

5

Autocompletion

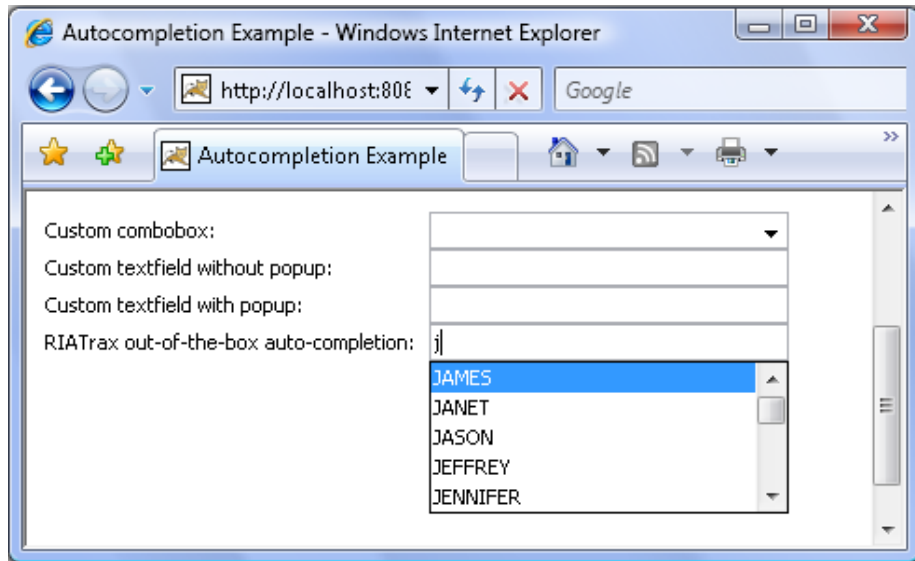
Modern web applications use complex and sophisticated AJAX functionalities to add better user experience. However, AJAX applications suffer from serious problems such as browser compatibility issues, are hard to understand and difficult to maintain. In short, they are too complex compared to the benefit they provide.

With Riatrax4Applet, adding AJAX like functionality is as easy as adding standard client-server communication to the application. This example shows how to implement auto-completion as the user types some value into a text field. Riatrax4Applet provides a built-in implementation. Just use the following method on any text component: `Autocomplete.decorate`

This chapter shows three alternative implementations with slightly different behavior.

5.1 Existing Solutions

Some solutions for auto-completion exist already, namely Glazed Lists, SwingX, JIDE and Laf-Widget. However, they are only applicable if the set of auto-complete proposals is strongly limited because they allow to set the list of available suggestions at initialization time only. For large-scale applications with a potentially large number of suggestions, downloading the whole list from the server is not efficient.



In this example, we show how to implement auto-completion in three text components: a combo box, a text field without a popup and a text field with a popup. What they all have in common is the fact that on each key stroke, the server is invoked to retrieve proposals that match the input.

5.2 Server Implementation

As a first step, we draft an interface offering at least one method to retrieve a list of proposals based on a string input:

```
public interface IAutoCompletionServer {
    public List<String> getProposals (String enteredString, int max);
}
```

Next, we create an implementation of the interface:

```
public class AutoCompletionServer implements IAutoCompletionServer {
    // A very trivial proposal cache:
    private String[] proposals = null;

    public AutoCompletionServer () {
        this.proposals = new String[] {
            "James",
            "Janet"
            // and more...
        };
    }

    public List<String> getProposals (String enteredString, int max) {
        List<String> results = new ArrayList<String>();
        int remaining = max;

        if (enteredString != null && enteredString.length() > 0) {
```



```

        String lowerCaseString = enteredString.toLowerCase();
        for (String proposal : getProposals()) {
            if (proposal.startsWith(lowerCaseString)) {
                remaining--;
                results.add(proposal);
                if (remaining == 0) {
                    break;
                }
            }
        }
        return results;
    }

    private String[] getProposals() {
        return proposals;
    }
}

```

The server is very simple: We iterate over an array of strings using Java's `startsWith` method to find matches. We can set the maximal number of proposals to be returned.

