Table of Contents

Das Ziel dieses Tutorials ist die Erstellung einer Applikation mit dem Enterprise Application Framework. Dazu wird ein erster Einblick in die Möglichkeiten des Frameworks gegeben.

Die Aufgabe der Applikation ist, die Daten aus einer Datenbanktabelle darzustellen und editierbar zu machen. Die Applikation erfordert eine Authentifizierung mit Benutzername und Passwort.

Wir setzen folgende Kenntnisse und Hilfsmittel voraus:

- JVx Binärpaket
- Eclipse IDE (>= 3.4) mit JDT (Empfohlen wird: Eclipse IDE für Java EE Entwickler)
- JDK 8.0 (1.8) or höher
- HSQLDB Bibliothek (http://www.hsqldb.org)
- Datenbank- bzw. SQL Kenntnisse

Diese Dokumentation beschreibt folgende Bereiche:

- JVx Verzeichnisstruktur
- Eclipse Projektkonfiguration
- Applikationsentwicklung
    - Client
    - Server
- Erstellen eines WorkScreens
- Verwenden einer HSQL Datenbank

Verzeichnisstruktur

Für die Applikationsentwicklung mit JVx wird eine spezielle Ordnerstruktur empfohlen. Diese erleichtert den Build Prozess und trennt von vornherein Abhängigkeiten zwischen Client und Server. Diese Struktur ist wie folgt zu erstellen:

Auf Wunsch kann auch eine herkömmliche Strukturierung:

verwendet werden. Die Dokumentation bezieht sich jedoch auf die empfohlene Struktur.

Ordner	Beschreibung
rad	Enthält Applikations- und Serverspezifische Dateien.
apps	Enthält alle verfügbaren Applikationen. In diesem konkreten Beispiel ist nur eine Applikation enthaltene.
firstapp	CEnthält die Applikation mit Projektkonfiguration, Sourcen, Bibliotheken.
help	Enthält den Client für die Online Hilfe und die Hilfeseiten.
libs	Enthält alle Bibliotheken, die sowohl am Client als auch am Server benötigt werden.
libs/client	Enthält alle Bibliotheken die ausschließlich am Client benötigt werden.
libs/server	Enthält alle Bibliotheken die ausschließlich am Server benötigt werden.
src.client	Enthält alle Sourcen die ausschließlich am Client benötigt werden.
src.server	Enthält alle Sourcen die ausschließlich am Server benötigt werden.
test	Enthält Unit tests für Client und Server bzw. Bibliotheken.

Nachdem die Ordnerstruktur erstellt wurde, kopieren Sie die Bibliothek `jvxclient.jar` in den Ordner `libs/client` und die Bibliothek `jvx.jar` in den Ordner `libs/server`. Beide Bibliotheken sind im JVx Binärpaket enthalten.

Projektkonfiguration

Nachdem die Konfiguration durchgeführt wurde, kann mit Eclipse ein neues Projekt erstellt werden:

- **File** / **New** / **Java Project**
- Zu beachten ist, dass das Projekt im Applikationsverzeichnis `firstapp` abgelegt wird.
- **Entfernen** des `src` Ordner von den **Source Folders**
  Setzen der Ordner `src.client`, `src.server` und `test` als **Source Folder**
- **Hinzufügen** der `jvx.jar` Bibliothek, aus dem Projektverzeichnis `JVxFirstApp/libs/server`
- Das Projekt kann nun erstellt werden

Das Projekt wird von Eclipse nun wie folgt dargestellt:

Zur Vollständigkeit kann der src Ordner gelöscht werden. Dieser wird in unserer Applikation nicht benötigt.

Applikationsentwicklung

Die Applikation benötigt Serverseitig eine Konfigurationsdatei für Einstellungen die nur die Applikation betreffen. Für die Konfiguration des Servers wird zusätzlich eine Konfigurationsdatei benötigt. Zuerst erstellen wir die Datei für die Applikation:

- **File** / **New** / **File** - config.xml
  (Erstellung direkt im Applikationsverzeichnis **JVxFirstApp**)

Die Datei wird wie folgt befüllt:

config.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<application>
  <securitymanager>
<class>com.sibvisions.rad.server.security.XmlSecurityManager</class>
    <userfile>users.xml</userfile>
  </securitymanager>

