IBM

Rich Ajax Platform, Part 2: Developing applications

Web 2.0, the Eclipse way

Chris Aniszczyk, Software Engineer, IBM, Software Group
Benjamin Muskalla (bmuskalla@innoopract.com), Software Engineer, Innoopract Informationssysteme GmbH

Summary: The Rich Client Platform (RCP) is a powerful platform technology to build enterprise applications. With the help of Rich Ajax Platform (RAP), it gets more interesting because you can reuse your existing code base and development skills for a Web application, as you saw in Part 1 of this "Rich Client Platform" series. Additionally, RAP has some noteworthy features, making Web development even more attractive. The article goes beyond the Hello World example, and explains some key concepts and how to use advanced features provided by RAP.

Date: 11 Dec 2007
Activity: 1206 views
Comments: 0 (Add comments)

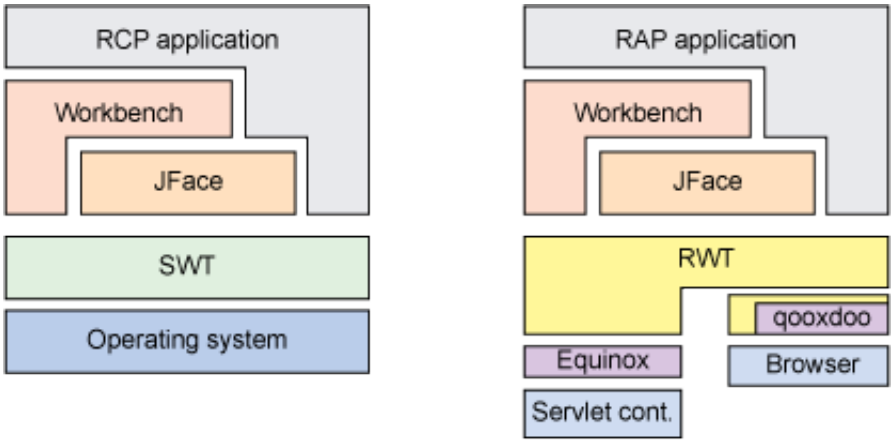
Average rating

Before diving into new features, we would like to provide a little overview of how RAP is structured and why this is important. After that, we will go over to some interesting topics like background jobs, user-interface (UI) customization, and we'll tell you a bit about more advanced topics, such as custom widgets and themes. Return to Part 1 or see Resources if this discussion is beyond your present grasp of RAP.

The architecture of RAP

To develop applications based on RAP efficiently, there are cases where you need to know how it *works*. We don't want to bore you with too much about the internal mechanisms, but we need to explain some basic concepts to aid your development with RAP. The following illustration is a good starting point into how RAP is organized in contrast to RCP.

Figure 1. RAP in contrast to RCP



As you can see, RAP is split into two parts. On the one side, we have the *server-side* piece that runs on top of Equinox (the Eclipse implementation of the OSGi specification). And on the other side, we have the *client* piece, which is what you see in a browser. Both sides exchange events, which the UI updates appropriately. This means that most of the code is being executed on the server, and the *thin-client* side is only updated when needed. This has the advantage of running clean Java™ applications without having the need to install the application itself on the client machine.

RAP is not RCP

Structuring your application to live in both worlds

Even though RAP isn't RCP, it's possible to share a lot of code between a RAP and RCP application. To do this, we recommend having one plug-in that serves as a base plug-in that contains UI code that can be used in RAP and RCP. To accomplish this, we recommend you use Import-Package in your MANIFEST.MF to specify dependencies. This will allow your base plug-in to use packages whether they come from RAP or RCP. You aren't coupling yourself to a specific implementation. After you have your base plug-in set up, we recommend two more plug-ins, one for RAP-specific code, the other for RCP-specific code. Finally, having two workspaces, where one has a target platform set up for RAP and the other target platform for RCP. In the future, we hope this becomes easier as the tooling evolves to accommodate application developers writing code to run in RAP and RCP.

