

NEF Sample Books Application

Tutorial



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

(301) 879-3321

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

Copyright

Copyright © 1997-2004 Netspective Communications LLC. All Rights Reserved. Netspective, the Netspective logo, Sparx, and the Sparx logo, ACE, and the ACE logo are trademarks of Netspective, and may be registered in some jurisdictions.

Disclaimer and limitation of liability

Netspective and its suppliers assume no responsibility for any damage or loss resulting from the use of this tutorial. Netspective and its suppliers assume no responsibility for any loss or claims by third parties that may arise through the use of this software or documentation.

Customer Support

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

Books Application Tutorial

Table of Contents

1. Objectives	5
1.1. Learning Objectives Checklist.....	5
2. Assumptions.....	5
2.1. Downloading Java Developer's Kit (JDK).....	5
2.2. Application Server (Servlet Container)	5
3. Setting up the Starter Application.....	6
3.1. Downloading the Starter Application	6
3.2. Setting up the Starter Application (Using Application Server)	6
3.2.1. Auto Deploying the Starter Application (Using Application Server).....	6
3.2.2. Setting up the Starter Application Manually	6
3.3. Testing the Starter Application in a Browser.....	7
3.4. Verifying the Console	8
4. Key Concepts	8
4.1. The NEF Application Directory Structure	8
4.2. The NEF Project File (project.xml)	10
4.3. web.xml.....	10
5. Functionality	10
6. Design	11
6.1. Application Design	11
6.2. Database Design.....	11
7. Renaming the Starter Application.....	11
8. The NEF Project File (project.xml)	11
8.1. Dissecting the project.xml	11
9. Creating the Data Layer	12
9.1. Setting up the Data Source.....	12
9.1.1. Unit Testing the Data Source	13
9.2. Creating the Schema	13
9.2.1. Unit Testing the Schema	15
9.3. Generating Data Definition Language (DDL)	16
9.3.1. Using the Ant Build in Console	16
9.3.2. Automating the HSQL Database Generation.....	17
9.3.3. Populating the HSQL Database with Test Data.....	18
9.3.4. DDL and HSQL Database Files.....	20
9.3.5. Unit Testing the HSQL Database	20
9.4. Creating the Data Management Layer	20
9.4.1. Declaring a Static Query	21
9.4.2. Presenting the Results of a Static Query	21
9.4.3. Unit Testing a Static SQL	22
9.4.4. Declaring a Dynamic Query (Query Definition)	23
9.4.5. Presenting the Results of a Dynamic Query (Query Definition)	24
9.4.6. Unit Testing a Query Definition	26
10. Creating the Presentation Layer.....	27
10.1. Creating the Home Page	27

10.2. Creating the Add Book Page.....	28
10.3. Creating the Edit Book Page.....	28
10.4. Creating the Delete Book Page	29
10.5. Creating the Search Books Page	29
10.6. Creating the Console Page	29
10.7. Creating the Sample Apps Home Page	30
10.8. Custom-Handling of Dialog's Next Action.....	30
10.9. Testing the Books Application.....	31
11. Moving to Another Database	35
12. Conclusion	36

1. Objectives

The Books Application is a project meant to get you familiar with the NEF development by creating a simple but complete and functional real world application. The Books Application deals with the user interface as well as the back end part of the application. This involves the database access and the implementation of business logic. The Books Application will lead you through everything it takes to get an actual application up and running with special focus on developing the SQL and data management layers of your application.

1.1. Learning Objectives Checklist

At the end of this tutorial, you should be able to understand:

- the NEF Application Directory Structure
- the NEF Project File
- how to set up the sample application
- use of NEF Starter Application
- creation and testing of Data Layer
- creation and testing of Presentation Layer

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)

[2] <http://jakarta.apache.org/tomcat>

[3] <http://www.caucho.com>

- 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.

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

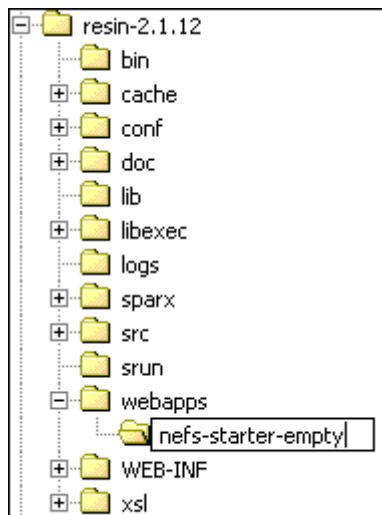
[4] <http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/server>

[5] <http://www.ibm.com/websphere>

[6] <http://www.oracle.com/appserver/>

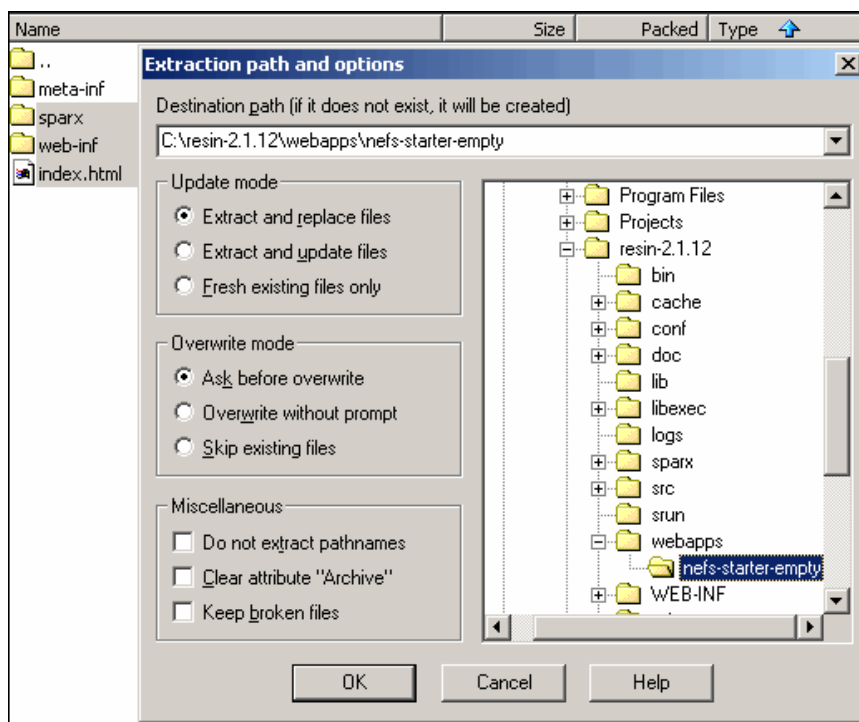
[7] <http://www.macromedia.com/software/jrun/>

[8] <http://www.caucho.com>



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.

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



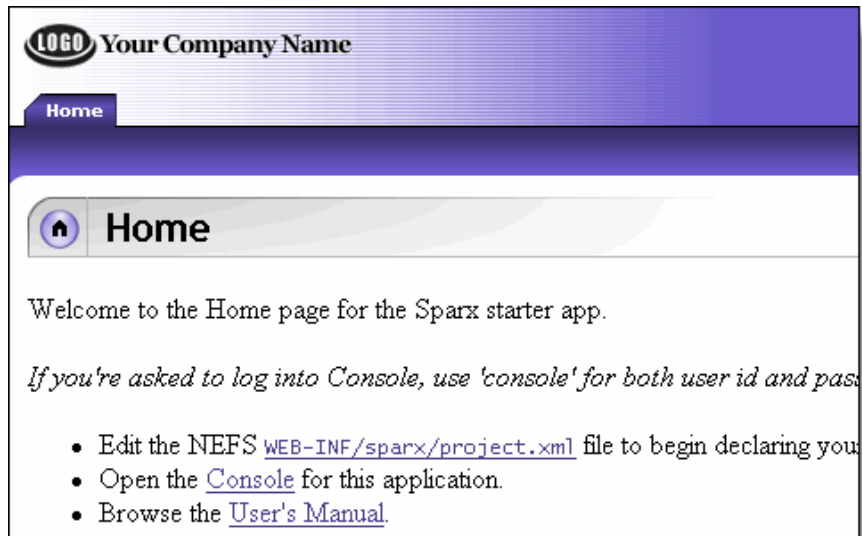
This creates an Application Directory Structure (see Section 4.1, “The NEF Application Directory Structure”), containing the necessary NEF files and sub-folders, within the `nefs-starter-empty` folder.