5.3 Generic Auto-Completion Interface for Components

The interface for the server is independent of the widgets that are using an autocompletion service. An implementation can offer several methods like `getCountryProposals`, `getNameProposals`, `getProductProposals` and so forth. It is therefore necessary to define an interface on which concrete components operate on:

```

public interface AutoCompletionService {
    public List<String> getProposals (String enteredString, int max);
}

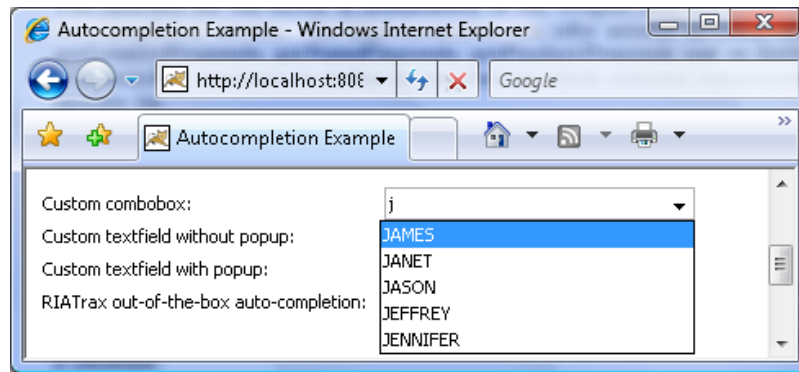
```

The interface offers a method to retrieve proposals. It does not need to know about servers and it does not necessarily have to communicate with servers. For example, an implementation of the interface could be an instance accessing a database.

5.4 Auto-Completion for JComboBox

In order to transparently add auto-completion capabilities to a combo box, we design a class extending `JComboBox` that adds the required functionalities.





```

public class AutoCompleteComboBox extends JComboBox implements KeyListener {
    private AutoCompletionService autoCompletionService = null;
    private int maxResults = -1;

    public AutoCompleteComboBox(AutoCompletionService service, int maxResults) {
        super();
        setAutoCompletionService(service);
        setMaxResults(maxResults);
        setVerifyInputWhenFocusTarget(false);
        setEditable(true);
        getEditor().getEditorComponent().addKeyListener(this);
    }
}

```

Note that an implementation of `AutoCompletionService` is an argument of the constructor. The last statement installs a key listener to the editor (text field) of the combo box that reacts on user input:

```

public void keyReleased(KeyEvent e) {
    final JTextField editor = (JTextField) getEditor().getEditorComponent();
    char charPressed = e.getKeyChar();
    int charCodePressed = e.getKeyCode();
    if (charCodePressed == KeyEvent.VK_DELETE
        || charPressed == KeyEvent.CHAR_UNDEFINED) {
        return;
    }
    if (charCodePressed == KeyEvent.VK_ENTER) {
        return;
    }

    String input = editor.getText();
    if (charCodePressed == KeyEvent.VK_UP
        || charCodePressed == KeyEvent.VK_DOWN
        || charCodePressed == KeyEvent.VK_KP_DOWN
        || charCodePressed == KeyEvent.VK_KP_UP) {
        editor.setText(input);
        return;
    }
}

```



```

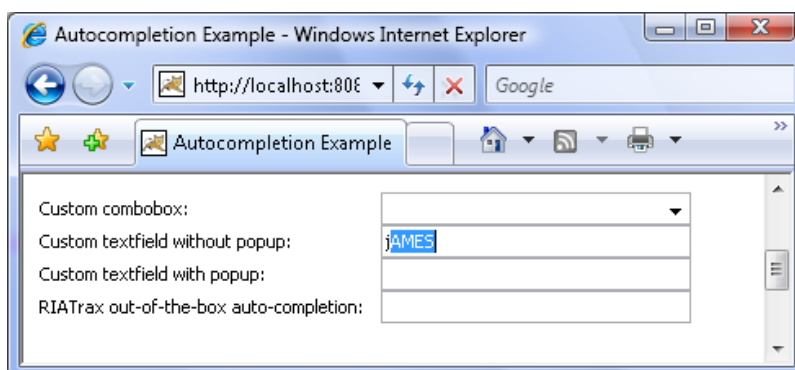
List<String> proposals = getAutoCompletionService().
getProposals(input, getMaxResults());
if (proposals.size() > 0) {
    ComboBoxModel model = new DefaultComboBoxModel(proposals.toArray());
    setModel(model);
    setPopupVisible(true);
    editor.setText(input);
} else {
    setPopupVisible(false);
}
}