The provocative title of this section shouldn't scare you, but it is a fact you need to be aware of. As we saw in Part 1, the migration of an RCP application to a Web-enabled RAP counterpart is straightforward. While developing RCP applications, it is generally the case that only one user will work with the application at a time. In contrast, a RAP application that runs on a server will be used by several (if not thousands of) users at the same time. This leads to rethinking some concepts used in RAP application development. One interesting aspect is the singularity of classes

implementing the singleton pattern. A singleton is unique in the so-called "application scope." We distinguish between several scopes in a RAP application:

Application scope
The application scope is the most far-reaching scope in that it's available for every user. A singleton is unique for the whole application and, therefore, lives in the application scope. This means that all users of the application are using the same instance of the singleton. This could be a good thing, but not always if the instance holds some user-specific information.

Session scope
The session scope is bound to the current user session available for the current user only. To implement a singleton bound only to a specific session, you can use the class `SessionSingletonBase` provided by RAP. By extending this class and overriding its `getInstance` method, the class behaves like a singleton, but is available session-wide only.

Request scope
The request scope is the smallest of the three. It's available only when a request is being processed. Most of the time, this scope isn't really relevant for the normal developer (except when you want to take part of RAP's life-cycle concept).

Background jobs

One of the most exciting features of RAP is support for the Eclipse Jobs framework. With the help of jobs, you can run long-running tasks in the background while the UI is still responsive. To see how it works, we will extend our mail application from Part 1, with a new action that schedules a job (see Listing 1).

Listing 1. Adding an action (ApplicationActionBarAdvisor.java)

```
....
private Action progressAction;
...
protected void makeActions(final IWorkbenchWindow window) {
    ...
    progressAction = new SampleProgressAction();
                progressAction.setText("Count me!");
                progressAction.setId("progress.action");
                register(progressAction);

}
...
protected void fillCoolBar(ICoolBarManager coolBar) {
...
```

```
    toolbar.add(progressAction);
}
```

In addition to the jobs API, RAP offers a well-known UI for handling jobs, as we will see later. To activate this feature in the workbench, we tell Eclipse to show the progress indicator (see Listing 2).

Listing 2. Enabling the progress indicator (ApplicationWorkbenchWindowAdvisor.java)

```
...
public void preWindowOpen() {
...
    configurer.setShowProgressIndicator(true);
}
...
```

The most interesting part comes next: the *job*. Our example is a simple implementation (see Listing 3), which just increases a variable to a specific amount and waits after every step so we can see something happening in the UI.