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.

[10] <http://www.winzip.com/download.htm>

[11] <http://www.rarsoft.com/download.htm>



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 relation 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.



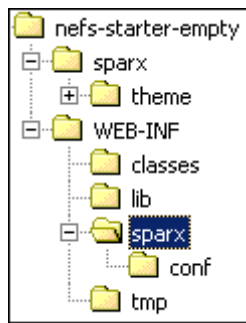
Congratulations! You now have an empty application upon which you can build. You can log in to your application's Console. The Console's default User Id is 'console' and the default Password is 'console' (each without quotes). Unless otherwise specified, that is the User Id and Password combination you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

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 <code>WEB-INF/sparx</code> directory to store its project component descriptors. There is usually at least one <code>project.xml</code> file and may contain subdirectories if you wish to split up your application component declarations. The <code>APP_ROOT/WEB-INF/sparx/project.xml</code> is the file that drives all of the Sparx functionality in your application.
WEB-INF/sparx/conf	This directory contains contains sample <code>web.xml</code> configuration files for different application servers like WebLogic, WebSphere, Resin and Tomcat. It also contains Ant build files for compiling your application's classes.

4.2. The NEF Project File (`project.xml`)

All of the NEF (Sparx, Axiom, and Commons) components are declared in an *input source* file known as the *Project File*. The Project File may be either a single file with all components or may be broken up into multiple files and pulled into the main Project File by using `<xdm:include>` tags.

The default Project File is `APP_ROOT/WEB-INF/sparx/project.xml` but may be configured to be a different file. Although you may choose an alternate location for the Project File, the remainder of this document assumes that the Project File is `APP_ROOT/WEB-INF/sparx/project.xml`.

4.3. `web.xml`

This file, which is a requirement of the Java Servlet Specification¹, contains information specific to the application server under which the application is running. As an example, it might contain a mapping of URLs to specific servlets that should handle those URLs. It might also contain references to data sources (i.e. database handles) referenced in the main application server's configuration. For application servers that run according to a standards compliant servlet specification, this file is necessary.

5. Functionality

The NEF Books Application is a small application meant to be used for a personal library of books. It allows you to track the books you have and add more books to your collection or edit information stored about existing books. It also allows you to search your collection for a particular book based on your own custom search criteria.

The overall functionality of the application is limited but complete. As such it demonstrates a few of the main types of data manipulation that developers need to take care of in every application. In the end, the goal is to show you just how much power can be wielded with just a few lines of XML and Java code when armed with the strength of Sparx.

¹You can get a copy of the Java Servlet Specification directly from Sun Microsystems' web site at the following URL: <http://java.sun.com/products/servlet/download.html>

6. Design

6.1. Application Design

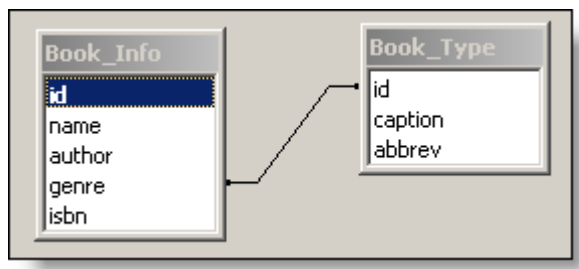
The Books Application is designed around the basic Sparx components. It will use static SQL and associated reports to help you track the books stored in the application. Searches, on the other hand, require a dynamically generated SQL in the form of a Query Definition.

The last aspect to take care of is the data storage. It is possible to use expensive abd overkill (in all certainty) database like Oracle for this application. Instead, the data storage of choice is the Java-based embedded database that is included in the Sparx Starter Application: HypersonicSQL^{TM2}. All the application components are included in the `project.xml` file for the Books Application.

6.2. Database Design

The Books Application deals with books and only books. Therefore, the information that needs to be stored in the database will be about books. The four pieces of information that the Books Application will store for this example are its ID, name, author, genre and ISBN. Of these, the genre is the only one that can be common across multiple books. In database language, the genre (book type) has a one-to-many relationship with the books.

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



The figure shows the entity-relationship diagram for the data we will be using. The database for the Books 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-books` as the root folder name for the Books Application.

8. The NEF Project File (project.xml)

8.1. Dissecting the project.xml

Example 1. Project File of Starter Application

²To learn more about HypersonicSQL, please go to <http://hsqldb.sourceforge.net/>TM

```
<?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 Books Application: the database.

9.1. Setting up the Data Source

To set up the Books 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 Books Application

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

class="com.netspective.axiom.connection.JakartaCommonsDbcConnectionProvider"> 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.

- ② 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.

- ③ The `driver-class` tag is used to provide the driver to be used for the specified database. Since the Books Application uses HSQL database, our sample code specifies the appropriate JDBC driver.
- ④ 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.

- ⑤ 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.
- ⑥ The `password` tag is used to provide the password for the log in user. The default `'sa'` user has no password.

Note

Please see the NEF User's Manual or review Tag Documentation in the Console to get further details on the each tag.

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

AVAILABLE DATA SOURCES		
Connection Provider: com.netspective.axiom.connection.JakartaCommonsDbcCon Underlying Implementation: com.netspective.axiom.connection.JakartaCommonsDb		
Identifier	Default	Properties
jdbc/default	Yes	Database: HSQL Database Engine Version 1.7.1 Driver: HSQL Database Engine Driver Version 1.7.1 URL: jdbc:hsqldb:C:\resin-2.1.12\webapps\nefs-start User: SA ResultSet Type: scrollable (insensitive) Database Policy: com.netspective.axiom.policy.HSqlDbDatabaseP

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 Books Application. It is a very simple

schema consisting of only two tables:

- *Book Type*: This table is used to store information about the different genres (types) of books.
- *Book Info*: This table stores all the attributes of the books.

The two tables are 1:n related by the genre (in Book Info) and type (in Book Type) fields. Once entered as XML, this schema is available for platform-independent database access from your application.

Note

The entire schema, and the other larger and more complex ones that you might develop for enterprise applications, can be represented entirely in `project.xml` file of your application.

Following is the code that creates the Books Application schema:

Example 3. Creating the Books Application Schema

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

  <table name="Book Info" abbrev="bkI" type="Presentation"> ❸
    <column name="id" type="text" size="10" primary-key="yes"
      descr="Unique ID for every book in the database"/> ❹
    <column name="name" type="text" size="64" descr="Name of the book"/>
    <column name="author" type="text" size="64" descr="Name of the author(s)"/>
    <column name="genre" lookup-ref="Book Type"/> ❺
    <column name="isbn" type="text" size="10"
      unique="yes" descr="The 10 digit ISBN number"/> ❻
  </table>

  <table name="Book Type" abbrev="bkT" type="Enumeration"> ❼
    <enumerations> ❽
      <enum>Science Fiction</enum>
      <enum>Mystery</enum>
      <enum>Business</enum>
      <enum>Information Technology</enum>
      <enum>Nuclear Physics</enum>
      <enum>Chemistry</enum>
    </enumerations>
  </table>
</schema>
```

- ❶ All schemas are declared using the `<schema>` tag and are uniquely identified with a name (db in this case). You may define multiple schemas within the same project.
- ❷ The default `com/netspective/axiom/conf/schema.xml` resource file contains dozens of built-in data types and table types that may be extended or just used. Types such as text, integer, float, currency, date, and even composite types such as duration are built-into Axiom.
- ❸ Each schema may contain one or more table tags to define database tables. Each table has a name and abbrev(iation) attribute associated with it. The table also has a type attribute. Axiom supplies various built-in table types. The type attribute for the `Book_Info` table is set to 'presentation' which gives it a default dialog.
- ❹ The column tag is used to define a field (column) in the table. Each column has name, type, size and descr(ption) attributes associated with it. To specify a field as the primary key of the table, set the primary-key attribute's value to 'yes'.
- ❺ The lookup-ref attribute specifies a general foreign key relationship. The format is `Table_X.Column_Y`. This creates a 1:1 or 1:N relationship from the defining column which references the foreign `Column_Y` of `Table_X`. The example code only specifies the table name (`Book_Type`) for the lookup-ref attribute. This defines the foreign key relationship between the referencing column (genre in this case) and the primary key column (type in this case) of the referenced (`Book_Type`) table.

