
10.7.3. Loans Page

Explore the Loans tab which contains View Loans, Add Loan, Edit Loan and Delete Loan pages. Use the View Loans page to view a list of all the loans.

Use the Add Loan page to add a loan.



Enter the loan information and click the **Save** button. You can see the newly added loan in the list available on the View Loans pages. You may also edit an existing loan's information by selecting it from the list of Loans using the Edit icon. This opens the Edit Loan page:



Try editing the details of the loan. Select the loan from the Loans List by clicking on the loan's ID. The Edit Loan page is displayed containing the selected loan's information.

Go to View Loans page and select the loan to delete. The Delete Loan page is displayed containing the selected loan's information. Click on the Delete button to delete the selected loan's record.

10.7.4. Console Page

You may access the Library Application Console through the Console page. Use '`console`' and '`console`' (without quotes) for Console's `User Id` and `Password`.

10.7.5. Sample Apps Home Page

Try the Sample Apps Home page to see the NEFS Sample Apps Home Page loaded from the netspective web site.

11. Conclusion

Congratulations! Your Library application is now complete. This application is relatively more complex in its use of nested pages and multiple panels per page. You can now move on to more complex sample applications provided by Netspective Enterprise Suite to help you learn more features provided by NEFS.