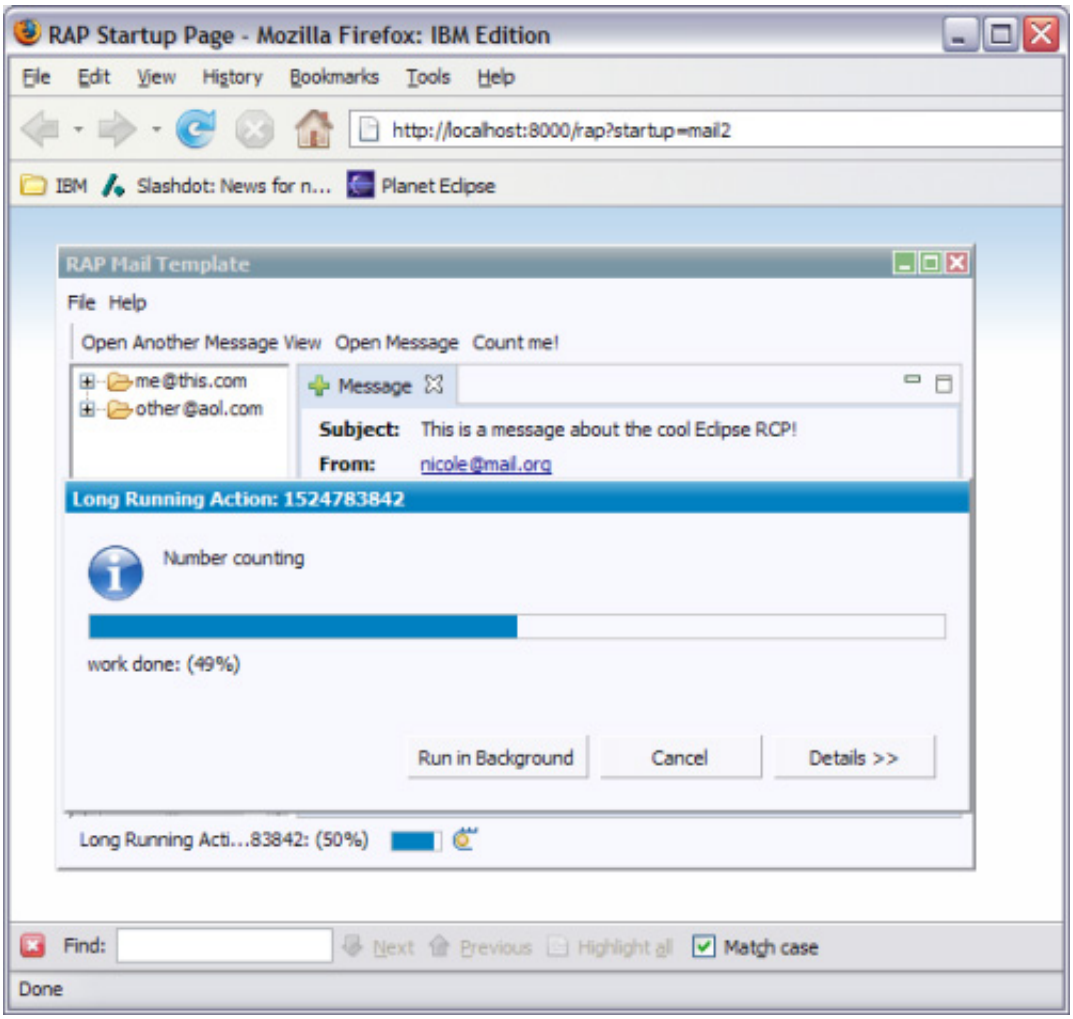
Listing 3. The job itself (SampleProgressAction.java)

```
package rap.mail;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.core.runtime.jobs.Job;
import org.eclipse.jface.action.Action;

public class SampleProgressAction extends Action {

        private static final int TASK_AMOUNT = 100;

        public void run() {
                Job job = new Job("Long Running Action:") {
                        protected IStatus run(final IProgressMonitor monitor) {
                                monitor.beginTask("Number counting", TASK_AMOUNT);
                                for (int i = 0; i < TASK_AMOUNT; i++) {
                                        if (monitor.isCanceled()) {
                                                monitor.done();
                                                return Status.CANCEL_STATUS;
```

```
                                        }
                                        int done = i % TASK_AMOUNT;
                                        monitor.subTask("work done: (" + done + "%)");
                                        monitor.worked(1);
                                        try {
                                                Thread.sleep(200);
                                        } catch (InterruptedException e) {
                                                e.printStackTrace();
                                        }
                                }
                                monitor.done();
                                return Status.OK_STATUS;
                        }
                };
                job.setName(job.getName() + " " + job.hashCode());
                job.setUser(true);
                job.schedule();
        }
}
```

After starting the application and clicking the action, a new dialog appears to show us the progress of the currently running job. If we don't want to wait for the job to finish, we can hide it by clicking **Run in background**. Furthermore, it is possible to open the Progress View by clicking on the progress icon in the status bar. If there are more jobs running at the same time, we can easily see them in the Progress View.

