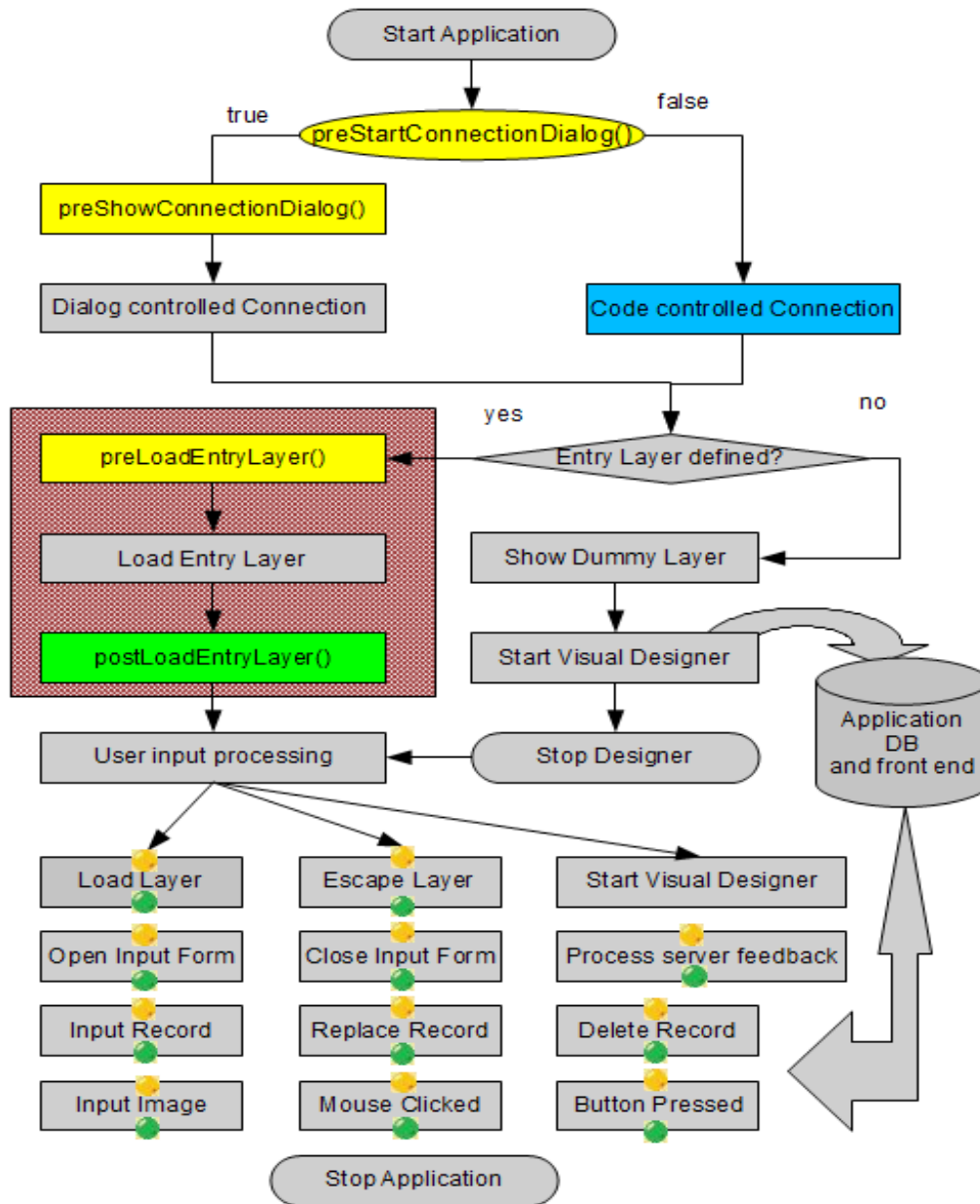


Datalator 3 Programming Guide.

1. Application Life Cycle



Gray shapes represent default Datalator code, yellow and green – user code callbacks, blue – available utility methods. Yellow callbacks - preprocessing callbacks; they may enhance or replace default event processing. Green callbacks are postprocessing user code "hooks"; they may only enhance default actions. Gray rectangles with color dots represent major types of system events, processing of each of them is preceded by a preprocessing callback (yellow dot) and followed by a respectful postprocessing callback (green dot) of user code.

Hatched area shows typical sequence of event processing: yellow preprocessor (if defined) called first, followed by gray default processor and concluded with a green postprocessor (if defined).

Simple CRUD application does not define any callback methods while complex one could employ all about 50 of them plus about 100 utility helpers, as described below.

2. Starting a New Project

To program a Datalator application is to enhance default CRUD functionality of visually built application. We assume that prior to programming an administrator and/or developer(s) created an application, containing a few Layers-components. That assures that the name of the application, terms of connection to the server, the names and relations of Layers are already defined.

The part of a Datalator distribution package – **DatalatorClient3.jar** – is both a generic client (with administrator and developer code built-in) and also a library to create a new project upon it. All the programming done on the client side by extending your class, containing business logic, from **datalator.user.User** like this:

```
package myPackage;
public class MyApp extends datalator.user.User{
    public static void setUserCode(){
        user = new MyApp(); // defining pointer to business logic container
    }
// callbacks defined below, business logic defined by callbacks
.....
}
```

If you plan to deploy your application as an applet you have to define an entry point of your applet like that:

```
package myAnotherPackage;
import javax.swing.*;
public class MyApplet extends datalator.user.UserApplet{
    public void init() {
        try{ // on some systems your applet might look different without this...
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
            SwingUtilities.updateComponentTreeUI(this);
        }
        catch(Exception e){...}
        MyApp.setUserCode(); // defining pointer to business logic container
        User.initApplet(this, "MyApp 1.0 build 2011-xx-31");
    }
}
```

If your deployment method is a Java application you have to define an entry point of your application as shown:

```
package myMainPackage;
import javax.swing.*;
public class Main {
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                MyApp.setUserCode(); // defining pointer to business logic container
                datalator.user.MyApp.startClientApplication(50, 40, 1200, 800,
                                                            "127.0.0.1",
                                                            "MyApp",
                                                            "", "",
                                                            "MyApp3 build 07-29-2011");
            }
        });
    }
}
```

After defining your **Main** and **MyApp** classes you must be able to compile and start your application (assuming you have library **DatalatorClient3.jar** in your classpath). The application frame with geometry (50, 40, 1200, 800) and default connection dialog will be visualized. Nothing exciting yet – it started up just like the library **DatalatorClient3.jar** would start. The latest jar contains it own manifest defining default entry point as **datalator.main.Main**. When starting an application for the first time make sure **your** entry point is an entry point of the application.

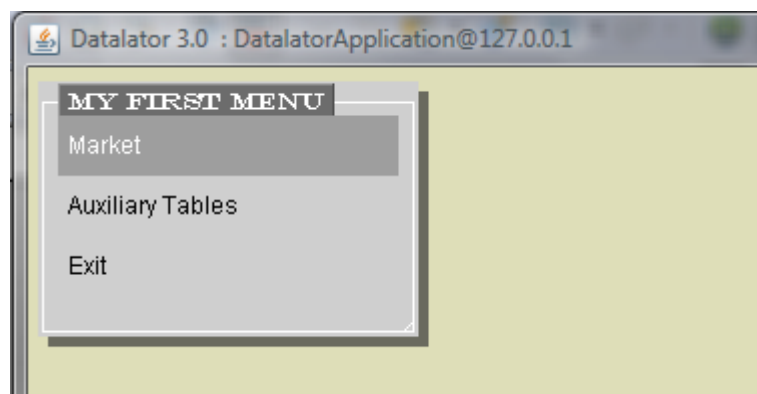
To run your application as an applet you may code the following html tag (there are a few more ways to define an applet on a web page):

```
<APPLET      archive='MyApp.jar'      code='myAnotherPackage.MyApplet'      WIDTH=1200
HEIGHT=800>
<PARAM NAME="AppName"  VALUE="MyApp">
<PARAM NAME="ip"       VALUE="127.0.0.1">
<PARAM NAME="port1"    VALUE="8081">
<PARAM NAME="port2"    VALUE="8082">
</APPLET>
```

Remember, you may define all 6 connection parameters in the html <applet> tag: **AppName**, **ip**, **port1**, **port2**, **id** and **pass** to customize the connection dialog. If not defined a respectful dialog text field will be left empty.