  <!-- predefined life-cycle object names -->
  <lifecycle>
```

```
    <mastersession>apps.firstapp.Session</mastersession>
    <application>apps.firstapp.Application</application>
  </lifecycle>
</application>
```

Parameter	Beschreibung
securitymanager/class	Der Sicherheitsmanager für die Überprüfung von Benutzernamen/Passwort bei der Anmeldung an die Applikation.
securitymanager/usersfile	Die Datei mit den erlaubten Benutzername/Passwort Kombinationen.
lifecycle/mastersession	Die Klassenbezeichnung des Server Objektes, das instanziert wird, wenn der Client eine Anmeldung durchführt bzw. eine neue MasterSession startet.
lifecycle/application	Die Klassenbezeichnung des Server Objektes das instanziert wird beim ersten Zugriff auf die Applikation. Für alle weiteren Zugriffe wird dieses Objekt wiederverwendet.

Die Konfigurationsdatei des Servers muss im Verzeichnis `../JVxFirstApp/rad/server` abgelegt werden.

Dieses Verzeichnis scheint in unserem Eclipse Projekt jedoch nicht auf, da es sich auf einer höheren Verzeichnisebene befindet. Die Konfigurationsdatei könnte direkt im Dateisystem erstellt werden oder wir erstellen einen Verzeichnis-Link in unserem Projekt:

- **File** / **New** / **Folder**

Anschließend kann die Konfigurationsdatei erstellt werden:

- **File** / **New** / **File** - config.xml

Die Datei wird wie folgt befüllt:

config.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<server>
</server>
```

Der Server benötigt für unsere Applikation keine speziellen Parameter.

Für den Client benötigen wir nun eine Klasse die vom Typ `jvx.rad.application.IApplication` ist. Von JVx wird eine Standard Implementierung durch `com.sibvisions.rad.application.Application` implementiert. Von dieser werden wir unseren Client ableiten und erstellen somit eine Klasse, im Verzeichnis `src.client`, mit folgendem Source Code:

FirstApplication.java

```
package apps.firstapp;

import jvx.rad.application.genui.UILauncher;
import jvx.rad.remote.IConnection;

import com.sibvisions.rad.application.Application;
import com.sibvisions.rad.server.DirectServerConnection;

/**
 * First application with JVx, Enterprise Application Framework.
 *
 * @author René Jahn
 */
public class FirstApplication extends Application
{
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // Initialization
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  /**
   * Creates a new instance of <code>FirstApplication</code> with a
technology
   * dependent launcher.
   *
   * @param pLauncher the technology dependent launcher
   * @throws Exception if initialization fails
   */
  public FirstApplication(UILauncher pLauncher) throws Exception
  {
    super(pLauncher);
  }

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // Overwritten methods
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  /**
   * {@inheritDoc}
   */
  @Override
  protected IConnection createConnection() throws Exception
  {
    return new DirectServerConnection();
  }

  /**
   * {@inheritDoc}
   */
  @Override
  protected String getApplicationName()
  {
```

```
    return "firstapp";
  }

}  // FirstApplication
```

Methode	Beschreibung
Konstruktor	Der Standardkonstruktor kann nicht verwendet werden, da jede Applikation mit einem Technologieabhängigen Launcher gestartet wird. Dieser Launcher wird bereits im Konstruktor an die Applikation übergeben.
createConnection	Das Kommunikationsprotokoll wird initialisiert. Für unsere Applikation ist eine `DirectServerConnection` ausreichend, da sowohl Client als auch Server in der selben VM gestartet werden. Wird jedoch ein Applikationsserver eingesetzt könnte alternativ eine `HttpConnection` verwendet werden.
getApplicationName	Legt den Applikationsnamen fest. Dieser Name wird für die Kommunikation mit dem Server benötigt, da dieser abhängig vom Applikationsnamen die passende Applikationskonfiguration verwendet. In unserem Fall muss der Applikationsname `firstapp` lauten, da das Arbeitsverzeichnis `../JVxFirstApp/rad/`**`firstapp`**`/` ebenso lautet. Der Applikationsname MUSS immer dem Verzeichnisnamen entsprechen!

