Table of Contents

In den folgenden Absätzen zeigen wir Ihnen wie Sie mit wie wenig Aufwand und Code ihre erste JVx Applikation erstellen können.

Die Aufgabe der Applikation ist, die Daten aus einer Datenbanktabelle darzustellen und bearbeitbar zu machen.

Vor dem Start benötigen Sie folgende Libraries und Werkzeuge:

- JVx Binärpaket
- Eclipse IDE (>= 3.4) mit JDT (Empfohlen wird: Eclipse IDE für Java EE Entwickler)
- JDK 8.0 (1.8) oder höher
- HSQLDB Bibliothek (http://www.hsqldb.org)
- JVx Beispiel Eclipse Projekt

Für unsere JVx Beispiel Applikation benötigen wir folgende Teile:

- Applikation
- Workscreen
- Business Objekt
- Datenbankverbindung

Erstellen einer Applikation

Für den Client benötigen wir eine Applikation als Rahmen. Jede Applikation muss das Interface `jvx.rad.application.IApplication` implementieren. Wir leiten uns in unserem Beispiel von der Standard Implementierung `com.sibvisions.rad.application.Application` ab, wobei wir folgenden Code verwenden:

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
   * @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";
  }

  /**
  * {@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
                         ("doOpenFrame", null, "DB Edit",
                          UIImage.getImage(UIImage.SEARCH_LARGE));

    tbMasterData.add(butDBEdit);

    getLauncher().addToolBar(tbMasterData);
  }

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

  /**
   * 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
Konstruktor	Es wird ein eigener Konstruktor benötigt, da jede Applikation mit einem Technologieabhängigen Launcher gestartet wird. Dieser Launcher wird bereits im Konstruktor an die Applikation übergeben.

Methode	Beschreibung
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..
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.
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.
doOpenDBEdit	Diese Methode wird aufgerufen wenn das Menü oder der ToolBar Button gedrückt werden und ruft den entsprechenden Workscreen auf.
configureFrame	Diese Methode wird von der Superklasse bereitgestellt und sorgt dafür, dass alle Frames einheitlich aussehen. Dazu zählt unter anderem das Menü Icon.

Erstellen eines Workscreens

Nachdem wir den Applikationsrahmen fertig erstellt haben, erstellen wir nun unseren ersten Workscreen mit folgenden 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));
  }

} // DBEditFrame
```

Methode	Beschreibung
initializeModel	Instanziert die Client Objekte für den Zugriff auf den Server bzw. die Daten.
InitializeUI	Layouting des WorkScreen.

Methode	Beschreibung
createSubConnection	Wir erstellen eine eigene Verbindung zum Server. Das hat den Vorteil, dass am Server ein eigenes Business 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 Business Objekt wird mit der Klassenbezeichnung definiert: `apps.firstapp.frames.DBEdit`. Die Klasse erstellen wir im Anschluß.

Member	Beschreibung
connection	TDie 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 die Datenquelle 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 Business Objekt zu finden ist.

Erstellen des Business Objekts

Nachdem wir den Client fertig erstellt haben, benötigen wir am Server noch das dazugehörige Business Objekt, um die Quelle der Daten näher zu definieren. Dazu verwenden wir folgenden Code:

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 Business 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.

Erstellen der Datenbankverbindung

Im Business Objekt haben wir mit der Methode getDBAccess() auf die Datenquelle verwiesen. In unserem Fall verwenden wir eine HyperSQL Datenbank. In folgender Klasse definieren wir die Datenbankverbindung:

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
```

Methode	Beschreibung
getDBAccess	Öffnet eine neue Verbindung zu einer HSQL Datenbank, falls dies nicht bereits geschehen ist. 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. Diese ist im Eclipse Projekt bereits erstellt worden und kann mit `../JVxFirstApp/db/startHSqlDB.bat` gestartet werden.

Die erste JVx Applikation

Nachdem die Datenbank gestartet wurde, kann die Applikation über das Run Menü von Eclipse gestartet werden. Die fertige Applikation sollte nun wie folgt aussehen:

Mehr Details zu den Bestandteilen einer JVx Applikation sowie eine Schritt für Schritt Eclipse Anleitung finden sie unter Schritt für Schritt Anleitung zur JVx Applikation.