By now you have to be able to start your app both as an application and an applet. Lets see how we can go about programming business logic.

Lets imagine we have visually created an application with "MY FIRST MENU" as a starting Layer, which contains the line "Exit" like shown:



To make our application to shut down when the line "Exit" is activated, we have to define a "hook" in MyApp class – a callback method overwriting **preEnterLayer()** like this:

```
public boolean preEnterLayer(Layer layer, SpravRecord spravRecord){
    String lineContent = spravRecord.content[0];
    if(layer.layerName.equals("MY FIRST MENU")&& lineContent.equals("Exit")){
        shutDown();      // utility method
    }
    return true;        // default processing authorized if "Exit" is not detected
}
```

We will discuss API in details below, for now the fact we could customize an application generic behavior by overwriting a callback preprocessing method **preEnterLayer()** and employing a utility method **shutDown()** is important. All the programming is done this way.

3. Callbacks

There are following groups of callback methods available:

Server connection and starting an application callbacks:

- preStartConnectionDialog
- preShowConnectionDialog
- pre- and postLoadEntryLayer

Navigation callbacks:

- pre- and postEnterLayer
- pre- and postEscapeLayer
- pre- and postOpenServiceMenu
- pre- and postOpenFilterWindow (FilterForm has been "activated")
- pre- and postOpenInputWindow (InputForm has been "activated")
- pre- and postStartSupportLayer ("supporting" layer has been "activated")
- preReportChangingLayer (sending navigational info to the server)

Content modification callbacks:

- pre- and postAddRecord
- pre- and postReplaceRecord
- pre- and postDeleteRecord
- preSwitchInputMode
- pre- and postEnterPressedInputLayer ("Enter" button pressed when Input Form is active and cursor is in the LAST text field - will activate addRecord or replaceRecord)
- pre- and postInputFieldTypedIn ("Enter" button pressed when Input Form is active and cursor is in any text field except the LAST one – means user just filling in the form)
- pre- and postFocusGainedInputField
- pre- and postSettingSupport (a "support" layer is about to set value for supported Field)
- pre- and postImageWindowClicked

Server feedback callbacks:

- contentChangedInternally, contentChangedExternally
- contentLoaded
- contentUpdatedAsynchronously
- recordLockStateChanged
- pre- and postReloadContent

Low level mouse events callbacks:

- pre- and postMouseLeftClicked
- pre- and postMouseRightClicked

Paint callbacks:

- pre- and postPaintCustomComponent
- pre- and postPaintLayer
- pre- and postPaintLayerWindow

There is detailed description of those methods in API JavaDoc and examples bellow.

4. Utility methods

There are following groups of utility methods available:

Starting/Stopping an Applet/Application and Connection to a Server:

- connect, shutDown
- initApplet, startClientApplication
- getUserCode, setUserCode

Content Access and Modification:

- addActiveSpravRecord, appendSpravRecord, appendDocsRecord
- clearRecordLocks
- commitLongTransaction, rollbackLongTransaction, startLongTransaction
- copySameFieldsValues
- createSpravRecord
- deleteRecord
- getTextLayerText, setTextLayerText, saveTextContentAndWait, saveTextLayerText
- reloadContent, reloadContentAndWait
- replaceSpravRecordAndWait
- setField, setInputField, setInputFieldText

Administrator Utilities:

- createApplication, getApplicationSize, removeApplication
- getServerRunTimeInfo
- restartServer, shutDownServer, startServer, stopServer

Navigation Utilities:

- enter, enterMenuLine, escape
- enterInvisibleDocs, enterInvisibleMenu, enterInvisibleSprav
- enterVisibleDocs, enterVisibleMenu, enterVisibleSprav
- openInput, switchInputMode
- restoreKeyboardFocus
- setActiveMenuRecord, setActiveRecord
- startInputSupportLayer, startFilterSupportLayer

Filter Form Control:

- removeFilterFieldSupport, setSupportMode
- setFilterField, setAuxFilterFieldText, setFilterFieldText
- setFilterFieldOperation

Convenience Methods:

- downloadFile, stopDownload
- formatPresentDate, formatPresentTime
- logLine
- showMessage, showMessageImmediately

Data Structures Getters:

- getActiveLayer, getParentLayer, getLayerByName, getLayerFloor, getLayers
- getActiveRecord, getRecordsNumber
- getBackgroundNames
- getCustomComponentByName, getCustomComponents
- getField, getFilterField, getFilterFields, getInputField, getInputFields
- getHtmlLayerModeButton, getTextLayerSaveButton
- getInputLayerOwner
- getInputMode
- getLocalIP, getUserGroup, getUserName
- getMainPanelHeight, getMainPanelWidth, getTopContainer
- isApplet, getWebPageParameter

Appearance Control:

- repaintAll
- setBackground
- setDefaultCursor, setWaitCursor
- setLayerMainWindowCaption
- setLocation, setSize, setShadowDefault, setShadowExtended
- setTextFieldColors
- setTextLayerButtonsVisible
- showRecordLockPeopleWorkingBelow

There is detailed description of those methods in API JavaDoc and examples bellow.

5. Accessible Objects

Layers.

Top accessible objects in an application are Layers. Four simple Layers consist of one single visible window: those are Menu Layer, Text and HTML Layer, Custom Layer. More complex Table consists of a Main Table Window(one layer) and an Input Form(connected layer). Plus there might be a few of associated Report Layers. A Filtered Table adds to that set a Filter Form. Those objects, if visualized, are represented by usually overlaid JPanels.

Lifecycle of those Layers, including Forms and Reports, starts with "entering" them when the layer definition and content are loaded and (optionally) visualized. Then visualized Layers accept user input, updates from the server and programmed actions. A Layer could be loaded in invisible mode. Those "stealth" layers serve primarily to access and/or modify their content on the background asynchronously. Non-visuals could only be accessed programmatically. The lifecycle of a Layer ends when it is "escaped" by user actions or by calling a utility method.

The following diagram shows Layers hierarchy. It is useful when casting Layers or passing them as parameters:

```
javax.swing.JPanel
├── datalator.layers.Layer
│   ├── datalator.layers.DummyLayer
│   ├── datalator.layers.InputLayer
│   │   ├── datalator.layers.FilterLayer
│   ├── datalator.layers.sprav.SpravLayer    (Table Layer implementation)
│   │   ├── datalator.layers.docs.DocsLayer (Filtered Table implementation)
│   │   ├── datalator.layers.report.ReportLayer
│   │   ├── datalator.layers.menu.MenuLayer
│   │   │   ├── datalator.layers.custom.CustomLayer
│   │   │   │   ├── datalator.layers.text.TextLayer
│   │   │   │   └── datalator.layers.text.HtmlLayer
```

The following snippet shows accessible fields of the class Layer:

```
package datalator.layers.Layer;
public class Layer extends JPanel ...{
    public int            x, y, dx, dy;           // layer geometry
    public int            verticalSpread = 30;    // pixels between lines
    public Shadow         shadow          = null; // pointer to a Shadow
    public String         layerName       = "";
    public String         layerType      = "";
    public Font           font            = null;
    public Font           captionFont    = null;
    public Vector<SpravRecord> records      = null; // data container
    public Vector<InputField> fields       = null; // data fields definition
    public Vector<UIElement> elements     = null; // visually designed UI elements
    public Layer          parentLayer     = null; // pointer to a parent
    public String         parentLayerName = "";  // parent-layer parameters
    public String         parentLayerType = "";
    public long           parentRecordId  = 0;
    .....
}
```

Content of a Layer (Model).

All Layers, except empty "custom" one, assume existence of persistent content, shared among all instances of the Layer on all computers, and residing on the server. By default that content is synchronized – if change is detected, all Layers will be notified and will attempt to reload the content.

Content represented by **java.util.Vector records**, consisting of of a number of Records (table rows), which in turn consist of a number of Fields (table columns) – all represented by **java.lang.Strings**. There are also a few service Fields, containing link and unique record identifier. There are utility methods to load or reload content of a given Layer along with methods to access/change records and/or records fields.