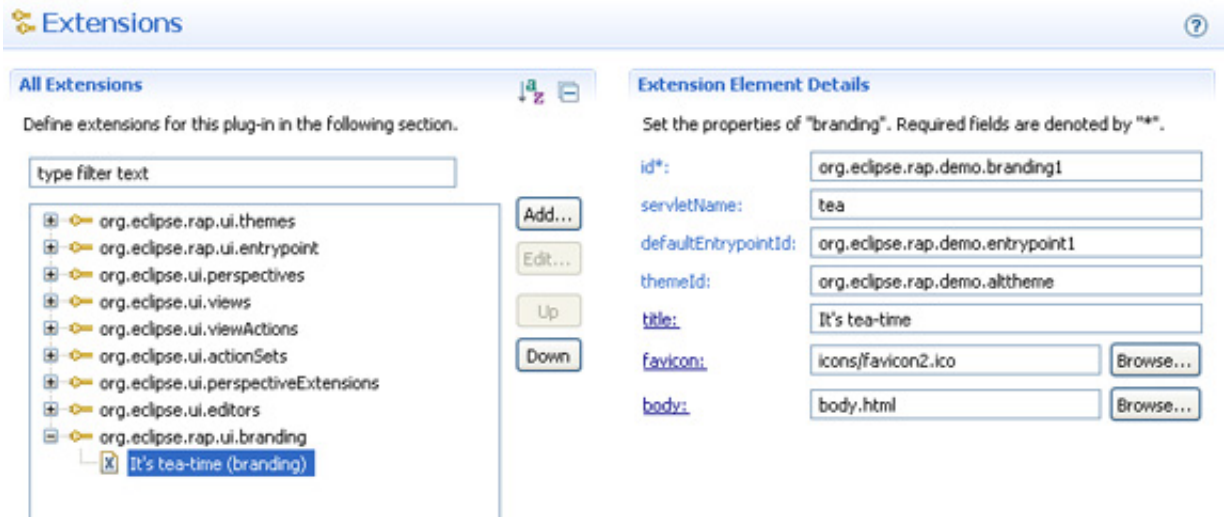
Figure 2. Background jobs



Branding

For most RCP-based and RAP-based applications, there is a need to *brand* them to your liking. Branding means to modify the presentation of your application to fit your intentions, like conforming to an existing corporate branding. If you ever worked on an RCP application, there's a concept of `.product` files to help brand your application. Since RAP targets another platform (the Web), we need to do some additional work to customize our application.