Nun ist es an der Zeit für den ersten Start der Applikation. Dafür erstellen wir eine Runtime Konfiguration:

- **Run** / **Run Configurations...** / **Application - New launch configuration** - mit den Einstellungen:

Parameter	Beschreibung
Main class	Hier wird der Technologie Abhängige Launcher festgelegt. Wir verwenden für unsere Applikation die Technologie Swing und starten eine Swing Applikation.
Program arguments	Dem Launcher muss mitgeteilt werden, welche Applikation gestartet wird. Für unsere Swing Applikation können wir dafür den Mechanismus der Programm Argumente nutzen und übergeben den Klassennamen unserer Applikation.

Die Applikation kann nun gestartet werden und sollte wie folgt aussehen:

Der erste Anmeldeversuch scheitert mit dem Hinweis:

```
Userfile 'users.xml' does not exist!
```

Diese Datei wurde im `config.xml` der Applikation definiert, bisher jedoch noch nicht erstellt. Das holen wir an dieser Stelle nach:

- **File** / **New** / **File** - users.xml

Die Datei befüllen wir mit:

users.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<users>
  <user name="admin" password="admin"/>
</users>
```

Es können beliebig viele `user` Zeilen eingetragen werden!

Nun ist die Anmeldung an die Applikation ohne Probleme möglich. Zur Erfüllung unserer Aufgabenstellung fehlt jedoch noch die Möglichkeit eine Datenbanktabelle anzuzeigen bzw. zu editieren. Diesem Teil der Aufgabe widmen wir uns jetzt.

Erstellen eines WorkScreens

Bevor wir den WorkScreen erstellen, bereiten wir die Applikation für die Anzeige des WorkScreens vor. Dazu erweitern wir unsere `FirstApplication` Klasse wie folgt:

FirstApplication.java

```
package apps.firstapp;

import jvx.rad.application.genui.UILauncher;
import jvx.rad.genui.UIImage;
import jvx.rad.genui.component.UIButton;
import jvx.rad.genui.container.UIToolBar;
import jvx.rad.genui.menu.UIMenu;
import jvx.rad.genui.menu.UIMenuItem;
import jvx.rad.remote.IConnection;

import com.sibvisions.rad.application.Application;
import com.sibvisions.rad.server.DirectServerConnection;

/**
 * First application with JVx, Enterprise Application Framework.
 *
 * @author René Jahn
 */
public class FirstApplication extends Application
{
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // Initialization
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  /**
   * Creates a new instance of <code>FirstApplication</code> with a
technology
   * dependent launcher.
   *
```

```
   * @param pLauncher the technology dependent launcher
   */
  public FirstApplication(UILauncher pLauncher)
  {
    super(pLauncher);
  }

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // Overwritten methods
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  /**
   * {@inheritDoc}
   */
  @Override
  protected IConnection createConnection() throws Exception
  {
    return new DirectServerConnection();
  }

  /**
   * {@inheritDoc}
   */
  @Override
  protected String getApplicationName()
  {
    return "firstapp";
  }

  /**
  * {@inheritDoc}
  */
  @Override
  protected void afterLogin()
  {
    super.afterLogin();

    //configure MenuBar

    UIMenu menuMasterData = new UIMenu();
    menuMasterData.setText("Master data");

    UIMenuItem miDBEdit = createMenuItem
                          ("doOpenDBEdit", null, "DB Edit",
                           UIImage.getImage(UIImage.SEARCH_LARGE));

    menuMasterData.add(miDBEdit);

    //insert before Help
    getMenuBar().add(menuMasterData, 1);
```

```
    //configure ToolBar

    UIToolBar tbMasterData = new UIToolBar();

    UIButton butDBEdit = createToolBarButton
                         ("doOpenDBEdit", null, "DB Edit",
                          UIImage.getImage(UIImage.SEARCH_LARGE));

    tbMasterData.add(butDBEdit);

    getLauncher().addToolBar(tbMasterData);
  }

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // Actions
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  /**
   * Opens the edit screen.
   */
  public void doOpenDBEdit()
  {
    //TODO open the workscreen
  }