Note

If you use this attribute, the type attribute is not required. It is set to the same type as the referenced column.

- ⑥ The unique attribute specifies whether the column's values should be unique. This means that no two rows should share the same value for this column. When this value is set to 'yes', this attribute creates a unique index based on this single column.

Note

If more than one column need to be unique (as a composite), use the index child element of the table element to create a unique index based on multiple columns.

- ⑦ The second table tag block is used to define the `Book_Type` table. This table is defined as an Enumeration. An Enumeration is a special type of table that is generated by Sparx. It consists of the following three fields per record:

- `id`: contains a unique value which is used to relate the enumeration table in a 1:n manner with other tables
- `caption`: contains non-null value that is used to provide a short description of each value in the enumeration
- `abbrev`: an optional field containing the abbreviation for the caption.

The syntax of an enumeration table is unlike that of regular tables. However, once parsed and interpreted, enumeration tables are translated into a set of regular tables for relational integrity purposes.

Note

An enumeration table is used to establish a 1:n relationship between an attribute of an object (e.g. genre of the book in this case) and the object itself (the book in this case). It does this by letting the `id` field of the attribute enumeration table be inserted as a foreign key in the table containing records for the object. In this particular scenario, the `lookup-ref` attribute of the `genre` field in the `Book_Info` table makes that happen.

- ⑧ Each enumeration type table contains an `enumerations` tag. The values for the enumeration table (book types in this case) are specified with the help of `enum` tag. Each value is enclosed within `enum` and `/enum` tag.

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 Books Application.

SCHEMA TABLES			
Overview			
SQL Table Name	XML Node Name	Columns	Indexes
Schema: 'db'			
Application Tables			
Book_Info	book-info	5	1
Enumeration Tables			
Book_Type	book-type	3	0
Lookup_Result_Type	lookup-result-type	3	0
Record_Status	record-status	3	1

There is a list of all the tables contained in the schema. It should list a total of 4 tables, of which the most important to you are the ones you explicitly created: `Book_Info` and `Book_Type`. Click on the `Book_Info` table to see more details about it.

TABLE HIERARCHY				
Overview Descriptions Table Type Inheritance				
SQL Table Name		Table Type(s) Inherited		
Schema: 'db'				
Application Tables				
Book_Info		Presentation		




TABLE COLUMNS				
Overview Descriptions Primary Keys Foreign Keys Validations				
	SQL Name	Domain	XML Name	SQL
	id	text	id	ansi: var mssql: va
	name	text	name	ansi: var mssql: va
	author	text	author	ansi: var mssql: va
	genre	enumeration-id	genre	ansi: inte oracle: n
	isbn	text	isbn	ansi: var mssql: va

TABLE INDEXES				
---------------	--	--	--	--

As you can see, the detailed view of the Book_Info tables gives a lot of information about the table and the information stored in it. For each field in the table, you can see its name, data type, actual SQL data type it was created as, whether it is a field referencing other fields (e.g. genre field in the Book_Info table) and the index(es) defined on this table.

You can view the details for the Book_Info and other tables in the same manner.

9.3. Generating Data Definition Language (DDL)

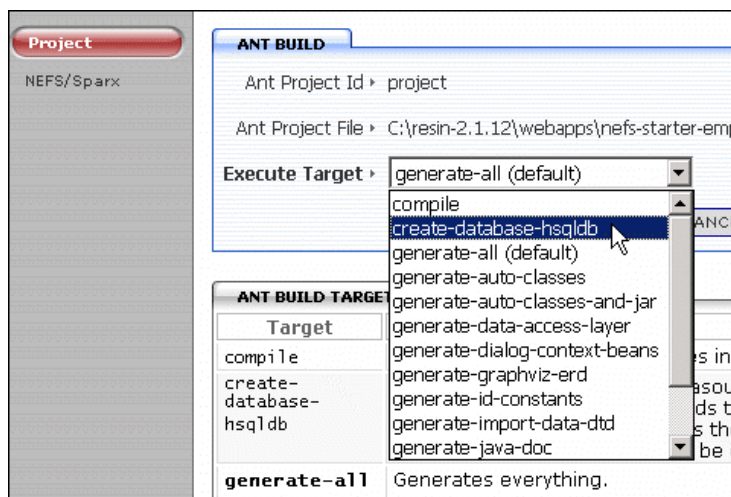
The DDL representation of your schema consists of the actual commands that you need to issue to a database to create the tables you specified in the schema and to populate them with any static data (such as the one stored in enumeration tables) if necessary. These commands are DBMS-specific.

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 9.3.3, "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-INF
[sparx] ansi DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-INF
[sparx] mssql DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-INF
[sparx] postgres DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-INF
[sparx] oracle DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-INF
[sparx] hsqldb DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-INF

create-database-hsqldb:
[echo] Hypersonic database name is 'db' and will be stored in C:\resin-2.1.12
[sparx] hsqldb DDL file C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-INF
[sql] Executing file: C:\resin-2.1.12\webapps\nefs-starter-empty\WEB-INF
[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. Automating the HSQL Database Generation

You may also edit the web.xml file to automatically create the database each time the servlet is run for the first time. to do this, you need to add a --init-first-time-using-ant servlet option in the Sparx navigation controller using the following steps:

1. Open the APP_ROOT/WEB-INF/web.xml file and look for the following Code:

```

<servlet>
<servlet-name>SparxNavigationController</servlet-name>
<servlet-class>
com.netspective.sparx.navigate.NavigationControllerServlet
</servlet-class>
</servlet>

```

2. Add the init-param tag to the above block of code. The resulting code will look like this:

```

<servlet>
<servlet-name>SparxNavigationController</servlet-name>
<servlet-class>
com.netspective.sparx.navigate.NavigationControllerServlet
</servlet-class>
<init-param>
<param-name>com.netspective.sparx.navigate.CONTROLLER_SERVLET_OPTIONS</param-name>
<param-value>
--init-first-time-using-ant=/WEB-INF/sparx/conf/ant-build-project.xml:create-
database-hsqldb
</param-value>
</init-param>
</servlet>

```

The above approach works because when the navigation controller runs, it creates a file called APP_ROOT/WEB-INF/sparx/conf/execution.properties. This properties file contains the execution count of the servlet. If the execution count is less than one, the Ant file (ant-build-project.xml) and the target you requested (create-database-hsqldb) will be run.

Note

This is the best method to create databases because it automatically creates the database the very first time the servlet is run. And, if you ever want the database recreated, you can simply erase the

execution.properties file.

9.3.3. Populating the HSQL Database with Test Data

You will need some test data to be stored in the Books 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 4. Loading Initial Test Data into Book_Info Table

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

<dal> ❷
  <book-info ❸
    id="test-001" name="Book 001"
    author="Smith, James R." genre="Science Fiction" isbn="test001"/>

    <book-info
      id="test-002" name="Book 002"
      author="Jones, Rober. J." genre="Mystery" isbn="test002"/>
  </dal>
```

- ❶ 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.
- ❷ The root tag for the initial-and-test-data.xml is dal.
- ❸ The root dal tag may contain one or more tags named as *table-name* (book-info in this case). Note that the underscore "_" in the original table name "book_info" (specified by schema tag in the project file) is replaced with a hyphen "-". The table field names become the attributes for the table-name tag (id, name, author, genre and isbn in this case). The values you provide for these fields are stored in the corresponding hsql table.

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

Example 5. 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-book-info-records"> ❹
    <xsl:call-template name="iterate-one">
      <xsl:with-param name="x"><xsl:value-of select="@count"/></xsl:with-param>
    </xsl:call-template>
  </xsl:template>

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

    <book-info> ❻
      <xsl:attribute name="id">BOOK_<xsl:value-of select="$x"/></xsl:attribute>
      <xsl:attribute name="name">Book <xsl:value-of select="$x"/></xsl:attribute>
      <xsl:attribute name="author">Author <xsl:value-of select="$x"/></xsl:attribute>
      <xsl:attribute name="genre">Science Fiction</xsl:attribute>
      <xsl:attribute name="isbn">ISBN <xsl:value-of select="$x"/></xsl:attribute>
    </book-info>

    <xsl:if test="$x > 1"> ❼
      <xsl:call-template name="iterate-one">
        <xsl:with-param name="x" select="$x - 1"/>
      </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-info-records template. It generates the values for all the fields of Book_Info table.
- ❻ The book-info 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 greater than 1, decreases it by one to call the iterate-one template with this new value.

This XSL can be incorporated in your XML file using the following code:

Example 6. Incorporating XSL in the Test Data XSL