That following snippet shows accessible fields in a Table data container – **datalator.layers.sprav.SpravRecord**:

```
package datalator.layers.sprav;
public class SpravRecord{
    public String[] content      = null; // user's data container
    public String  link         = "";   // service field
    public String  linkType     = "";   // service field
    public long    parentRecordId = 0;    // service field
    public long    recordId      = 0;    // service field - UID
    .....
}
```

Normally we do not need to access those fields directly – there is API in place. For example, to load a Table named "My Friends" and to get a value from a field named "Phone #" of the 10th record, you might use the following code:

```
...
SpravLayer myFriendsLayer = enterInvisibleSprav (null, "My Friends");
setActiveRecord(myFriendsLayer, 9); // set the pointer to the 10th record
SpravRecord activeRecord  = getActiveRecord(myFriendsLayer);
String      phone         = getField(myFriendsLayer, activeRecord, "Phone #");
escape(myFriendLayer);    // release the Layer
...
```

A Table Layer supports a list of Data Fields, corresponding to **String[] content** of the Layer's SpravRecord. That list is a **Vector<InputField> fields** and accessible internals of the InputField class are:

```
package datalator.layers;
public class InputField {
    public int x,y; // location in the Input Form
    public String name = ""; // name, i.e. "Phone #"
    public String type = "";
    public JLabel label = null;
    public InputTextField textField = null;
    public JButton button = null;
    public Layer ownerLayer = null; // pointer to InputLayer
}
```

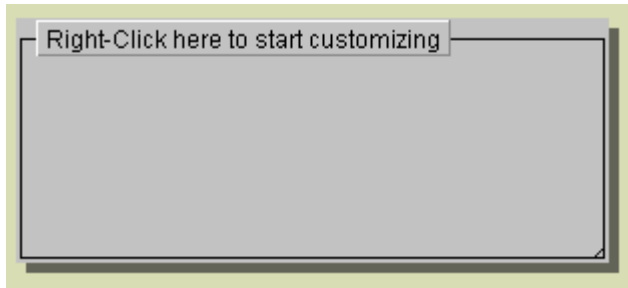
A Filtered Table Layer supports additional list of Filter Fields, corresponding to the set of text fields presented in FilterForm. They are contained in public **Vector<FilterField> filterFields**. FilterField class is extended from InputField.

Normally the Datalator Java API hides those internals and uses only following objects when communication with user code:

```
Layer, SpravLayer, DocsLayer, MenuLayer, TextLayer, HtmlLayer,
CustomLayer;
SpravRecord and MenuRecord, InputField and FilterField;
Java String, Java Image and primitive Java data types: int and long.
```

Visual components of a Layer (View).

A gray Dummy Layer and empty **Custom Layer** consists of just one Main Window and it's Shadow, represented by Java Swing JPanels. A **Menu Layer** consists of Main Window (bearing a few lines) and a Shadow as well. Location, size, caption, fonts and colors of a Main Window and it's Shadow are defined as **public** and could be accessed/changed programmatically if needed. The same applicable to one-window **Text** and **HTML Layers** – design elements and content accessible from Java code.



A **Table Layer** is represented visually by Main Window with a Shadow and number of Field Windows without Shadows, hosted inside the Main Window. Default painting of all windows could be complemented or substituted by custom code if needed. Associated **Input Form** is a Layer itself, thus providing the same access rules to all it's design components, which are Main Window, Shadow, labels, text fields, "activation" button and additional UI elements(if defined). **Table Reports** when defined, are also Layers with all applicable access rules to design elements.



And a **Filtered Table Layer** is no different from the plain **Table Layer**. It contains the same old Main Window, Shadow, Field Windows, **Input Form** with its components, optional set of **Reports** plus a **Filter Form**, which is also a Layer.

The screenshot displays a software interface with a sidebar menu on the left containing 'Boss Menu', 'Manager Menu', and 'Trips'. The main area features a 'Filter' form at the top right with fields for Dispatcher, Market, Driver, Truck, Trailer, and Date. Below this is a table titled 'Trips = 5:42' with columns: Trip, Driver, Truck, Date, Price Out, Price In, Stops, Price, Expenses, and Mileage. The table lists various trips, with trip AD482 highlighted in red. Below the table is a 'Browse Record' form for trip AD482, containing fields for Skids Out/In, Spots Out/In, Stops, Weight Out/In, Market, Dispatcher, Price Out/In, Border Out/In, Mileage, Fuel used, Fuel paid, Tolls paid, Repairs, and Other expenses. At the bottom, summary statistics are shown: Average stops = 5.45, Average = \$2,516.57, and Gross = \$105,696.10.

Trip	Driver	Truck	Date	Price Out	Price In	Stops	Price	Expenses	Mileage
VC234	Vlad1	804	2011-09-23	\$550.00	\$2,050.00	6	\$2,600.00	-\$0.00	
VC233	Vlad1	804	2011-09-21	\$450.00	\$2,050.00	5	\$2,500.00	-\$0.00	
VC232	Vlad1	804	2011-09-19	\$525.00	\$2,615.00	6	\$3,140.00	-\$0.00	
AD483	Andrey	804	2011-09-15	\$525.00	\$2,550.00	8	\$3,075.00	-\$0.00	
AD482	Andrey	804	2011-09-14	\$575.00	\$2,817.50	8	\$3,392.50	-\$0.00	
VC231	Vlad1	804	2011-09-09	\$500.00	\$2,025.00	5	\$2,525.00	-\$0.00	
VC230	Vlad1	804	2011-09-07	\$450.00	\$2,150.00	5	\$2,600.00	-\$0.00	
VC229	Vlad1	804	2011-09-02	\$750.00	\$1,702.50	5	\$2,452.50	-\$0.00	
VC228	Vlad1	804	2011-08-31	\$600.00	\$1,675.00	4	\$2,275.00	-\$0.00	

Trip	AD482	Skids Out	26	Skids In	26	Stops	52	Mileage	
Driver	Andrey	Spots Out	26	Spots In	26	Stops	8	Fuel used	
Truck	804	Weight Out	35000	Weight In	33833	Market	Pittsburgh, PA	Fuel paid	
Trailer	4081	Stops Out	1	Stops In	7	Dispatcher	Ella	Tolls paid	
Date	2011-09-14	Price Out	575.0	Price In	2817.5	Price	3392.5	Repairs	
		Border Out		Border In		Expenses	-0.0	Other expenses	

Average stops = 5.45 Average = \$2,516.57 Gross = \$105,696.10

6. Programming of Some Common Tasks.

Default CRUD functionality of a Datalator application is enough to input, maintain and share your Tables. It is usually not enough for a real-life application which has to calculate summaries, prepare reports and diagrams, tune default user access to tables or present a table content in some fancy 3D style. To modify an application behavior one has to overwrite a callback function(s) to intercept a specific event(s) of the application life cycle and program that event custom processing. Lets take a look at the common tasks of application modification.

6.1. Bypassing the default Connection Dialog when connecting to a server:

```
public boolean preStartConnectionDialog(){
    setWaitCursor();
    if(!connect(8081, 8082,                // server ports
               "myServer.com", "myApp 1.0", // server URL(IP) and app name
               "guest", "guest",           // connection credentials
               true,                        // server feedback allowed
               "1111222211112222")){        // server id
        System.out.println("Cannot connect...");
        shutDown();                        // shut the application down
    }
    setDefaultCursor();
    return false;                          // connected successfully, do not show connection
                                           // dialog, start myApp 1.0
}
```

6.2. Modification of the default Connection Dialog:

```
public void preShowConnectionDialog(Properties comps){
    JTextField port1 = (JTextField)comps.get("TextField Server Port");
    if(port1 != null) {port1.setEnabled(false); port1.setText("8081");}
    JTextField port2 = (JTextField)comps.get("TextField Feedback Port");
    if(port2 != null) {port2.setEnabled(false); port2.setText("8082");}
    JTextField server = (JTextField)comps.get("TextField Server");
    if(server != null) {server.setText("mycloudserver.com");}
    JTextField app = (JTextField)comps.get("TextField Application");
    if(app != null) { app.setText("My Cloud Application");}

    JTextField user = (JTextField)comps.get("TextField User");
    if(user != null) { user.requestFocusInWindow();}
}
```