}  // FirstApplication
```

Methode	Beschreibung
afterLogin	Diese Methode wird von der Superklasse aufgerufen nachdem eine erfolgreiche Anmeldung durchgeführt wurde. Wir verwenden diese Methode um unser Menü und unsere ToolBar zu erweitern. Es ist nicht nötig nach der Abmeldung die Änderungen rückgängig zu machen, da dies von der Superklasse übernommen wird.
doOpenDBEdit	Diese Methode wird aufgefufen wenn das Menü oder der ToolBar Button gedrückt werden.
createMenuItem	Wird von der Superklasse bereitgestellt um Menü Einträge zu erstellen. Der erste Parameter enthält die Bezeichnung der Methode die aufgerufen werden soll wenn der Menü Eintrag gedrückt wird. Der zweite Parameter enthält den Befehl (ActionCommand) der in unserem Fall keine Rolle spielt. Im dritten Parameter ist der Text des Menü Eintrags zu definieren und abschließend wird das Bild für den Eintrag übergeben.
createToolBarButton	Ähnlich wie `createMenuItem` nur wird hierbei ein Button erzeugt, der sich dem Layout der ToolBar anpasst.
UIImage.getImage	Liefert ein vordefiniertes Bild aus der Bild Bibliothek von JVx Wir verwenden zwecks Komfort ein vordefiniertes Bild.

Wir erstellen nun die Client Klasse für unseren WorkScreen:

- **File** / **New** / **Class**
  `src.client`, `apps.firstapp.frames.DBEditFrame`

und verwenden folgenden Source Code:

DBEditFrame.java

```
package apps.firstapp.frames;

import jvx.rad.genui.container.UIGroupPanel;
import jvx.rad.genui.container.UIInternalFrame;
import jvx.rad.genui.control.UITable;
import jvx.rad.genui.layout.UIBorderLayout;
import jvx.rad.remote.AbstractConnection;
import jvx.rad.remote.MasterConnection;

import com.sibvisions.rad.application.Application;
import com.sibvisions.rad.model.remote.RemoteDataBook;
import com.sibvisions.rad.model.remote.RemoteDataSource;

/**
 * A simple database table editor.
 *
 * @author René Jahn
 */
public class DBEditFrame extends UIInternalFrame
{
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // Class members
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  /** the application. */
  private Application application;

  /** the communication connection to the server. */
  private AbstractConnection connection;

  /** the DataSource for fetching table data. */
  private RemoteDataSource dataSource = new RemoteDataSource();

  /** the contacts tabl. */
  private RemoteDataBook rdbContacts = new RemoteDataBook();

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // Initialization
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  /**
   * Creates a new instance of DBEditFrame for a specific application.
   *
   * @param pApp the application
   * @throws Throwable if the remote access fails
```

```
 */
public DBEditFrame(Application pApp) throws Throwable
{
  super(pApp.getDesktopPane());

  application = pApp;

  initializeModel();
  initializeUI();
}

/**
 * Initializes the model.
 *
 * @throws Throwable if the initialization throws an error
 */
private void initializeModel() throws Throwable
{
  //we use a new "session" for the screen
  connection = ((MasterConnection)application.getConnection()).
  createSubConnection("apps.firstapp.frames.DBEdit");
  connection.open();

  //data connection
  dataSource.setConnection(connection);
  dataSource.open();

  //table
  rdbContacts.setDataSource(dataSource);
  rdbContacts.setName("contacts");
  rdbContacts.open();
}

/**
 * Initializes the UI.
 *
 * @throws Exception if the initialization throws an error
 */
private void initializeUI() throws Exception
{
  UIGroupPanel group = new UIGroupPanel();
  group.setText("Available Contacts");

  UITable table = new UITable();
  table.setDataBook(rdbContacts);

  group.setLayout(new UIBorderLayout());
  group.add(table);

  //same behaviour as centered component in BorderLayout
  setLayout(new UIBorderLayout());
```

```
    add(group);

    setTitle("Contacts");
    setSize(new UIDimension(400, 500));
  }

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // Overwritten methods
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  /**
   * Closes the communication connection and disposes the frame.
   */
  @Override
  public void dispose()
  {
    try
    {
      connection.close();
    }
    catch (Throwable th)
    {
      //nothing to be done
    }
    finally
    {
      super.dispose();
    }
  }