```

<?transform --xslt initial-and-test-data.xslt?> ❶
<!DOCTYPE dal SYSTEM "../defn/db-import.dtd">
<dal>
  <generate-book-info-records count="100"/> ❷
</dal>

```

- ❶ The xdm-transform processing instruction tells Sparx to filter special tags through the XSLT before processing.
- ❷ Calling the the template from XSL to generate 100 test records for Book_Info table.

Note

The XSL is optional. It is merely a way of automating the creation of lots of test data. You can manually create the XML and leave out the XSL.

9.3.4. DDL and HSQL Database Files

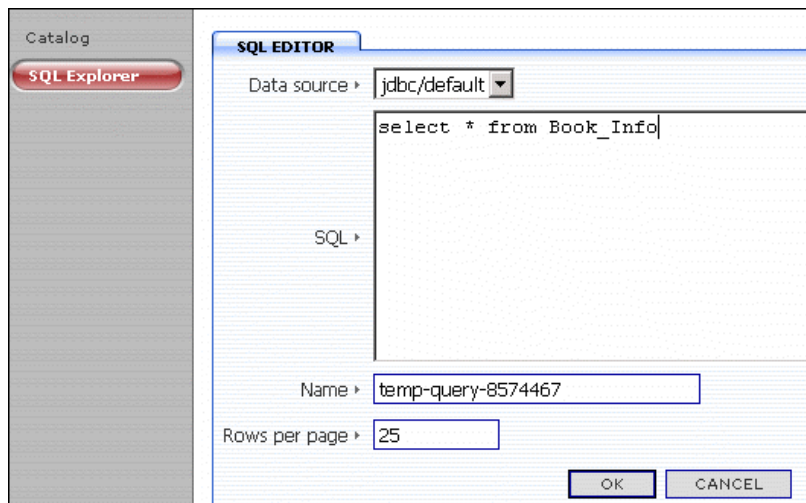
The DDL generation process creates the WEB-INF/database folder. This folder contains further sub-folders: defn and instance.

defn This folder contains the SQL files containing DBMS-specific database creation scripts.

instance This folder contains the HSQL database files: db.properties, db.script, db.data and db.backup.

9.3.5. 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 Book_Info table in the SQL Editor.



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

Id	Name	Author	Genre	Isbn
BOOK_1	Book 1	Author 1	0	ISBN 1
BOOK_10	Book 10	Author 10	0	ISBN 10
BOOK_100	Book 100	Author 100	0	ISBN 100
BOOK_11	Book 11	Author 11	0	ISBN 11
BOOK_12	Book 12	Author 12	0	ISBN 12
BOOK_13	Book 13	Author 13	0	ISBN 13
BOOK_14	Book 14	Author 14	0	ISBN 14
BOOK_15	Book 15	Author 15	0	ISBN 15
BOOK_16	Book 16	Author 16	0	ISBN 16
BOOK_17	Book 17	Author 17	0	ISBN 17
BOOK_18	Book 18	Author 18	0	ISBN 18
BOOK_19	Book 19	Author 19	0	ISBN 19
BOOK_2	Book 2	Author 2	0	ISBN 2

9.4. Creating the Data Management Layer

The Books Application uses two types of queries for data management:

Static SQL

A static SQL statement is merely an encapsulation of a regular SQL statement within the XML definition required by Sparx to interpret it. Once defined, a single or multiple SQL statements may be used in reports, dialogs (forms), Servlets, templates, or JSP-pages. All the static SQL statements with the bind parameters are declared in the Books Application Project File.

Dynamic SQL (Query-Defn) One of Axiom's powerful features is to generate dynamic SQL statements based upon user input. The dynamic SQL generator can also generate different output formats such as pageable HTML reports with configurable rows per page or comma separated value (CSV) reports. The main component of the dynamic queries is the query definition dialog which is used to define bind parameters of the dynamic SQL statement. Thus, the dynamic query generator can be thought of as a combination of Dialog and Static Query features.

Note

All the static and dynamic SQL statements are declared in the Books Application Project File (project.xml)

9.4.1. Declaring a Static Query

The Books Application uses a simple join query to get information about all the books and their genres. Following is the xml declaration that performs this task:

Example 7. Declaring a Static Query

```
<queries package="books"> ❶
  <query name="get-all-books"> ❷
    select id, name, author, book type.caption, isbn ❸
      from book_info, book_type
      where genre = book type.id
  </query>
</queries>
```

- ❶ All the static in Axiom must belong to a statement package represented by queries XML tag. The statement package is identified by its name (books in the above example). You can define multiple packages within your application's project.xml file.
- ❷ A static query is defined (with or without bind parameters) by using the query tag. Each query is identified by its name (get-all-books in the above example).

Note

The namespace of a query is always *queryPkg.queryName*. This means you can have more than one queries with same name if each of them belongs to a different query package.

- 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 Book_Type and Book_Info tables using the genre (from Book_Info) and id (from Book_Type) fields.

9.4.2. Presenting the Results of a Static Query

The next step is to define the customized way of displaying the result of your get-all-books query.

Example 8. Presenting the Results of a Static Query

```
<queries package="books">
  <query name="get-all-books">
    select id, name, author, book type.caption, isbn
      from book_info, book_type
      where genre = book type.id

  <presentation> ❶
    <panel name="main" height="300"> ❷
      <frame heading="static:All available books"/> ❸
    </panel> ❹
  </presentation>
```

```

<actions> ❸
  <action type="add" caption="Add Book" redirect="page-id:/add"/> ❹
  <action type="edit" redirect="page-id:/edit?id=${0}"/> ❺
  <action type="delete" redirect="page-id:/delete?id=${0}"/> ❻
</actions>

<column heading="ID" format="plain" redirect="page-id:/edit?id=${0}"/> ❼
<column heading="Name" />