```

The major part of the code just deals with special characters like enter, delete, up or down. At the end, the proxy is invoked to retrieve the proposals on the user input. If there are any proposals, the combo box model is updated and the popup is shown, otherwise the popup is hidden.

5.5 Auto-Completion for JTextField

As for the combo box, the text field can also be extended to support auto-completion. Yet, the text field is a little bit more complex because auto-completion can either take place in the text field itself or yield a popup with a list of suggestions. Moreover, while the popup component is part of JComboBox, it must be fully re-implemented for JTextField using a JList and a JPopupMenu.



```

public class AutoCompleteTextField extends JTextField implements KeyListener
    private JList list = null;
    private JPopupMenu popup = null;
    private AutoCompletionService autoCompletionService = null;
    private int maxResults = -1;
    private boolean showPopUp = false;

    public AutoCompleteTextField(AutoCompletionService proxy,
        maxResults, boolean showPopUp) {
        setAutoCompletionService(proxy);
        setMaxResults(showPopUp ? maxResults : 1);
        setShowPopUp(showPopUp);

        if (showPopUp) {

```



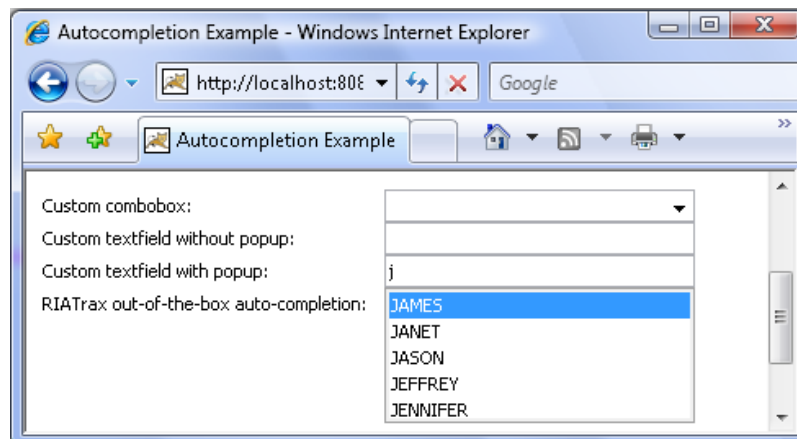
```

        setList(new JList());
        getList().setFocusable(false);
        getList().setBorder(null);
        setPopup(new JPopupMenu());
        getPopup().setLightWeightPopupEnabled(true);
        getPopup().setLayout(new BorderLayout());
        getPopup().add(getList());
        getPopup().addKeyListener(this);
    }

    addKeyListener(this);
    setVisible(true);
}
}

```

The creation of the `JPopupMenu` and the `JList` is only necessary if the creator wishes to have a popup for the text field. Finally, a key listener is installed on the text field.



```

public void keyReleased(KeyEvent e) {
    if (isShowPopUp()) {
        switch (e.getKeyCode()) {
            case KeyEvent.VK_DOWN:
                getList().setSelectedIndex(getList().getSelectedIndex()+1);
                break;
            case KeyEvent.VK_UP:
                getList().setSelectedIndex(getList().getSelectedIndex()-1);
                break;
            case KeyEvent.VK_ENTER:
                if (getPopup().isShowing()) {
                    this.setText(getList().getSelectedValue().toString());
                }
                break;
            case KeyEvent.VK_ESCAPE:
                getPopup().setVisible(false);
                break;
            default: {