} // DBEditFrame
```

Methode	Beschreibung
initializeModel	Instanziert die Objekte für den Zugriff auf den Server bzw. die Daten.
InitializeUI	Layouting des WorkScreen.
dispose	Beendet die Verbindung zum Server für den WorkScreen und schließt den Frame. Die Verbindung müsste nicht explizit geschlossen werden, da dies beim Verwerfen durch den GarbageCollector vollautomatisch passiert. In unserer ersten Applikation ist das aber auch kein Nachteil.
createSubConnection	Wir erstellen eine eigene Verbindung zum Server. Das hat den Vorteil, dass am Server ein eigenes Lifecycle Objekt verwendet wird. Dieses Objekt hält alle Objekte, die vom WorkScreen benötigt werden. Nachdem der WorkScreen geschlossen wird, wird auch der benutzte Speicher wieder freigeben. Weiters kann jede Verbindung spezielle Parameter und Timeouts haben. Das gewünschte Lifecycle Objekt wird mit der Klassenbezeichnung definitert: apps.firstapp.frames.DBEdit. Die Klasse erstellen wir im Anschluß.

Member	Beschreibung
connection	Die Verbindung zum Server, speziell für den WorkScreen. Im Hintergrund wird ein spezielles Kommunikationsprotokoll verwendet. In unserem Fall spiegelt dieses die Klasse `DirectServerConnection` wieder.
dataSource	Die DataSource ist unabhängig vom Kommunikationsprotokoll und kümmert sich um die Übertragung der Daten zwischen Client und Server. Für den Transfer wird die `connection` verwendet.
rdbContacts	Das Model und der Controller für die Datenanzeige. Der Name `contacts` legt fest unter welchen Namen das serverseitige Objekt im Lifecycle Objekt zu finden ist.
table	Die View für die Datenanzeige.

Der WorkScreen ist nun fertig und kann in die Applikation integriert werden. Wir implementieren nun den fehlenden Aufruf:

FirstApplication.java

```
public class FirstApplication extends Application
{
  ...
  ...
  ...

  /**
   * Opens the edit screen.
   *
   * @throws Throwable if the edit frame can not be opened
   */
  public void doOpenDBEdit() throws Throwable
  {
    DBEditFrame frame = new DBEditFrame(this);

    configureFrame(frame);

    frame.setVisible(true);
  }

}  // FirstApplication
```

Methode	Beschreibung
doOpenDBEdit	Die Methode kann ohne Probleme `Throwable` werfen. Sämtliche Applikationsfehler werden vom Applikationsrahmen abgefangen und in einem Informationsdialog angezeigt.
configureFrame	Diese Methode wird von der Superklasse bereitgestellt und sorgt dafür, dass alle Frames einheitlich aussehen. Dazu zählt unter anderem das Menü Icon.

Die Client Implementierung ist nun abgeschlossen. Bevor wir die Applikation verwenden können müssen die fehlenden Server Klassen erstellt werden. Wir erstellen folgende Klassen:

- **File** / **New** / **Class**

`src.server`, `apps.firstapp.Application`

Application.java

```
package apps.firstapp;

import com.sibvisions.rad.server.GenericBean;

/**
 * The LCO for the application.
 *
 * @author René Jahn
 */
public class Application extends GenericBean
{

}  // Application
```

Beschreibung
Die Klasse spiegelt das Lifecycle Objekt für eine Applikation wieder. Pro Applikation existiert genau eine Instanz dieser Klasse. Es können somit Session übergreifende Objekte verwendet werden.

- **File** / **New** / **Class**
  `src.server`, `apps.firstapp.Session`

Session.java

```
package apps.firstapp;

import com.sibvisions.rad.persist.jdbc.DBAccess;
import com.sibvisions.rad.persist.jdbc.IDBAccess

/**
 * The LCO for the session.
 *
 * @author René Jahn
 */
public class Session extends Application
{
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // User-defined methods
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  /**
   * Returns access to the database.
   *
   * @return the database access
   * @throws Exception if a connection error occurs
```