  <column heading="Author"/>
  <column heading="Genre"/>
  <column heading="ISBN"/>
</report>
</panel>
</presentation>
</query>
</queries>

```

- ❶ Each query tag contains a `presentation` tag associated with it. This tag defines presentation aspects of the query.
- ❷ The `panel` tag is used to display a panel of given height. This panel serves as a container for the displayed query result.
- ❸ The `frame` tag defines a frame to be drawn around the query result. The `heading` attribute is assigned a value using static value source.
- ❹ A `report` is a way to customize the output of a SQL statement on the page.

Note

Each SQL statement may have any number of `report` tags as necessary. That way, a single statement can be viewed differently depending upon what context it's being called in.

- ❺ You can declare a set of actions for your query's result. This is done using the `actions` tag. 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.
- ❻ 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.
- ❼ Declaring `edit` action. You may also supply parameters and their values within the redirect URL (selected record's `id` field in this case).

Any `${XXX}` specifies a dynamic replacement. Usually, value sources are used within the brackets but when the dynamic replacement definition is used within a report definition (declared using `report` tag), you can insert a number within the brackets to indicate the column index of the report. In the above example, `${0}` is used to indicate that it should be replaced with the value of the first column of current row.

- ❽ Declaring the `delete` action with `redirect` attribute pointing to the `delete` page.
- ❾ 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).

9.4.3. 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.

AVAILABLE STATIC QUERIES								
Query	Params	Executed	Avg	Max	Conn	Bind	SQL	Fail
books								
get-all-books								

Click on the get-all-books query to see the SQL statement.

QUERY SQL TEXTS

DBMS	SQL Text
ansi	<pre>select id, name, author, book_type.caption, isbn from book_info, book_type where genre = book_type.id</pre>

QUERY PARAMETERS

Index	Name	JDBC Type	Java Type	Value Source	Value Source Class
Query has no parameters.					

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):

Static SQL: books.get-all-books

Static Queries | Dynamic Queries | Schemas | Data Sources | Database Policies | Connection Profiles

Catalog
Documentation
Unit Test

ALL AVAILABLE BOOKS

ID	Name	Author	Genre	ISBN
BOOK_1	Book 1	Author 1	Science Fiction	ISBN 1
BOOK_10	Book 10	Author 10	Science Fiction	ISBN 10
BOOK_100	Book 100	Author 100	Science Fiction	ISBN 100
BOOK_11	Book 11	Author 11	Science Fiction	ISBN 11
BOOK_12	Book 12	Author 12	Science Fiction	ISBN 12
BOOK_13	Book 13	Author 13	Science Fiction	ISBN 13
BOOK_14	Book 14	Author 14	Science Fiction	ISBN 14
BOOK_15	Book 15	Author 15	Science Fiction	ISBN 15
BOOK_16	Book 16	Author 16	Science Fiction	ISBN 16
BOOK_17	Book 17	Author 17	Science Fiction	ISBN 17

Page 1 of 11 [NEXT](#) [LAST](#) 103 total rows [DONE](#)

9.4.4. Declaring a Dynamic Query (Query Definition)

Query definitions are one of the most powerful features that Sparx provides developers. Using query definitions a developer wields extreme flexibility and power with an ease rivaled by few, if any, other components of Sparx.

The Books Application will make use of the following query definition:

Example 9. Declaring a Dynamic Query (query Definition)

```
<query-defn name="books"> ❶
  <field name="book_id" caption="Book ID" join="BookInfo" column="id"/> ❷
  <field name="name" caption="Name" join="BookInfo" column="name"/>
  <field name="author" caption="Author" join="BookInfo" column="author"/>
  <field name="genre_id" caption="Genre ID" join="BookInfo" column="genre"/> ❸
  <field name="genre_caption" caption="Genre" join="BookType" column="caption"/>
  <field name="isbn" caption="ISBN" join="BookInfo" column="isbn"/>
```

```

<join name="BookInfo" table="book info"/>
<join name="BookType" table="book type" condition="BookType.id = BookInfo.genre"/> ❷
</query-defn>

```

- ❶ All query definitions are declared using the `query-defn` tag and are uniquely identified with a name (books in this case).
- ❷ Each query definition contains one or more `field` tags that declare the selectable fields. The `field` tags are specified so that Axiom knows what fields the user should be allowed to select. Query definition fields may come from a mix of tables so long as they have appropriately defined `join` attribute values. The value of the `join` attribute is a reference to a `join` tag later on in the query definition (`BookInfo` and `bookType` in this case).

Note

In the context of the final SQL statement that Sparx generates, the `field` tags become a part of the `select` clause, i.e. the part of a `select` statement that determines which fields need to be returned.

- ❸ The `column` attribute of `field` tag contains the field name (name in this case) as defined in the schema table (`Book_Info` in this case). The `name` attribute is used to define a different name for the schema table field (`genre_id` and `genre`, respectively, in this case).
- ❹ The `join` tag is used to let the query definition know of the list of tables and joins. This is necessary to be able to get all the fields that are a part of the query definition. Each `join` tag is uniquely identified through its name. The `table` attribute contains the schema table name whereas the `condition` attribute defines the join condition.

Note

In the context of the final SQL statement that Sparx generates, the `join` tags become a part of the `where` clause to signify the relationships between tables (if any exists).

9.4.5. Presenting the Results of a Dynamic Query (Query Definition)

After defining the basic join on Books Application tables, the next step is to define the search dialog and the presentation of the search result. This is handled through the use of `presentation` tag and its child tags under `query-defn` tag.

Example 10. Declaring the Search Dialog and Query Result Presentation

```

<query-defn name="books">
  <presentation> ❶
    <select-dialog name="searchBooksDialog" allow-debug="yes" ❷
      hide-output-dests="no" hide-readonly-hints="yes" >

      <frame heading="Search for Books" /> ❸
      <field type="text" name="book_id" caption="Book ID" />
      <field type="text" name="name" caption="Book Name"/>
      <field type="text" name="author" caption="Author"/>
      <field type="text" name="isbn" caption="ISBN"/>
      <field type="select" style="list" name="genre"
        caption="Genre" choices="schema-enum:Book Type"/> ❹

    <select name="test"> ❺
      <display field="book_id"/>
      <display field="name"/>
      <display field="author"/>
      <display field="genre_caption"/> ❻
    </select>
  </presentation>
</query-defn>

```



```

        <display field="isbn"/>

        <condition field="book_id" allow-null="no" comparison="equals" ❷
            value="field:book id" connector="and"/>
        <condition field="name" allow-null="no" comparison="starts-with"
            value="field:name" connector="and"/>
        <condition field="author" allow-null="no" comparison="equals"
            value="field:author" connector="and"/>
        <condition field="genre_id" allow-null="no" comparison="equals"
            value="field:genre" connector="and"/>