```



```
int index = -1;
getList().removeAll();
List<String> proposals = getAutoCompletionService().
getProposals(getText(),
getMaxResults());
getList().setVisibleRowCount(proposals.size());
if (proposals.size() > 0) {
    getList().setListData(proposals.toArray());
} else {
    getPopup().setVisible(false);
}
if (!"".equals(getText()) && getList().getModel().getSize() >
    index = getList().getNextMatch(getText(),
    0, Position.Bias.Forward);
}
if (index != -1) {
    getList().setSelectedIndex(index);
    getPopup().show(this, 0, getHeight());
} else {
    getPopup().setVisible(false);
}
Rectangle rect = getList().getCellBounds(0, proposals.size())
int width = (int)getBounds().getWidth();
int height = (int)rect.getHeight() * proposals.size();
getPopup().setPopupSize(width, height + 3);
requestFocus();
}
}
if (getList().getSelectedValue() != null
    && getList().getSelectedValue().equals(getText())) {
    getPopup().setVisible(false);
}
} else {
    char charPressed = e.getKeyChar();
    int charCodePressed = e.getKeyCode();
    if (charCodePressed == KeyEvent.VK_DELETE
        || charPressed == KeyEvent.CHAR_UNDEFINED) {
        return;
    }
    if (charCodePressed == KeyEvent.VK_ENTER) {
        int length = getText().length();
        setSelectionStart(length);
        setSelectionEnd(length);
        setCaretPosition(length);
    }
    if (getSelectionStart() != getSelectionEnd()) {
        setText(getText().substring(0, getSelectionStart()));
    }
}
```



```

String input = getText();
List<String> proposals =
    getAutoCompletionService().getProposals(input, 1);
if (proposals.size() > 0) {
    setText(input + proposals.get(0).substring(input.length()));
    setSelectionStart(input.length());
    setSelectionEnd(getText().length());
}
}
}

```

The first part of the key pressed function attends to the popup window, while the second part directly operates on the text field. Both have in common that they first deal with a bunch of special characters before they invoke `getAutoCompletionService().getProposals(...)` to get the available suggestions.

5.6 Usage of `AutoCompleteComboBox` and `AutoCompleteTextField`

Given the server interface, the generic service interface and the concrete implementations, they can all be used in one applet:

```

public class AutoCompletionClient extends RiatraxApplet {

    @Services(allocation = Allocation.Global,
              implementation = AutoCompletionServer.class)
    protected static IAutoCompletionServer server;

    public void initContent() {
        GridBagConstraints labelConstraints = /* some initializations */
        GridBagConstraints controlConstraints = /* some initializations */

        AutoCompletionService service = new AutoCompletionService() {
            @Override
            public List<String> getProposals(String enteredString, int max) {
                return server.getProposals(enteredString, max);
            }
        };

        JLabel comboLabel = new JLabel("Combobox:");
        JComboBox comboBox = new AutoCompleteComboBox(service, 5);
        labelConstraints.gridy++;
        controlConstraints.gridy++;
        add(comboLabel, labelConstraints);
        add(comboBox, controlConstraints);

        JLabel label = new JLabel("Textfield without popup: ");
        JTextField customTextField = new AutoCompleteTextField

```



```
        (service, 1, false);
        labelConstraints.gridy++;
        controlConstraints.gridy++;
        add(label, labelConstraints);
        add(customTextField, controlConstraints);

        JLabel extendedLabel = new JLabel("Textfield with popup:");
        AutoCompleteTextField extendedField = new AutoCompleteTextField(
            service, 5, true);

        labelConstraints.gridy++;
        controlConstraints.gridy++;
        add(extendedLabel, labelConstraints);
        add(extendedField, controlConstraints);
    }
}
```

An important part is the creation of an instance of `AutoCompletionService` which invokes the server internally. This instance is passed to the combo box and both text fields.

To conclude, we have seen in this chapter that adding smart AJAX-like functionalities to a rich client application is very easy and offers the convenience of step by step debugging of Java code on both, the server and client side.