```
   */
  public IDBAccess getDBAccess() throws Exception
  {
    DBAccess dba = (DBAccess)get("dBAccess");

    if (dba == null)
    {
      dba = new HSQLDBAccess();

      dba.setUrl("jdbc:hsqldb:hsql://localhost/firstappdb");
      dba.setUsername("sa");
      dba.setPassword("");
      dba.open();

      put("dBAccess", dba);
    }

    return dba;
  }

}  // Session
```

Beschreibung	
Die Klasse spiegelt das Lifecycle Objekt für eine Session wieder. Eine Session beginnt in unserem Fall mit der Anmeldung an die Applikation und endet mit der Abmeldung. Pro Session existiert genau eine Instanz dieses Objektes. Es können somit Objekte für die Dauer der Anmeldung verwendet werden. Durch die Ableitung von `apps.firstapp.Application` ist es auf einfachste Art und Weise möglich, auch die Applikationsobjekte zu verwenden.	
Methode	**Beschreibung**
getDBAccess	Öffnet eine neue Verbindung zu einer HSQL Datenbank, falls dies nicht bereits geschehen ist. Das Exception Handling wird vom Server übernommen.

- **File** / **New** / **Class**
  `src.server`, `apps.firstapp.frames.DBEdit`

DBEdit.java

```
package apps.firstapp.frames;

import jvx.rad.persist.IStorage;

import com.sibvisions.rad.persist.jdbc.DBStorage;

import apps.firstapp.Session;

/**
 * The LCO for the DBEdit WorkScreen.
```

```
 *
 * @author René Jahn
 */
public class DBEdit extends Session
{
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // User-defined methods
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  /**
   * Returns the contacts storage.
   *
   * @return the contacts storage
   * @throws Exception if the initialization throws an error
   */
  public IStorage getContacts() throws Exception
  {
    DBStorage dbsContacts = (DBStorage)get("contacts");

    if (dbsContacts == null)
    {
      dbsContacts = new DBStorage();
      dbsContacts.setDBAccess(getDBAccess());
      dbsContacts.setFromClause("CONTACTS");
      dbsContacts.setWritebackTable("CONTACTS");
      dbsContacts.open();

      put("contacts", dbsContacts);
    }

    return dbsContacts;
  }

}  // DBEdit
```

Beschreibung	
Die Klasse spiegelt das Lifecycle Objekt für den `DBEditFrame` WorkScreen wieder. Auf die Objekte kann ausschließlich über die SubConnection des WorkScreens zugegriffen werden. Durch die Ableitung von `apps.firstapp.Session` kann auf einfachste Art und Weise auf sämtliche Objekte der `Session` und der `Application` zugegriffen werden.	
Methode	**Beschreibung**
getContacts	Ermöglicht den Zugriff auf die Datenbanktabelle `CONTACTS`. Der Methodenname muss dem Objektnamen des `RemoteDataBook` entsprechen: `contacts ⇒ getContacts`. Das Exception Handling wird vom Server übernommen.

Die Applikation ist jetzt vollständig implementiert und lauffähig. Damit wir nun mit der Applikation arbeiten können benötigen wir die Datenbank inklusive Tabelle `CONTACTS` auf die wir zugreifen wollen. Die Konfiguration von HSQLDB wird in diesem Dokument nicht detailiert beschrieben, da die

Beispiele auf der Projektseite detailiert und ausreichend sind. In nachfolgendem Kapitel finden Sie eine kurze Zusammenfassung der notwendigen Schritte.

Datenbank erstellen

Folgende Schritte sollten durchgeführt werden um eine HSQLDB zu erstellen und zu starten.

- Kopieren Sie den HSQLDB JDBC-Teiber (`hsqldb.jar`) in das Verzeichnis `../JVxFirstApp/libs/server/`
- Fügen Sie den JDBC-Treiber dem CLASSPATH des JVxFirstApp Projektes hinzu
- Erstellen Sie eine Datenbank mit dem Alias `firstappdb` und folgender Tabelle:

```
CREATE TABLE CONTACTS
(
  ID            INTEGER IDENTITY,
  FIRSTNAME     VARCHAR(200) NOT NULL,
  LASTNAME      VARCHAR(200) NOT NULL,
  BIRTHDAY      DATE,
  STREET        VARCHAR(200),
  NR            VARCHAR(200),
  ZIP           VARCHAR(4),
  TOWN          VARCHAR(200)
)
```

- SStarten Sie die Datenbank z.B.:

```
java -cp ../libs/server/hsqldb.jar org.hsqldb.Server -database.0
     file:firstappdb -dbname.0 firstappdb
```

Die erste Applikation

Nachdem die Datenbank gestartet wurde kann die Applikation ebenfalls gestartet werden. Die fertige Applikation sollte nun wie folgt aussehen:

Den Source Code und das Eclipse Projekt finden Sie auch im Download Bereich.