6.3. Modification behavior before loading an "Entry Layer".

Might be used to check reminders and show introductions, set the application background, navigate a specific user to a specific layer.

```
public boolean preLoadEntryLayer(String appName, String entryLayerName, String
userName, String password){
    setBackground("ice-blue");
    if(getUserGroup().equals("Dispatchers")){ // dispatcher goes to Op. Map only
        MenuLayer managerMenu = enterInvisibleMenu(null, "Manager Menu");
        setActiveMenuRecord(managerMenu, "Operating Map");
        enter(managerMenu);
        return false;
    }
    return true;
}
```

To modify an application behavior after loading of the "entry Layer" use:

```
public void postLoadEntryLayer()
```

6.4. Association of specific actions with specific lines of a Menu Layer.

Code shown below detects when a specific line of a specific menu was "entered" (double-clicked or enter-keyed) and executes associated method.

```
public boolean preEnterLayer(Layer layer, SpravRecord record){
    String layerName = layer.layerName;

    if(layerName.equals("Boss Menu") && record.content[0].equals("Exit")){
        shutDown();
    }
    if(layerName.equals("Trip Menu") && record.content[0].equals("Log Trip")){
        logTrip(); // some custom logic
        return false; // deny default processing
    }

    return true; // default processing allowed
}
```

6.5. Processing content of already loaded Layer.

In snippets below **postEnterLayer()** gets invoked **after** associated Layer was loaded. In the first <if> one can assume safely that the Custom Layer "Operating Map" has been loaded and visualized and Java object with the same name can start drawing on the top of the layer. The second <if> calls the constructor of another Java object, assuming that respectful Layer(JPanel) is ready to start working with. In both cases the easiest way to access just loaded Layer is to employ **getActiveLayer()** method.

```
public void postEnterLayer(Layer layer, SpravRecord record){
    String layerName = layer.layerName;
    String content    = record.content[0];

    if(layerName.equals("Manager Menu")&& content.equals("Operating Map")){
        operatingMap = new OperatingMap();
        operatingMap.startOperatingMap();
    }
    if(layerName.equals("Manager Menu")&& content.equals("Statistics")){
        statistics = new Statistics();
    }
}
```

6.6. Prevention of a specific user from leaving ("escaping") a specific Layer.

Or might be used to introduce some specific action when a Layer is "escaped". The Java boolean **flagShuttingItDown** is the only static field of the class User; it serves to prevent recursion when a user initiates application shutdown, which forces to escape all the Layers first.

```
public boolean preEscapeLayer(Layer layer){
    if(flagShuttingItDown) return true;    // allow App to shut down (default)

    String layerName = layer.layerName;

    if(layerName.equals("Operating Map") && getUserGroup().equals("Dispatchers")){
        return false;           // to prevent leaving the "Operating map"
    }
    if(layerName.equals("Border Status")){
        restoreBorderStatus(); // special processing required when leaving "Border
                               // Status" Menu Layer
    }
    return true;                // default processing authorized
}
```

To detect and process an event when a specific Layer is already closed or "escaped" use

```
public void postEscapeLayer(Layer layer)
```

6.7. Interception of notification that content was just changed by ANOTHER user.

By default all affected loaded Layers will try to asynchronously reload their content – no intervention required if you are OK with default data representation in the form of tables. If you program a game-like interface or you must recalculate some dependant Layers, you have to intercept the event to introduce your special business logic.

```
public boolean contentChangedExternally(String layerName){
    if(layerName.equals("New Trips")){
        Layer operatingLayer = getLayerByName("Operating Map");
        if(operatingLayer != null){
            operatingMap.loadTrips();    // manually update required tables
            operatingMap.layoutMap();    // do some custom painting
        }
    }
    return true;    // allow default update of "New Trips" on the background
}
```

6.8. Interception of notification that content was just changed by YOURSELF.

A case when you might want to overwrite this callback is when you want to highlight your changes in one color and somebody's else – in another.

```
public void contentChangedInternally(String layerName)
```

6.9. Interception of user's attempt to open an InputForm.

Used to deny modification of a Layer content.

```
public boolean preOpenInputWindow(SpravLayer spravLayer){
    if(spravLayer.layerName.equals("Who Is Online")) return false;
    return true;
}
```

To populate some text fields of the open InputForm (after the Form is visualized) use:

```
public void postOpenInputWindow(SpravLayer spravLayer)
```

There is no obvious reason to check when FilterForm is opened (InputForm is more interesting from the user point of view) but callbacks are in place:

```
public boolean preOpenFilterWindow(DocsLayer docsLayer)
public void postOpenFilterWindow(DocsLayer docsLayer)
```

6.10. Interception of user's attempt to open a "Service Menu".

Used to prevent the "Service Menu" from "opening" for specific users or to change lines or appearance of the Menu.

```
public boolean preOpenServiceMenu(SpravLayer layer, String serviceName){
    if(layer.layerName.equals("Applications")) return false; // prevent from
    if(layer.layerName.equals("Users Groups")) return false; // "opening"
    return true;
}
```

There is also a postprocessor callback:

```
public void postOpenServiceMenu(SpravLayer spravLayer, String supportMenuName)
```

6.11. Controlling of reporting on changing Layers by user.

Every time when a human user or a Java code "enters" or "escapes" a Layer the server gets notification like: user "A" has moved from Layer "B" to a Layer "C".

That information is useful in SaaS applications. The server administrator may also see "who's where" in real time in the "Who Is Online" Table. To stop navigation reporting and conserve some bandwidth or to create some custom logic use:

```
public boolean preReportChangingLayer(Layer source, Layer dest, boolean enter)
```

6.12. Interception of user activities which alter content.

To deny or assist or update dependant Tables when user changes content of a specific Table, use one of pre-processors or post-processors as shown below.

```
public boolean preAddRecord(SpravLayer spravLayer, SpravRecord spravRecord)
public boolean preDeleteRecord(SpravLayer spravLayer)
public boolean preReplaceRecord(SpravLayer spravLayer, SpravRecord spravRecord)
public void postAddRecord(SpravLayer spravLayer, SpravRecord spravRecord)
public void postDeleteRecord(SpravLayer spravLayer)
public void postReplaceRecord(SpravLayer layer, SpravRecord spravRecord)
```

```
public boolean preAddRecord(SpravLayer layer, SpravRecord spravRecord){
    if(layer.layerName.equals("Users")){
        setUserUp(layer, spravRecord); //record is complemented with a timestamp+
    }
    return true;
}
```

```
public boolean preDeleteRecord(SpravLayer spravLayer){
    if(spravLayer.layerName.equals("Who Is Online")) return false; // no erasing!
    return true;
}
```

```
public boolean preReplaceRecord(SpravLayer layer, SpravRecord spravRecord){
    if(layer.layerName.equals("Profiles")){
        replaceProfilesRecord(layer, spravRecord); //special record processing added
    }
    return true;
}
```

6.13. Interception of user activities which alter content.

To deny or assist or update dependant Fields or Tables when user typing in the text fields of the opened InputForm use one of the following callbacks.

To process input focus movement hooks pre- and postFocusGainedInputField are used.

To prevent user from switching to some InputForm modes use preSwitchInputMode.

There is a pair of callbacks to catch an event when human user presses "Enter" button in the LAST text field (same as user presses an InputForm "activator" button) ; they are pre- and postEnterPressedInputLayer.

And there is a pair to detect when user presses "Enter" button in any other text field – say, he's just filling the form - pre- and postInputFieldTypedIn.

See examples below:

```
public boolean preSwitchInputMode(SpravLayer spravLayer, int editMode){
    if(getUserName().equals("guest")) return false; // only input allowed
    return true;
}
```

```
public boolean preFocusGainedInputField(Layer layer, InputField inputField){
    if(layer.layerName.equals("Layers") && inputField.name.equals("Type")){
        .... // business logic
        inputField.textField.setBackground(Color.red); // some actions...
        return false;
    }
    return true;
}
```

And we have no examples for:

```
public boolean preInputFieldTypedIn (Layer layer)
public void postInputFieldTypedIn(Layer layer)
public boolean preEnterPressedInputLayer (SpravLayer l, InputField f, int mode)
public void postEnterPressedInputLayer (SpravLayer l, InputField f, int mode)
```

6.14. Interception of a user attempt to alter an image.

A Table field type could be defined as "Image" when visually developing the Table Layer. By default everybody allowed to left-click on the image to upload it on the server – effectively changing content (right-click will start "Save Image on Disk" dialog). If you want to prevent non-authorized users to alter your images and to download them use:

```
public boolean preImageWindowClicked(String imageId){
    if(getUserName().equals("guest")) return false; // no guests access allowed
    return true;
}
```

A Table field type could be defined as "Image" when visually developing the Table Layer. Hide fields from Input Form. Hide Menus/Tables from direct access.