Figure 3. Manifest Editor with sample branding

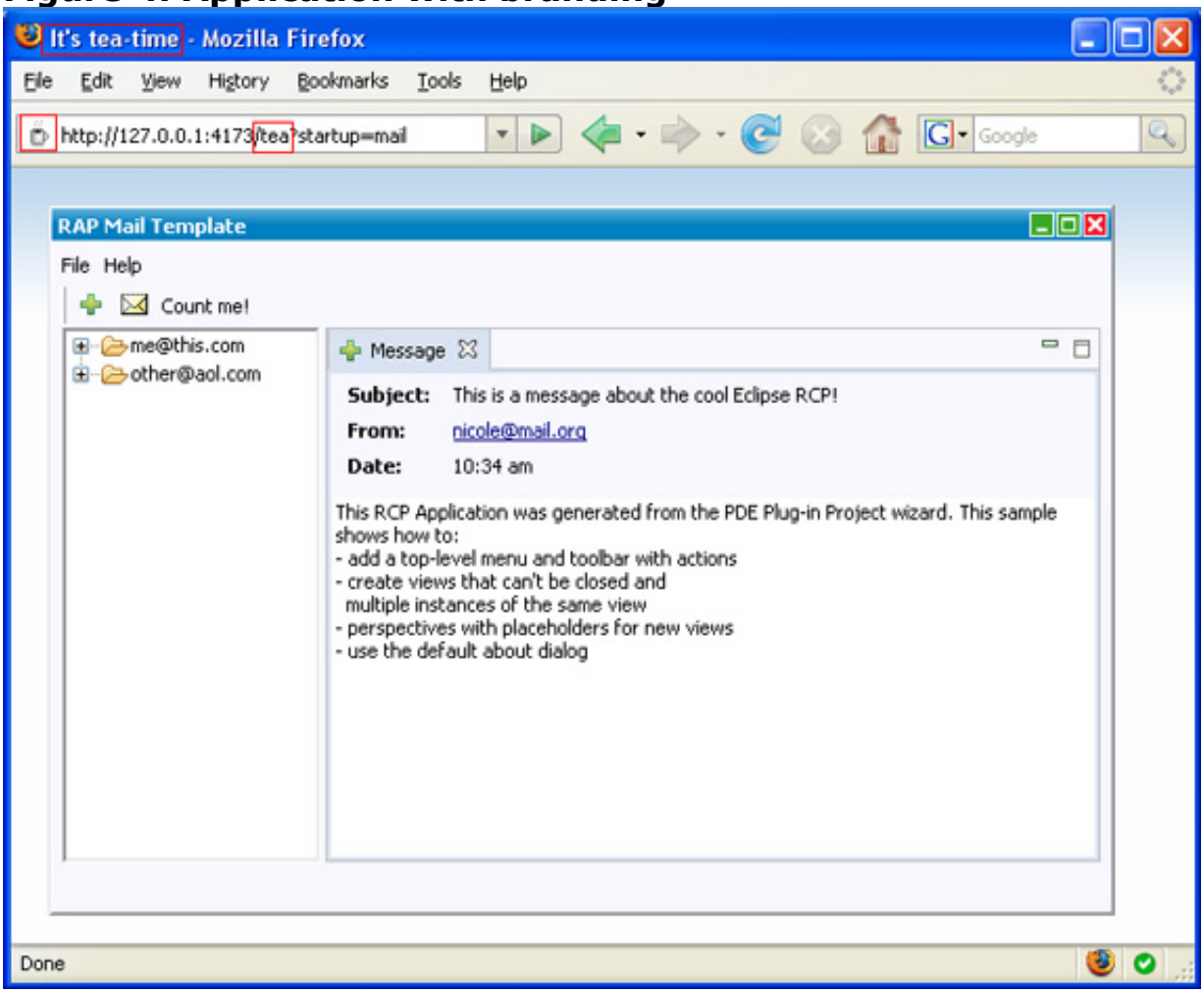


As you can see, there are several ways to customize an RAP application.

Attribute	Description
servletName	The `servletName` defines the name of the servlet the application will be available to the outer world. This means the URL of your application will be shown as http://<host><port>/<servletName>
defaultEntrypointId	As you normally have one entry point that should be loaded by default when accessing the application without the `startup` parameter, you can define `defaultEntrypointId`.
themeId	The theme to be used for this branding can be specified here. You need to provide an ID for the theme defined by an extension of `org.eclipse.rap.ui.themes`. Without the `themeId`, the RAP standard theme will be used.
title	The title of the page — the part that will be shown as window title in most browsers or the title of the tab — can be customized with this attribute.
favicon	The `favicon` is the icon shown in the address bar of the browser. This has to be a valid .ico file to be displayed in the browser.
body	If you have the need for some custom (X)HTML markup in the RAP index page, you can register a valid (X)HTML file with the `body` attribute that will be included in the body tag of the generated index page.
exitConfirmation	If not empty, the string in `exitConfirmation` will be shown to the user if he wants to leave the RAP application. This is useful to prevent users to accidentally close the browser window/tab.

After restarting the application with a new branding, we can see the result.

Figure 4. Application with branding



Themes

The theme of a RAP application defines the look and feel your application users will have. Don't look at it like the theme support in the RCP workbench, but more like the support of your favorite operation system to define the look of windows, buttons, etc. In general, RAP theming allows you to define:

- Custom colors
- Custom fonts
- Custom borders
- Custom dimensions and box dimensions (paddings etc.)
- Custom images and icons

However, which aspects that are actually customizable depends on the particular control you're theming. Some controls allow for more flexibility, while others are more hard-wired. As an example, the difference between the default look and feel,

and a custom theme could look like Figure 5.

Figure 5. Custom theme in action

Creating a theme

In RAP, theme files are simple Java property files. A template named theme-template.properties can be found in the src/ directory of the RAP org.eclipse.rap.rwt plug-in. You only have to specify those properties that are relevant for your customization (undefined properties stay at their default value). See below for the syntax of the property values. The following is an example for a simple theme file.

Listing 4. A simple theme file

```
# Alternative Demo Theme
#

# Frame border for group boxes
# default: 1 solid #aca899
group.frame.border: 2 dotted #56a0ea

# Font for title bar of Shells
# default: bold 11 "Segoe UI", Corbel, Calibri, Tahoma, "Lucida Sans Unicode", sans-serif
shell.title.font: bold 15 "Trebuchet MS", Arial, Helvetica, sans-serif

# Height of the title bar of Shells
# default: 18
shell.title.height: 25

...
```

Registering the theme

To make your theme available, you have to register it with the extension point `org.eclipse.rap.swt.themes`. In the plugin.xml of your application project, add an

extension like Listing 5.

