Table of Contents

In the following paragraphs, we will show you how to create your first JVx application with minimal effort and code.

The application's task is to display data from a database table and make the data editable.

Before you start, you will need the following libraries and tools:

- JVx Binary Package
- Eclipse IDE (>= 3.4) with JDT (recommended: Eclipse IDE for Java EE Developers)
- JDK 8.0 (1.8) or higher
- HSQLDB library (http://www.hsqldb.org)
- JVx Sample Eclipse Project

For our JVx sample application, we need the following parts:

- Application
- Workscreen
- Business Object
- Database Connection

Create an Application

We need an application as a frame for the client. Each application must implement the interface `jvx.rad.application.IApplication`. In our example, we derive from the standard implementation `com.sibvisions.rad.application.Application`, whereby we use the following code:

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
```

Method	Description
Constructor	A specific constructor is needed, as each application is started with a launcher that depends on the technology used. This launcher is passed over to the application in the constructor.
createConnection	The communication protocol is initialized. A `DirectServerConnection` is sufficient for our application, as both the client and the server are started in the same VM. However, if an application server is integrated, a `HttpConnection` could also be used.

Method	Description
getApplicationName	Sets the application name. This name is needed for the communication with the server, as the latter uses the appropriate application configuration depending on the application name. In our case, the application name must be “firstapp”, as the working directory is also called `../JVxFirstApp/rad/`**`firstapp`**`/`.
afterLogin	This method is called by the super class after a successful login. We use this method to extend our menu and our toolbar. It is not necessary to reset the changes after logout, as this is done automatically by the super class.
createMenuItem	Provided by the super class to create menu entries. The first parameter contains the name of the method which is to be called when the menu entry is selected. The second parameter contains the command (ActionCommand), which plays no role in our case. The text of the menu entry is to be defined in the third parameter, and, lastly, the image for the entry is passed over.
createToolBarButton	Similar to `createMenuItem`, except that this method creates a button which adapts to the layout of the toolbar.
UIImage.getImage	Provides a predefined image from the JVx image library. For ease of use, we use a predefined image.
doOpenDBEdit	This method is called when the menu or the toolbar button is pressed and calls the corresponding workscreen.
configureFrame	This method is provided by the super class and ensures that all frames look the same. This includes the menu icon.

Create a Workscreen

Once we have created the application framework, we create our first workscreen with the following 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
```

Method	Description
initializeModel	Initialises the client objects for the access to the server or the data.
InitializeUI	Laying out of the workscreen.
createSubConnection	We create a separate connection to the server. This has the advantage that a separate business object is used on the server. This object contains all objects needed by the workscreen. Once the work screen is closed, the used memory is released. Moreover, each connection can have special parameters and timeouts. The requested business object is defined with the following class: `apps.firstapp.frames.DBEdit`. The class will be created later on.

Member	Description
connection	The connection to the server specially for the workscreen. A special communication protocol is used in the background. In our case, it mirrors the `DirectServerConnection` class.

Member	Description
dataSource	This is the data source and looks after the transfer of data between the client and the server. The connection is used for the transfer.
rdbContacts	The model and the controller for data display. The name contacts defines under which name the server-side business object can be found.

Create a Business Object

Once we have created the client, we need the corresponding business object on the server so as to define the source of the data more precisely. To do so, we use the following 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
```

Description	
The class mirrors the business object for the `DBEditFrame` workscreen. The objects can only be accessed via the SubConnection of the workscreen. Through the class derivation of `apps.firstapp.Session`, it is very easy to access all objects of the session and the `Application`.	
Method	**Description**
getContacts	Enables the access to the database table `CONTACTS`. The method name must match the object name of the RemoteDataBook: `contacts ⇒ getContacts`. Exception handling is taken over by the server.

Create a Database Connection

In the business object, we referred to the data source with the method `getDBAccess()`. In our case, we use a HyperSQL database. We define the database connection in the following class:

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
```

Method	Description
getDBAccess	Opens a new connection to a HSQL database if this has not yet occurred. Exception handling is taken on by the server.

The application is now fully implemented and ready to run. To be able to work with the application, we need the database including the CONTACTS table, which we want to access. It has already been created in the Eclipse project and can be started with `../JVxFirstApp/db/startHSqlDB.bat`.

The First JVx Application

Once the database has been started, the application can be started via the run menu in Eclipse. The finished application should now look as follows:

You can find more details about the components of a JVx application as well as step-by-step instructions for Eclipse under Step by step introductions to the JVx application.