7. Controlling Access to Tables and Table Fields.

When visually creating an application you may want to restrict users access to specific "auxiliary" Tables or Menus, like list of countries or currencies or a layer with some special parameters. To restrict access to some Layers from UI employ the following technique:



Designate a line of the Menu which is associated with "hidable" Data – in the example it is "Auxiliary Tables" line. Move this line of the Menu to the very bottom with help of the Construction Tool. Resize the Menu to "hide" that line. A user with no developer rights has no access to hidden structures anymore. A developer can always resize any Layer (with or without saving it) and thus gain access to those "hidden" Layers. Of course all those Layers still could be loaded and manipulated programmatically. Here is what "Auxiliary Tables" line was hiding:



When visually creating a Table you control how many Data Fields will be visible in the Main Layer Window – by checking a Field "Visible" attribute:

Design Data Fields Filter Fields UI Elements Special Ops Reports

Name	Type	Visible	Link
Goods Category	String	☒	
Brand	String	☒	
Item name	String	☒	
Price	Currency	☒	
Date Posted	Date	☒	
Item Picture	Image	☒	

That part is easy.

Another scenario is when you want to restrict access to some fields from a Table Input Form:

The screenshot shows a web application interface. At the top, there's a green header bar with the text "Classifieds = 1 : 1". Below this is a table with six columns: "Goods Category", "Brand", "Item name", "Price", "Date Posted", and "Item Picture". The first row of the table contains the following data: "Electronics", "HP", "Printer HP 1", "\$150.00", "2011-09-14", and an image of a printer. A "Browse Record" dialog box is open in the foreground, showing the "Item name" field with the value "Printer HP 1" and a "Price" field with the value "150". The "Item Picture" column in the table shows a small image of a printer.

In the picture above we have six Data Fields visible with only two of them accessible from the Input Form. It is done by placing Data Field beyond the borders of the Input Form, like that:

This screenshot shows the same web application interface as the previous one, but with a "Filter" section at the top. The "Filter" section has four fields: "Goods Category", "Brand", "Price Range", and "Date Posted", each with a dropdown menu and a text input field. The table below still shows the same record. The "Browse Record" dialog box is open, showing the "Item name" field with the value "Printer HP 1" and the "Price" field with the value "150". Additionally, the "Goods Category" field is set to "Electronics" and the "Brand" field is set to "HP". The "Date Posted" field is also visible in the dialog box, showing the value "2011-09-14".

In that example content of the Data Fields "Goods Category", "Brand" and "Date Posted" populated automatically – by default Datalator does it for all Data Fields matching Filter Fields. If we wanted to intercept the addition of a record to the database we would use

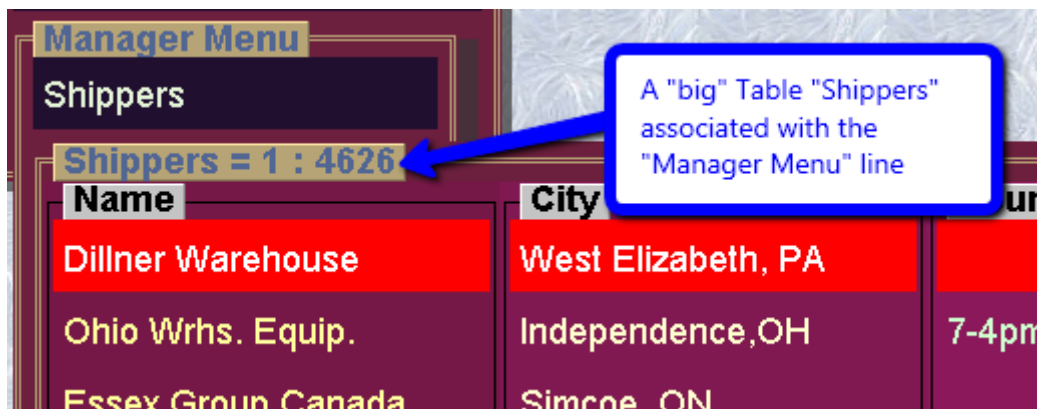
```
public boolean preAddRecord(SpravLayer spravLayer, SpravRecord record) {
    if(spravLayer.layerName.equals("Classifieds")) {
        ....
        setField(spravLayer, record, "Date Posted", formatPresentDate());
        ....
    }
    return true;
}
```

Another example is programming a forum-like application, with a Layer "Threads" with visible in the Main Window Data Fields "Number of Views" and "Number of Replays". Hide them in the Input Form and content of those fields will be modified strictly programmatically.

8. Some Specifics of Datalator API and Architecture.

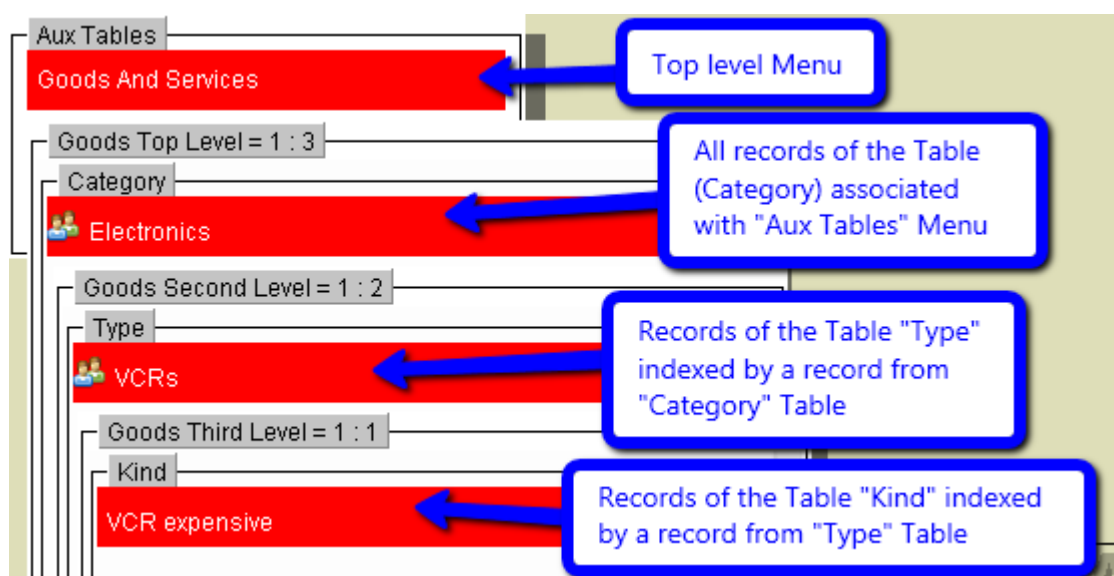
Datalator client-server architecture based on the category of a Layer – the main building block. Definitions and contents of an application's Layers reside in the server Database. **Programming of a Datalator applications is done by programming a Datalator client only.** Advantages of such approach are simplicity and security; the drawback is inefficient use of long flat plain Tables (as contrary to Filtered Tables) and the need to transmit the whole Table to process the whole Table. And a Table has to fit into the server (SQL ResultSet) and the client RAM.

When a user (or user code) loads a Table on the client side, first the Table's definition (think layout) and then content transmitted to the client. A Table associated with a Menu line gets loaded whole – means big tables at the top level of an application hierarchy might be inefficient. On the example below a Table, containing of 4626 records, is always transmitted to the client when "Shippers" line of the "Manager Menu" is activated. In our environment the Table content (~300Kbs) transmitted over the "broadband" connection (~2Mbps) with the server being Atom-minimalist (512 Mbs RAM) Linux server in about 2 seconds. Layers' definitions are always cached on server and client sides, content caching relies on abilities of the database employed.