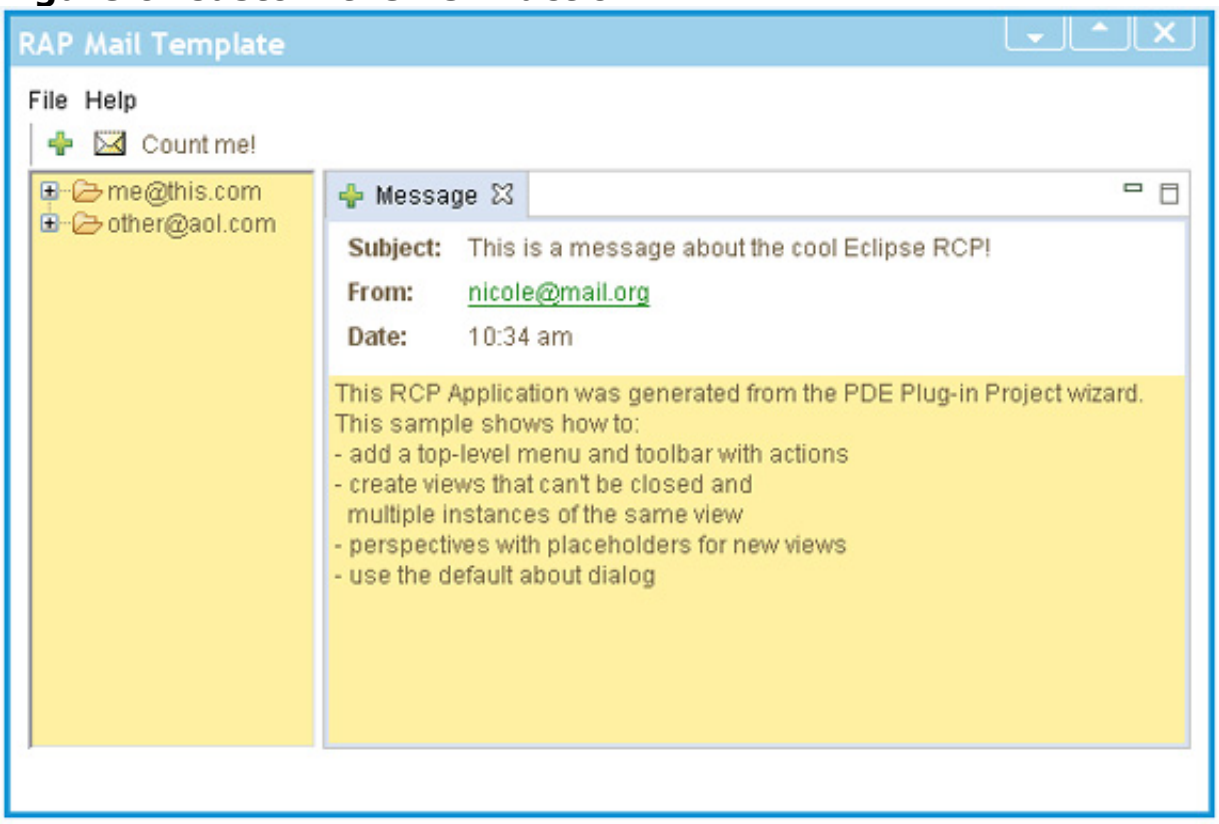
Listing 5. Making a theme available

```
<extension
       point="org.eclipse.rap.swt.themes">
    <theme
          id="org.eclipse.rap.dw.demotheme"
          name="Alternative Demo Theme"
          file="theme1/theme.properties" />
</extension>
```

Activating the theme

To activate the theme, we have to specify the `themeId` in our branding. We set it to the ID of your newly created theme: `org.eclipse.rap.demo.alttheme`. After restarting the application, we can proudly see our application in a new style.

Figure 6. Custom theme in action



Custom widgets

Structuring your application to live in both worlds

The Google Web Toolkit and RAP are similar in that they both allow you to use Java technology to code Rich Internet Applications. The big differences are that GWT is running on the client vs. RAP, which is mainly running on the server. Since RAP is running on the server, it allows you to access the full Java API and make use of the famous Eclipse plug-in model via OSGi. Another way to think about this is in Eclipse terms, GWT is like a stand-alone SWT application (just a widget toolkit) where RAP enabled an RCP-style approach for Web applications.