        <presentation> ❸
            <panel name="report">
                <frame heading="static:Book Search Results"/>
                <report> ❹
                    <column heading="ID" format="plain" redirect="page-id:/edit?id=${0}"/> ❺
                    <column heading="Name" />
                    <column heading="Author"/>
                    <column heading="Genre"/>
                    <column heading="ISBN"/>
                </report>
            </panel>
        </presentation>
    </select>
</select-dialog>
</presentation>
</query-defn>

```

- ❶ The search dialog and search field results are declared under a `presentation` tag in the `query-defn` hierarchy.
- ❷ The `select-dialog` tag is essentially a dialog embedded inside a query definition. The purpose of this dialog is to provide a user interface for the query definition. Each `select-dialog` is uniquely identified through its name.

Note

One query definition can have multiple `select-dialogs` declarations.

The `allow-debug` attribute of `select-dialog` tag is used to specify the query debug option. If it is set to "yes" (as in this example), a debug check box is displayed (with the caption *View Generated SQL*). When this checkbox is checked and the dialog is submitted, the generated SQL with its bind parameters is displayed for debugging purposes instead of execution of the generated SQL.

The `hide-output-dests` attribute is used to give the user several output options: HTML output (pageable report and row count per page), CSV output format, tab delimited format. This is done by setting the value of this attribute to "no". The `hide-readonly-hints` attribute is used to show or hide the field hints (as is the case in this example).

- ❸ The `frame` tag is used to draw a frame, with appropriate heading, around the search dialog.
- ❹ The first part of a `select-dialog` is a declaration of all the fields (and their corresponding UI representations) that the `select-dialog` will use for user input. Each field has associated name and caption attributes. The `genre` field is declared to be a "select". The choices for this field are populated with the values from `Book_Type` table using `schema-enum` value source.
- ❺ The second part of a `select-dialog` is a `select` section which has two parts in itself. The `select` component is what determines which of the `select-dialog`'s declared fields are actually displayed in the dialog for this `select-dialog`. It also determines how each field will be interpreted by the query definition engine once the dialog is submitted. The `select` tag is uniquely identified through its name (`test` in this case).
- ❻ The `display` tags have one attribute, `field`, which points to the name of a field declared in the main query definition.
- ❼ The `condition` tags determine how the data input from the `select-dialog` will be interpreted by the query definition engine. Basically, it defines the search criteria.

Each condition tag represents a condition on a field specified by the `field` attribute. The `allow-null` attribute of the `condition` tag determines the behavior of the query definition engine if this field is left empty when the dialog is submitted. The value of this attribute is set to “no” in this case. That is why the select generated will omit the field if the corresponding dialog field happens to be empty. The `comparison` attribute of the `condition` tag describes the relational operator to use (“*equals*” in this case).

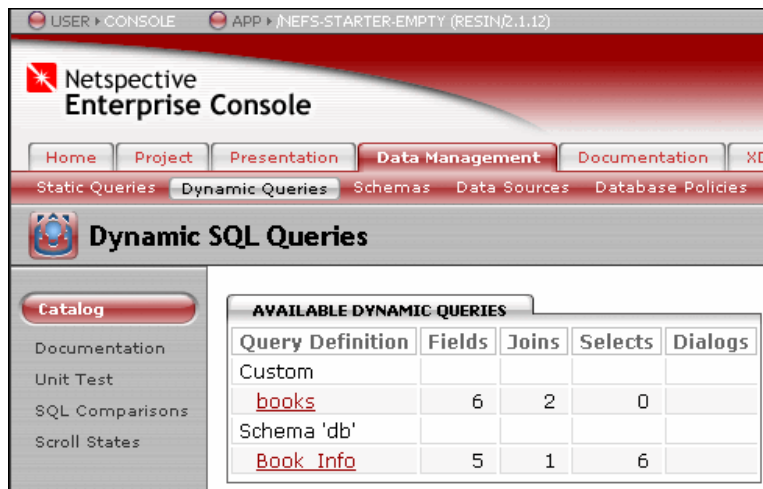
The `value` attribute of the `condition` tag describes the field from the main query definition (`book_id` in this case) with which this `select-dialog` field (`book_id` in this case) is to be compared. The `select-dialog` field is mentioned using the `field` value source.

The `connector` attribute determines how many field criteria each record in the database has to match before it is selected. In this example, all the fields have a connector value of “and” thus specifying that a record would have to match all the field criteria to be selected in the final result.

- 8 The result of this dynamically generated and executed query can be configured in different ways using presentation tag hierarchy.
- 9 The report tag customizes the displayed query result fields.
- 10 Each report tag contains column tags to customize the display of individual query result fields. You may also provide a redirect URL which a column through the `redirect` attribute. In this example, the ID field of the book record in the displayed result points to the `Edit Book` page. Clicking on this link will open the `Edit Book` page with the dialog fields populated with the selected book's information.

9.4.6. Unit Testing a Query Definition

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



Click on the `books` query to see the details of the query in a tabular form.

FIELDS				
Alias	Caption	Join	Column	Column Expr
book_id	Book ID	BookInfo	id	BookInfo.id
name	Name	BookInfo	name	BookInfo.name
author	Author	BookInfo	author	BookInfo.author
genre_id	Genre ID	BookInfo	genre	BookInfo.genre
genre_caption	Genre	BookType	caption	BookType.caption
isbn	ISBN	BookInfo	isbn	BookInfo.isbn

JOINS				
Alias	Table	Condition	Auto-inc	Im
BookInfo	book_info		No	
BookType	book_type	BookType.id = BookInfo.genre	No	

SELECTS		
Name	SQL Static	Distinct Rows
Query definition has no selects.		

Click on the Unit Test option in the left menu bar. [TODO: Documentation to be made available when the Dynamic Query Unit Test option is functional]

10. Creating the Presentation Layer

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

- Home Page
- Add Books Page
- Edit Books Page
- Delete Books Page
- Search Books Page
- Console Page
- NEFS Sample Apps Home Page

10.1. Creating the Home Page

Your Books Application Home page will display list of all the books. It will also provide Edit and Delete options for each of the displayed books.

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 Books Application navigation:

Example 11. Creating "Home" Page

```
<project xmlns:xdm="http://www.netspective.org/Framework/Commons/XMLDataModel">
...
  <navigation-tree name="app" default="yes"> ❶
    <page name="home" default="yes" caption="Home" ❷
      command="query,books.get-all-books,-,-,record-editor-compressed"/> ❸
    </page>
  </navigation-tree>
...
```

- ❶ The `navigation-tree` tag starts out the definition and appears as a child of the `project` tag. 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 servlet parameter or chosen dynamically at runtime based on some processing rules. There is only one instance of each navigation tree that you define and the definition is shared across all users and threads.

Note

You may declare as many navigation trees as your application needs.

- ❷ The `page` tag begins a definition of a single page and appears under the `navigation-tree` tag. Each page has a name (`home` in this case). The name attribute is required and must be unique within the navigation tree in which it is defined. A tree may be marked with `default=yes` if it is to be the default tree. 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. There is only one instance of each page that you define and the definition is shared across all users and threads.

Note

You may declare as many pages under a navigation-tree as your application needs.

- ❸ The `page` 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-books` from the query package named `books` (see Section 9.4.1, “Declaring a Static Query”). It also specifies one of the pre-defined report skins (`record-editor-compressed` in this case) for the displayed report.

10.2. Creating the Add Book Page

The next step is to create a page for adding books information to your database. This is achieved through the following XML declaration:

Example 12. Creating "Add Book" Page

```
<navigation-tree name="app" default="yes">
  <page name="add" caption="Add Book" command="dialog,schema.db.Book_Info,add"/> ❶
</navigation-tree>
```

- ❶ The `page` tag in the example code defines Books Application's `Add Book` page. The `command` attribute in this example displays and executes a dialog box for `Book_Info` table using the `"add"` perspective.

10.3. Creating the Edit Book Page

Your books application requires a way to edit the book information already stored in the database. For that, you will need an `Edit Book` page. This is achieved through the following XML declaration::

Example 13. Creating "Edit Book" Page

```
<navigation-tree name="app" default="yes">
  <page name="edit" caption="Edit Book" command="dialog,schema.db.Book_Info,edit" ❶
    require-request-param="id" ❷
    retain-params="id"> ❸
    <missing-params-body> ❹
      Please choose a book to edit from the &lt;a href='home'&gt; books list&lt;/a&gt;.
    </missing-params-body>
  </page>
</navigation-tree>
```

- ❶ The `page` tag declares the `Edit Book` page. The `command` attribute displays and execute a dialog box for `Book_Info` table using the `"edit"` perspective.

- ❷ The `require-request-parameter` attribute specifies `id` as the required request parameter. The `id` parameter contains unique identifier for the book record that is being edited.
- ❸ The `retain-params` attribute allows you to carry parameters from one page to another. In this example, the `"id"` parameter's value will be sent to another page (tab) when that tab/page is clicked.
- ❹ The `missing-params-body` tag is used to optionally produce automatic error message when required request parameter (`'id'` in this case) is not provided.

10.4. Creating the Delete Book Page

The Delete Book page is declared through the following XML:

Example 14. Creating "Delete Book" Page

```
<navigation-tree name="app" default="yes">
  <page name="delete" caption="Delete Book"
        command="dialog,schema.db.Book Info,delete" ❶
        require-request-param="id" ❷
        retain-params="id"> ❸
    <missing-params-body> ❹
      Please choose a book to delete from the <a href='home'> books
list</a>;.
    </missing-params-body>
  </page>
</navigation-tree>
```

- ❶ The page tag declares the Delete Book page. The `command` attribute displays and execute a dialog box for Book Info table using the `"delete"` perspective.
- ❷ The `require-request-parameter` attribute specifies `id` as the required request parameter. The `id` parameter contains unique identifier for the book record that is being deleted.
- ❸ The `retain-params` attribute allows you to carry parameters from one page to another. In this example, the `"id"` parameter's value will be sent to another page (tab) when that tab/page is clicked.
- ❹ The `missing-params-body` tag is used to optionally produce automatic error message when required request parameter (`'id'` in this case) is not provided.

10.5. Creating the Search Books Page

The Books Application has, at the moment, most of the functionality originally intended for it. The only bit of functionality that is left is that of being able to search for books matching any criteria the user chooses. For a small collection, the result of the `get-all-books` query is small enough to browse manually for the information pertaining to a book. For larger numbers of books, the power of a query definition can come in very handy to search the database for all books matching any criteria specified by the end user.

The following XML declaration defines the Search Books page:

Example 15. Creating "Search Books" Page