Name	City	ur
Dillner Warehouse	West Elizabeth, PA	
Ohio Wrhs. Equip.	Independence, OH	7-4pm
Essex Group Canada	Simcoe, ON	

To avoid transmission of big Tables a developer might want to replace one long Table by hierarchy of Tables like: facilities-departments-employees or years-months-days. In such hierarchies only records associated with a record of a "parent" Table will be transmitted to the client.



The best container for the long Tables is a Filtered Table. A Filter is a great tool to control the number of records being transmitted.

When modifying content only affected records transmitted to the server.

If a Table record deleted – all hierarchically associated records are deleted automatically – thus enforcing the Database consistency. Although Filtered Tables content will not be deleted when one deletes a Table record used to index that Filtered Table content. Example: a Filtered Table "Orders" depend on both "Departments" and "Employees" Tables from hierarchy departments-employees. When deleting one of the "Departments" - all "Employees", used to work in restructured Department will be deleted as well. Although the Filtered Table "Orders" content will not be changed automatically. Of course, such default behavior might be reprogrammed.

Shortly we can state that Tables load all the content and change it one record at a time. Images embedded in a Table record never cached and loaded for the current active record only – one image at a time. Text and Html Layers load all the content and transmit back the whole changed content. And Text/Html Layers content is never locked for editing. Conclusion: break big tables into hierarchies or use Docs.

To better illustrate chosen architecture implications and benefits, lets consider a scenario when an application supposed to provide users' "sign up" functionality.

Straight approach is to create a "Sign Up" plain Table, load it invisibly, open it's InputForm, enforce "input"-mode only, make the Form visible and allow the user to fill in required text fields. That approach will provide all functionality and authorized user might have full access to the "Sign Up" Table – all the names, dates, phones etc. of all users are right at the fingertips.

The first problem – all the content of "Sign Up" Table will be passed to the client application when a new user loads it to sign up. The resources are not used wise. Another problem – a hacker might decompile the application code to extract the Layer name and connection parameters. Nothing prevents the hacker now to program a simple application which connects to our host server and loads "Sign Up" Layer with all the content in visible mode.

Obfuscation might not prevent it. Employment of another Table(s) as "hidden container(s)" is not reliable either. As soon as the obfuscated code recovered – all the "secret" Layers become not so secret and might be loaded by malicious code if the Layer name is known.

The present workaround is to mimic server-side programming by developing two client applications. One is public, source is available, it serves the half of the process: connects to the server with credentials "junior user"/"junior user", loads the invisible "Sign Up" Table with visible Input Form and accepts the user input. By default the server will not load/reload Tables content for the "junior users". Another "secure" client has to be programmed to connect to the same server with different credentials. It will load "Sign Up" Table and will wait till notification "content changed" comes from the server . When notified, the secure client will reload the Table content, such getting access to the freshly created "Sign Up" record, and will move the record to another Table (or Filtered Table) . After completion the secure client will wait for notification again.

Now the sensitive information is protected by connection credentials and the name of the Table-container of sign up information.

The two-clients technique might help when creating bridges between HTTP clients and Datalator database.

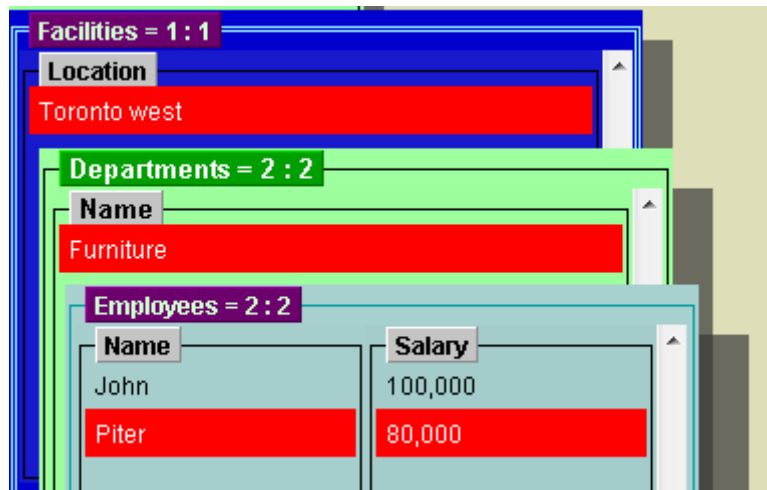
Still the most studied (and preferable) environment is LAN or Cloud based firewalled server with clients' access regulated by credentials and roles. Synchronized access to shared data in medium size workgroups is what it was made for. How it will scale still has to be seen. Datalator might do well in less regulated environments, but it was not intended for the "open" Internet yet. Still we are quietly running our cloud application for 2 years now, watching the office when on vacations... nice :-).

9. Using Tables Threes as Keys in Filters.

When we visually build Tables trees (or hierarchies) like classic "facility-department-employee" schemas, the Construction Tool prepares scenarios to generate SQL statements like:

SELECT * FROM 'departments' WHERE facility='Toronto west'

when a specific "Facility" record is "activated". Control of such hierarchies in Datalator is pure visual – it is created in point-and-click fashion and it used the same way.



Another situation is when we want to define and control schemas where the search (select) criteria is more complex, like:

SELECT * FROM 'Market' WHERE Brand Name='Toyota' AND ... AND Posting Date between 2011-08-01 and 2011-10-07

Luckily the control of such data structures is point-and-click as well – no programming of a single line of code required.

The screenshot shows a filter interface with the following fields and values:

Filter	Category	Brand Name	Price Range	Posting Date
	=	=	=	><
		Toyota		2011-08-01 2011-10-07

Below the filter interface is a table with the following columns and data:

Item Name	Price	Posting Date	Item Picture
A table	\$100.00	2011-09-15	

By the way, the most interesting and complex schemas which Datalator supports out of the box are those where a Filtered Table(s) is used to index(or key) another Filtered Table. For example we build a Filtered Table "Cars" which contain a list of vehicles of different types, makes, age and colors. Another Filtered Table might be the list of the "Claims" of different type of claims, price brackets and "Cars" involved. Such a schema might prove itself useful to quickly find a record and to analyze, f.e. implications of "black" car color on probability of a claim. No programming required, build in 20 minutes.

The only schemas which are not 100% of code free is when an application has to use plain Tables hierarchies as keys in Filtered Tables, like "Category-Type-Kind" in previous example. By default when a record from "Goods Top Level" Table is selected:

Filter

Category =

Type =

Goods Top Level = 2:3

- Category
- Electronics
- Cars**
- Furniture

Filter

Category =

Type =

Kind =

Goods Top Level = 2:3

- Category
- Electronics
- Cars**
- Furniture

the content "Cars" of the chosen record of the Table "Goods Top Level" is placed in the respectful text field of the Filter and "Goods Top Level" Table escaped. But it must stay – because if we click on the Filter Field "Type" now – the filter will load the whole content of the Table with the "Goods of the next Level", with types of "Electronics", "Cars" and "Furniture". Still sometimes "Goods Top Level" must go – when the Filter does not use lower Layer of Tables hierarchy. The desirable behavior of the Filter must be better defined and it will likely be done in the next version of the Datalator – the whole Construction Tool has to be redesigned.