First things first: There are two types of custom widgets in RAP. You can develop widgets by combining existing widgets to form a new one, or you can develop real custom widgets, which are also known as *owner-drawn* widgets. When talking about a compound widget, there is no difference to SWT (RCP) and the server-side (RAP). Most of the time, you should be able to just reuse existing compound widgets in a RAP application.

A more complex case is to create a real custom widget. When developing widgets with RAP, you have to be aware that we have two different run times: the *server-side* and the *client-side* implementation. On the server, the widget is the facade with public API consumed by the users of your custom widget. On the client, the real implementation is done with JavaScript. The trick is in bridging the client and server parts. RAP has an internal concept known as a *life cycle*, where every widget is taken into account. The life cycle is responsible for the initialization and exchange of data between the client and server of the widget. The implementation of custom widgets is a bit beyond the scope of this article, and we recommend reading more about custom widgets in the RAP help documentation (see Resources). Just to get an idea of what's possible, Figure 7 is a screenshot of Google Maps encapsulated in an SWT widget, which can be accessed like every other widget.

Figure 7. Google Maps RAP widget

Conclusion

In Part 1, we saw how easy it is to turn existing RCP applications into real Web applications. Here, we see that RAP is an exciting technology that offers many customizations for Web development. By using the power of the Rich Client Platform (RCP), Equinox on the server side and the Web browser, RAP enables everyone to run applications with no configuration. Furthermore, RAP has the interesting trait of allowing Eclipse developers to reuse existing skills and world-class tools to create Web applications. In the end, we see RAP embracing the best of both desktop and Web browser worlds.

Download

Description	Name	Size	Download method
Sample code	os-eclipse-richajax2-mail-project.zip	38KB	HTTP

Information about download methods

Resources

Learn

- Start with "Rich Ajax Platform, Part 1: An introduction" for an introduction to

RAP, how to set up a RAP development environment, and some demos and examples.

- Learn more about and download Rich Ajax Platform (RAP).
- Check out the Eclipse Foundation's RAP wiki to learn about new features and advanced use cases.
- See the Eclipse Foundation's RAP Theming wiki to learn more about themes.
- Read the RAP project official help documentation to learn more about RAP.
- See Eclipse Foundation's RAP custom widget help documentation to learn more about building a custom widget.
- Check out the Eclipse Corner for more information about the Eclipse Jobs API.
- Browse all the Eclipse content on developerWorks.
- New to Eclipse? Read the developerWorks article "Get started with Eclipse Platform" to learn its origin and architecture, and how to extend Eclipse with plug-ins.
- Expand your Eclipse skills by checking out IBM developerWorks' Eclipse project resources.
- To listen to interesting interviews and discussions for software developers, check out check out developerWorks podcasts.
- For an introduction to the Eclipse platform, see "Getting started with the Eclipse Platform."
- Stay current with developerWorks' Technical events and webcasts.
- Watch and learn about IBM and open source technologies and product functions with the no-cost developerWorks On demand demos.
- Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Download IBM product evaluation versions, and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

- The Eclipse Platform newsgroups should be your first stop to discuss questions regarding Eclipse. (Selecting this will launch your default Usenet news reader application and open eclipse.platform.)
- The Eclipse newsgroups has many resources for people interested in using and extending Eclipse.
- Participate in developerWorks blogs and get involved in the developerWorks community.

About the authors

Chris Aniszczyk is an Eclipse committer at IBM Lotus who works on OSGi-related development. His primary focus these days is improving Eclipse's Plug-in Development Environment (PDE) and spreading the Eclipse love inside of IBM's Lotus organization. He is an open source enthusiast at heart, specializing in open source evangelism. He evangelizes about Eclipse in his blog, and he's honored to represent the Eclipse committers on the Eclipse Foundation's board of directors. He's always available to discuss open source and Eclipse over a frosty beverage.

Benjamin Muskalla works as a software developer and consultant at Innoopract Informationssysteme in Karlsruhe. He is a committer on the Rich Ajax Platform (RAP)

project, mostly responsible for the workbench implementation. He is also an active contributor to the Eclipse Platform.