```
<navigation-tree name="app" default="yes">
  <page name="search" caption="Search Books"
        command="query-defn, books, test, searchBooksDialog"/> ❶
</navigation-tree>
```

- ❶ The page tag is used to declare the Search Books page. The `command` attribute uses the `"query-defn"` value source to display the search dialog and the query results. The example declaration specifies the `query-defn` named `books`. The select statement being used for creating the search criteria is `test` while the customization of query result is done by the `select-dialog` named `searchBooksDialog`.

10.6. Creating the Console Page

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

Example 16. Linking to the Books Application Console

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

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

10.7. Creating the Sample Apps Home Page

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

Example 17. Creating Sample Apps Home Page for Books Application

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

- ❶ 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.

10.8. Custom-Handling of Dialog's Next Action

You may implement the Sparx DialogNextActionProvider interface to customize what happens after your Books Application's dialogs (forms) are submitted. In this example, you will try to send the dialogs (forms) back to the Books Application Home page. Following is the Java class (Util.java) that provides this functionality. This Java file will be saved in the WEB-INF/classes/app folder along with its class file.

Example 18. Class for Custom-Handling of Dialog's Next Action

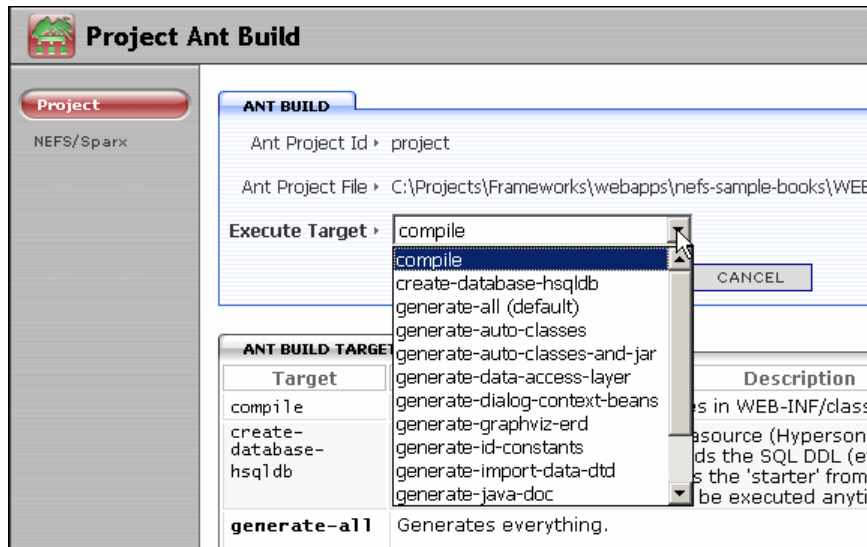
```
package app; ❶

import com.netspective.sparx.form.handler.DialogNextActionProvider; ❷
import com.netspective.sparx.form.DialogContext;

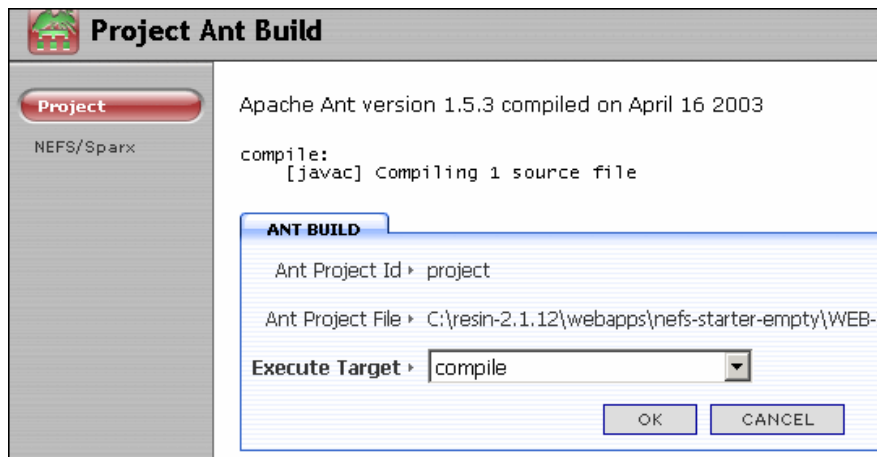
public class Util implements DialogNextActionProvider ❸
{
  public String getDialogNextActionUrl(DialogContext dc, String defaultUrl) ❹
  {
    return dc.getServletRootUrl(); ❺
  }
}
```

- ❶ Defines the package for this class (app in this case)
- ❷ Importing the required classes
- ❸ The class Util Implements the Sparx DialogNextActionProvider interface.
- ❹ Implementing the method that gets the URL for the dialog's next action.
- ❺ Returns the root URL of the current dialog context. The URL in this case will be the one for the Books Application Home page.

To compile the Util.java, go to Project | Ant Build section in Console and execute the Compile target.



The `Util.java` file is compiled and a compile message is displayed.



You can define the custom class, for handling the next action of your dialogs (forms), using the following XML declaration:

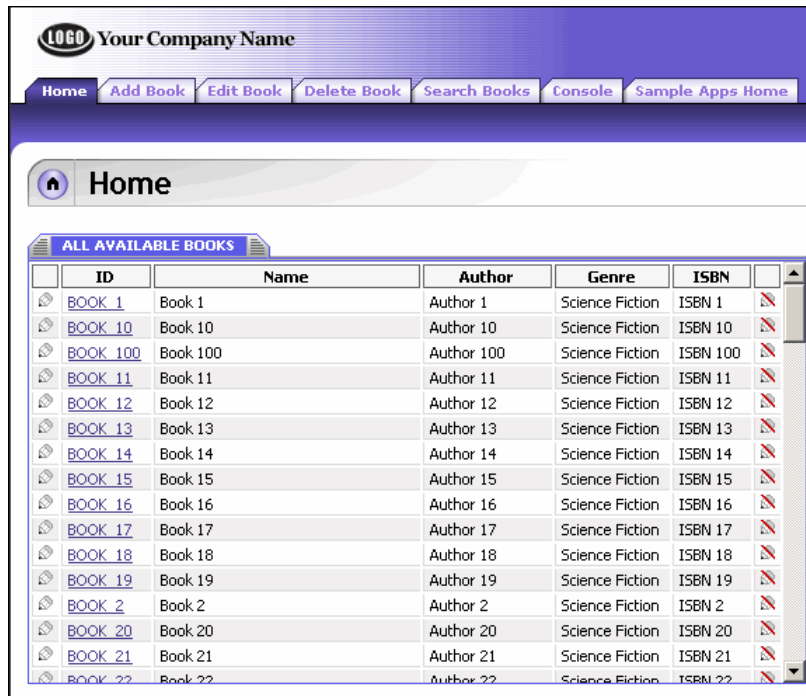
Example 19. Declaring Custom-Class for Dialog's Next Action Provider