As for now we suggest the following workaround to support Tables hierarchies as keys in Filtered Tables.

```
public class BuyAndSell extends User{
    .....

    String[] supportedFieldNames = {"Category", "Type", "Kind"};
    String[] levels={"Goods Top Level","Goods Second Level","Goods Third Level"};
    boolean[] levelActivatedForSupportFlag = new boolean[levels.length];
    boolean[] levelPreventReentryFlag = new boolean[levels.length];

    // Here we cycle thru the list of the Layers names used as keys
    // when the user's mouse click on the filter field is detected.
    // Then the code gets the pointer to our "Filter" casting it to InputLayer,
    // mimics default processing by extending the Shadow and
    // always start choosing from the "Top Level" in enterVisibleSprav(...)
    // on the way setting up some service boolean variables.

    public boolean preStartSupportLayer(String layerName){
        for(int i = 0; i < levels.length; i++){
            if(layerName.equals(levels[i])){
                InputLayer inputLayer = (InputLayer)getActiveLayer();
                setShadowExtended(inputLayer);
                enterVisibleSprav(null, levels[0]); // always load "Top Level" first
                setSupportMode(false);
                levelActivatedForSupportFlag[0] = true;
                return false; // suppress default processing
            }
        }
        return true;
    }
    // When attempt to "escape" the "key" layer is detected,
```

```

// we must to clean up our boolean flags and
// notify the Filter that it's text field is not waiting for input from a
// "key" Table anymore - removeFilterFieldSupport(...)

public boolean preEscapeLayer(Layer layer){
    for(int i = 0; i < levels.length; i++){
        if(layer.layerName.equals(levels[i]) && levelActivatedForSupportFlag[i]){
            removeFilterFieldSupport((SpravLayer)layer);
            levelActivatedForSupportFlag[i] = false;
        }
    }
    return true;
}

// the meat is here - modification of default "enter" event processing:
// - preventing from reentering the "next" Layer when issuing enter() below
// - checking the "current" Layer name against supportedFieldNames[]
// - setting up the supported text field - enter(...)
// - escaping(closing up) all already open layer of hierarchy

public boolean preEnterLayer(Layer layer){
    for(int i = 0; i < levels.length; i++){
        if(layer.layerName.equals(levels[i])){
            if(levelPreventReentryFlag[i]) {
                levelPreventReentryFlag [i] = false;
                return true;
            }
            Layer couldBeFilterLayer = getLayerByName("Filter");
            if(couldBeFilterLayer == null) return true;

            InputLayer filterLayer = (InputLayer) couldBeFilterLayer;
            String supName = filterLayer.supportedInputField.name;

            if(supName.equals(supportedFieldNames [i])){
                setSupportMode(true);
                levelPreventReentryFlag[i] = true;
                enter((SpravLayer)layer);          // here it fiils in the text field of
                                                    // the "Filter" and will invoke the
                                                    // preEnterLayer again - thus
                                                    // levelPreventReentryFlag[i] = true;
                while(!getActiveLayer().layerName.equals("Filter"))
                    escape(getActiveLayer());
                return false;
            }
            else{
                setSupportMode(false);
                levelActivatedForSupportFlag [i] = true;
                return true;
            }
        }
    }
    return true;
}
}

```

Well, although those are not exactly simple 55 lines of code, but it takes to change only 2 lines to incorporate them into your particular application:

```

String[] supportedFieldNames      = {"Category", "Type", "Kind"};
String[] levels={"Goods Top Level","Goods Second Level","Goods Third Level"};

```

10. More Examples.

There are about 150 methods in Datalator API which are equipped with JavaDoc. There are some useful examples of the whole applications below.

The following snippet shows what was removed from previous discussion of how to program hierarchies made of "key" Tables.

```
public class BuyAndSell extends User{
    public static void setUserCode(){ user = new BuyAndSell();}

    public boolean preEnterLayer(Layer layer, SpravRecord record){
        if(layer.layerName.equals("Main Menu") && record.content[0].equals("exit")){
            shutDown();
        }
        return true;
    }

    public void preShowConnectionDialog(Properties comps){
        if(!isApplet()){
            JTextField port1 = (JTextField)comps.get("TextField Server Port");
            if(port1 != null) {port1.setEnabled(false); port1.setText("8081");}
            JTextField port2 = (JTextField)comps.get("TextField Feedback Port");
            if(port2 != null) {port2.setEnabled(false); port2.setText("8082");}

            JTextField user = (JTextField)comps.get("TextField User");
            if(user != null) { user.requestFocusInWindow();}
        }
    }
}
```

And the following snippet is the entire source of FancyApplet serving our www.fancydata.com web site. It is incorporated into three web pages, which are different only by one line bearing "pageParameter" for pages "forums", "contact" and "howtos" :

```
<APPLET archive='FancyData.jar' code='datalator.user.FancyDataApplet' WIDTH=860
HEIGHT=520>
<PARAM NAME="AppName" VALUE="FancyData">
<PARAM NAME="ip" VALUE="fancydata.com">
<PARAM NAME="port1" VALUE="8081">
<PARAM NAME="port2" VALUE="8082">
<PARAM NAME="pageParameter" VALUE="forums">
</APPLET>
```

The source of the applet is:

```
package datalator.user;
import javax.swing.*;
public class FancyDataApplet extends UserApplet{
    public void init() {
        try{UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
            SwingUtilities.updateComponentTreeUI(this);
        }
        catch(Exception e){}
        FancyData.setUserCode();
        User.initApplet(this, "FancyData 1.0 build 2011-08-31");
    }
}
```

The whole source of the FancyData.java is here all 250 lines. That is a real life application, server as a Java applet.

It is possible to select, copy and paste that source and it will compile as an application if you have DatalatorClient3.jar in your classpath. It is possible to remove or alter some "protective" methods where guest is denied access to content modification, like in preSwitchInputMode() and preOpenInputWindow(). The altered code could connect to the Datalator at fancydata.com and enjoy full access to all the content if the altered code knew the server ID code, replaced by (...) in connect() statements. Not the absolute protection - the server ID could be extracted from the decompiled applet class.

Well we'll take our chances - the applet is just an example itself anyways. Besides we can always resort to granting the temporary credentials for those who deserve it. Or to degrade to plain HTML illustration without working example. Worth a try.

```
package datalator.user;

import datalator.layers.*;
import datalator.layers.menu.MenuLayer;
import datalator.layers.sprav.*;
import datalator.layers.text.HtmlLayer;
import datalator.layers.text.TextLayer;
import java.awt.Color;
import java.awt.event.ActionEvent;

import javax.swing.*;
import java.util.*;
import java.awt.event.ActionListener;

public class FancyDat extends User implements ActionListener{
    public static void setUserCode(){user = new FancyData();}

    JTextArea    messageArea        = null;
    JTextArea    forumMessageArea    = null;
    JScrollPane  areaScrollPane      = null;
    SpravLayer   contactUs           = null;
    JButton      submitButton        = null;
    String       webPageParameter    = "";

    public boolean preStartConnectionDialog(){
        setWaitCursor();
        webPageParameter = getWebPageParameter();
        if(webPageParameter.equals("forums")) {
            if(!connect(8081, 8082, "fancydata.com", "FancyData 1.0", "guest", "guest",
true, "...")){
                setDefaultCursor();
                return false;
            }
        }
        else{
            if(!connect(8081, 8082, "fancydata.com", "FancyData 1.0", "guest", "guest",
false, "...")){
                setDefaultCursor();
                return false;
            }
        }

        showRecordLockPeopleWorkingBelow(false); // do not show the icon
"peopleworkingBelow"
        if(webPageParameter.equals("contact")) serveContactUsPage();
        if(webPageParameter.equals("howtos"))  serveHowTosPage();
    }
}
```

```

        if(webPageParameter.equals("forums"))    serveForumPage();
        setDefaultCursor();
        return false;
    }

    public void serveHowTosPage(){
        if(getUserName().equals("guest")){
            MenuLayer chaptersMenu = enterVisibleMenu(null, "Chapters");
        }
    }

    public void serveForumPage(){
        if(getUserName().equals("guest")){
            MenuLayer mainMenu      = enterInvisibleMenu(null, "Main Menu");
            setActiveMenuRecord(mainMenu, "Start Discussion");
            SpravLayer discussionSprav = enterVisibleSprav(mainMenu, "Discussion");
        }
    }

    public void serveContactUsPage(){
        if(getUserName().equals("guest")){
            MenuLayer mainMenu = enterInvisibleMenu(null, "Main Menu");
            contactUs = enterInvisibleSprav(mainMenu, "Contact Us");
            openInput(contactUs);
            Vector fields = getInputFields(contactUs);
            InputField lastField = (InputField)fields.lastElement();
            lastField.button.setVisible(false);
            lastField.label.setVisible(true);
            Layer inputLayer = getActiveLayer();
            inputLayer.add(lastField.label);

            messageArea = new JTextArea(5, 5);
            messageArea.setFont                (inputLayer.font);
            messageArea.setWrapStyleWord (true);
            messageArea.setEditable(true);
            messageArea.setFocusable(true);
            areaScrollPane = new JScrollPane(messageArea);
            areaScrollPane.setVerticalScrollBarPolicy
(JSScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
            areaScrollPane.setHorizontalScrollBarPolicy(JSScrollPane.HORIZONTAL_SCROLLBAR
AS_NEEDED);
            areaScrollPane.setBounds(10, 120, inputLayer.dx-20, inputLayer.dy-170);
            inputLayer.add(areaScrollPane);
            setLayerMainWindowCaption(inputLayer, "Submit Your Message", null, null);
            submitButton = (JButton)getCustomComponentByName("Submit", inputLayer);
            submitButton.addActionListener(this);
        }
    }

    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();
        if(command.equals("Post It")){
            InputLayer inputLayer = (InputLayer)getActiveLayer();
            SpravLayer invisibleOwner = getInputLayerOwner(inputLayer);

            String name      = getInputField(invisibleOwner, "Name").getText();
            String email     = getInputField(invisibleOwner, "Email").getText();
            String subject   = getInputField(invisibleOwner, "Subject").getText();
            String message = forumMessageArea.getText();
            if(name.isEmpty() || email.isEmpty() || subject.isEmpty() ||
message.isEmpty()){
                setLayerMainWindowCaption(inputLayer, "All fields have to be filled
in...", null, Color.red); // back color
            }
        }
    }

```

```

        return;
    }

    HtmlLayer htmlLayer = (HtmlLayer)getLayerByName("Forum Messages");
    StringBuilder text = new StringBuilder(getTextLayerText(htmlLayer));
    int i = text.indexOf("</body>");
    String newText = "<p style=\"background-color:green;\"><font
style=\"color:white; font:bold\">"+formatPresentDate()+" "+formatPresentTime()+"
"+name+" "+email+" "+subject+"</font></p>"+message;

    text.insert(i-1, newText);
    setTextLayerText(htmlLayer, text.toString());
    getTextLayerSaveButton(htmlLayer).setVisible(false);

    String[] answers = saveTextLayerText(htmlLayer);
    escape(inputLayer);

    SpravLayer threads = (SpravLayer)getLayerByName("Threads");
    SpravRecord threadsRecord = getActiveRecord(threads);
    String replies = getField(threads, threadsRecord, "Replies");
    if(replies.isEmpty()) replies = "0";
    int newReplies = Integer.parseInt(replies) + 1;
    setField(threads, threadsRecord, "Replies", ""+newReplies);
    replaceSpravRecordAndWait(threads, threadsRecord);

    SpravLayer discussion = (SpravLayer)getLayerByName("Discussion");
    SpravRecord discussionRecord = getActiveRecord(discussion);
    String posts = getField(discussion, discussionRecord, "Posts");
    if(posts.isEmpty()) posts = "0";
    int newPosts = Integer.parseInt(posts) + 1;
    setField(discussion, discussionRecord, "Posts", ""+newPosts);
    String lastPost = getField(discussion, discussionRecord, "Last Post");
    setField(discussion, discussionRecord, "Last Post", formatPresentDate()+"
"+formatPresentTime()+" "+name);

    replaceSpravRecordAndWait(discussion, discussionRecord);
}
if(command.equals("Post Message")){
    MenuLayer mainMenu = enterInvisibleMenu(null, "Main Menu");
    setActiveMenuRecord(mainMenu, "Discussion Helpers");
    SpravLayer forumReplays = enterInvisibleSprav(mainMenu, "Forum Replays");
    openInput(forumReplays);

    Layer inputLayer = getActiveLayer();
    Vector fields = getInputFields(forumReplays);
    InputField lastField = (InputField)fields.lastElement();
    lastField.button.setVisible(false);
    lastField.label.setVisible(true);
    inputLayer.add(lastField.label);

    forumMessageArea = new JTextArea(5, 5);
    forumMessageArea.setFont          (inputLayer.font);
    forumMessageArea.setWrapStyleWord (true);
    forumMessageArea.setEditable(true);
    forumMessageArea.setFocusable(true);
    JScrollPane fromAreaScrollPane = new JScrollPane(forumMessageArea);
    fromAreaScrollPane.setVerticalScrollBarPolicy
(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
    fromAreaScrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCRO
LLBAR_AS_NEEDED);
    fromAreaScrollPane.setBounds(10, 120, inputLayer.dx-20, inputLayer.dy-170);

```

```

        inputLayer.add(froumAreaScrollPane);
        setLayerMainWindowCaption(inputLayer, "Submit Your Message", null, null);
        JButton postItButton = (JButton)getCustomComponentByName("Post It",
inputLayer);
        postItButton.addActionListener(this);
    }
    if(command.equals("Submit")){
        InputLayer inputLayer = (InputLayer)getActiveLayer();

        String name      = getInputField(contactUs, "Name").getText();
        String email      = getInputField(contactUs, "Email").getText();
        String subject    = getInputField(contactUs, "Subject").getText();
        String message    = messageArea.getText();
        if(name.isEmpty() || email.isEmpty() || subject.isEmpty() ||
message.isEmpty()){
            setLayerMainWindowCaption(inputLayer, "All fields have to be filled
in...", null, Color.red); // back color
            return;
        }

        SpravRecord messageRecord = createSpravRecord(contactUs);

        setField(contactUs, messageRecord, "Date", formatPresentDate());
        setField(contactUs, messageRecord, "Time", formatPresentTime());
        setField(contactUs, messageRecord, "Name", name);
        setField(contactUs, messageRecord, "Email", email);
        setField(contactUs, messageRecord, "Subject", subject);

        submitButton.setVisible(false);

        setWaitCursor();
        setLayerMainWindowCaption(inputLayer, "Sending...", null,
Color.green.darker()); // back color

        long recordId = appendSpravRecord(contactUs, messageRecord);
        saveTextContentAndWait("Messages", "Contact Us", recordId, message);

        setDefaultCursor();

        setLayerMainWindowCaption(inputLayer, "You message has been sent! Thank
you", null, Color.green.darker()); // back color
    }
}

public boolean preSwitchInputMode(SpravLayer spravLayer, int editMode){
    if(getUserName().equals("guest")) return false;
    return true;
}

public void postEnterLayer(Layer layer, SpravRecord spravRecord){
    if(layer.layerName.equals("Threads") && getUserName().equals("guest")){
        HtmlLayer htmlLayer = (HtmlLayer)getLayerByName("Forum Messages");
        JButton modeButton = getHtmlLayerModeButton(htmlLayer);
        modeButton.setVisible(false);
        JButton postMessage = new JButton("Post Message");
        postMessage.setLocation(modeButton.getLocation());
        postMessage.setSize(100, 20);
        htmlLayer.add(postMessage);
        postMessage.addActionListener(this);

        SpravLayer threads = (SpravLayer)getLayerByName("Threads");

```

```

        SpravRecord threadsRecord = getActiveRecord(threads);
        String replies = getField(threads, threadsRecord, "Views");
        if(replies.isEmpty()) replies = "0";
        int newReplies = Integer.parseInt(replies) + 1;
        setField(threads, threadsRecord, "Views", ""+newReplies);
        replaceSpravRecordAndWait(threads, threadsRecord);
    }
    Layer activeLayer = getActiveLayer();
    if(getUserName().equals("guest") && (activeLayer.layerType.equals("Text
Layer") || activeLayer.layerType.equals("Html Layer"))){
        setTextLayerButtonsVisible((TextLayer)activeLayer, false);
    }
}

public boolean preOpenInputWindow(SpravLayer spravLayer){
    if(getUserName().equals("guest") && spravLayer.layerName.equals("Discussion"))
return false;
    if(getUserName().equals("guest") && webPageParameter.equals("howtos")) return
false;
    return true;
}

public boolean preAddRecord(SpravLayer spravLayer, SpravRecord spravRecord){
    if(getUserName().equals("guest") && spravLayer.layerName.equals("Threads")){
        if(spravRecord.content[0].isEmpty()) return false;
    }
    return true;
}

public void postAddRecord(SpravLayer spravLayer, SpravRecord spravRecord){
    if(getUserName().equals("guest") && spravLayer.layerName.equals("Threads")){
        escape(spravLayer.inputLayer);
    }
}

public boolean preImageWindowClicked(String imageId){
    if(getUserName().equals("guest")) return false;
    return true;
}
}

```

Happy Datalating!