```
<navigation-tree>
  <dialog-next-action-provider class="app.Util"/> ❶
</navigation-tree>
```

- ❶ Declaring custom class (`app.Util` in this case) for custom-handling of dialog's next action. The class name is fully qualified.

10.9. Testing the Books Application

Congratulations! The Books Application is complete. Now you should open up your browser window and go to the Book 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-books` as your application's name, the URL for Books Application Home Page becomes: `http://localhost:8080/nefs-sample-books`



ID	Name	Author	Genre	ISBN	
BOOK 1	Book 1	Author 1	Science Fiction	ISBN 1	
BOOK 10	Book 10	Author 10	Science Fiction	ISBN 10	
BOOK 100	Book 100	Author 100	Science Fiction	ISBN 100	
BOOK 11	Book 11	Author 11	Science Fiction	ISBN 11	
BOOK 12	Book 12	Author 12	Science Fiction	ISBN 12	
BOOK 13	Book 13	Author 13	Science Fiction	ISBN 13	
BOOK 14	Book 14	Author 14	Science Fiction	ISBN 14	
BOOK 15	Book 15	Author 15	Science Fiction	ISBN 15	
BOOK 16	Book 16	Author 16	Science Fiction	ISBN 16	
BOOK 17	Book 17	Author 17	Science Fiction	ISBN 17	
BOOK 18	Book 18	Author 18	Science Fiction	ISBN 18	
BOOK 19	Book 19	Author 19	Science Fiction	ISBN 19	
BOOK 2	Book 2	Author 2	Science Fiction	ISBN 2	
BOOK 20	Book 20	Author 20	Science Fiction	ISBN 20	
BOOK 21	Book 21	Author 21	Science Fiction	ISBN 21	
BOOK 22	Book 22	Author 22	Science Fiction	ISBN 22	

The Home page lists all the books available in your Books Application database along with the Edit and Delete buttons. Try adding a new book from the Add Book section.



ADD BOOK INFO

Id: book-155

Name: Soul Music

Author: Terry Pratchett

Genre: Science Fiction

Isbn: 7513-44141

Enter the book information and click the **OK** button. You can see the newly added book in the list available on the Home page.

Book ID	Book Name	Author	Genre	ISBN	Action
BOOK_89	Book 89	Author 89	Science Fiction	ISBN 89	
BOOK_9	Book 9	Author 9	Science Fiction	ISBN 9	
BOOK_90	Book 90	Author 90	Science Fiction	ISBN 90	
BOOK_91	Book 91	Author 91	Science Fiction	ISBN 91	
BOOK_92	Book 92	Author 92	Science Fiction	ISBN 92	
BOOK_93	Book 93	Author 93	Science Fiction	ISBN 93	
BOOK_94	Book 94	Author 94	Science Fiction	ISBN 94	
BOOK_95	Book 95	Author 95	Science Fiction	ISBN 95	
BOOK_96	Book 96	Author 96	Science Fiction	ISBN 96	
BOOK_97	Book 97	Author 97	Science Fiction	ISBN 97	
BOOK_98	Book 98	Author 98	Science Fiction	ISBN 98	

Try editing the record of the book. Select the book from the list by clicking on its name. The Edit Book page is displayed containing the selected book's information.

Edit Book

EDIT BOOK INFO

Id ▶ Book_101

Name ▶

Author ▶

Genre ▶

Isbn ▶

Try searching for books by genre or name, or any other field for that matter, by using the Search Books page.

Search Books

SEARCH FOR BOOKS

Book ID ▶

Book Name ▶

Author ▶

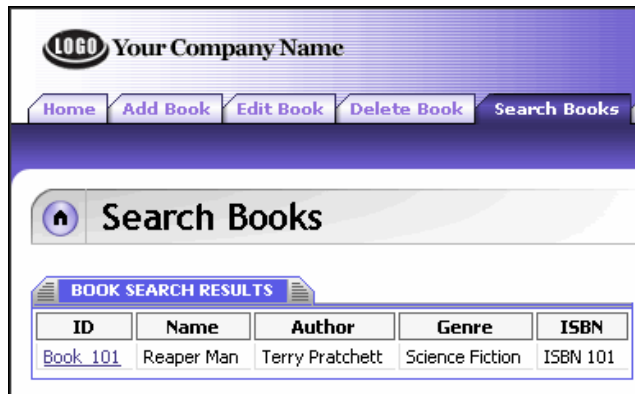
ISBN ▶

Genre ▶

Style Destination

☐ View Generated SQL

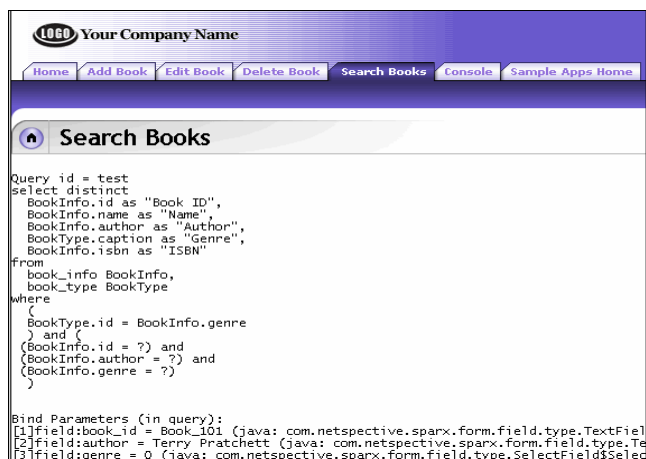
The result of the search is displayed in tabular form.



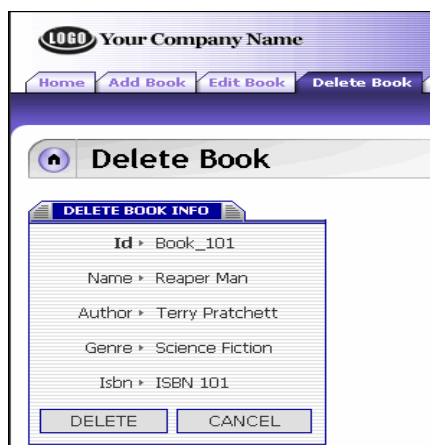
You may see the SQL statement corresponding to a search criteria by clicking on the *View Generated SQL* check box on the Search Books page. The SQL statement is displayed along with the bind parameters and their values.

Note

You may turn this option off in the production application by setting the `allow-debug` attribute of `select-dialog` tag within the corresponding query definition. See Section 9.4.5, “Presenting the Results of a Dynamic Query (Query Definition)”



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



You may access the Books Application Console through the Console page. Use 'console' and 'console'

(without quotes) for Console's User Id and Password.



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



11. Moving to Another Database

Because NEFS Axiom supports multiple databases right out of the box, when you generate the SQL DDL from either Console | Project | Ant Build or from an automated run of the Ant Build file you will have the following files (assuming you have `<schema name="db">` in your `project.xml`):

APP_ROOT/WEB-INF/database/defn

- db-ansi.sql (general ANSI DDL)
- db-erd.dot (GraphViz ERD diagram file)
- db-hsqldb.sql (HSQLDB DDL)
- db-mssql.sql (MS SQL Server DDL)
- db-mysql.sql (MySQL DDL)
- db-oracle.sql (ORACLE DDL)
- db-postgres.sql (PostgreSQL DDL)

Each database SQL file (for DDL and for DML) uses its own *database policy* class. The database policy is responsible for taking the db-independent `<project>`, `<schema>`, `<table>`, `<column>`, etc tags and converting them into database-specific DDL and DML calls.

If you want to manually take a DDL file and construct your database then all you need to do is

1. use the generated SQL file and use your database's SQL import method to create the database.
2. change the <data-source> in the project.xml file's <connection-provider> tag.

If you want to automatically create a database within the Ant file the same way the create-database-hsqldb target works, you should add the following target to the APP_ROOT/WEB-INF/sparx/conf/ant-build-projet.xml file:

```
<target name="create-database-XXXXX" depends=".init" description="...">
  <property name="jdbc.driver" value="XXXX.jdbcDriver"/>
  <property name="jdbc.url" value="jdbc:whatever"/>
  <property name="jdbc.userid" value="xxxxx"/>
  <property name="jdbc.password" value="xxxxx"/>

  <!-- run some initialization script that will create a user/db/etc -->
  <sql src="init-XXXX.sql"
      driver="${jdbc.driver}" url="${jdbc.url}" userid="${jdbc.userid}" password=""/>

  <!-- generate only the your XXXX DDL file -->
  <sparx action="generate-ddl" projectFile="${app.sparx.project.file}"

      destDir="${app.database.defn.path}"
      ddl="/^mssql/" /> <!-- use oracle, mssql, mysql, postres, ansi -->
  <property name="app.database.schema.ddl.loader.file"
      value="${app.database.defn.path}/${app.database.default-schema-name}-mssql.sql"/>

  <!-- load the DDL into the database -->
  <sql src="${app.database.schema.ddl.loader.file}"
      driver="${jdbc.driver}" url="${jdbc.url}" userid="${jdbc.userid}" password=""/>

  <!-- import the "starter" data into the database (assuming you have any) -->
  <sparx action="import-data"
      projectFile="${app.sparx.project.file}"
      import="${app.database.home}/data/initial-and-test-data.xml"
      driver="${jdbc.driver}" url="${jdbc.url}" userid="${jdbc.userid}" password=""/>
</target>
```

12. Conclusion

Your first Sparx application that does something meaningful is complete and you have gone through the shallowest parts of almost everything Sparx has to offer. You can now continue to improve the Books Application with newer ideas, delving deeper into Sparx or you can start a new application and use what you learned here to see how fast you can have a running system.