



Dreamteam

OSGi als Webserver

Markus Bach

Was verbirgt sich hinter dem Schlagwort „Server-Side-Eclipse“? Lassen sich Eclipse-Technologien gewinnbringend für serverseitige Anwendungen nutzen? Der folgende Artikel zeigt anhand eines Beispiels, wie man einen Eclipse-basierten Server für Webanwendungen entwickelt und darauf deployed. Zum Einsatz kommt dabei Equinox, ein von der Eclipse Foundation entwickeltes, quelloffenes, Java-basiertes Framework, welches die OSGi-Kernspezifikation implementiert.

► Im Bereich der Desktop-Anwendungen hat die Eclipse-RCP bereits unbestreitbar ihren Siegeszug eingeläutet. Bei größeren Anwendungsszenarien bleibt es aber meist nicht bei nur einem Desktop, auf dem man arbeitet, und die Applikationen sollen am besten noch mit einem Backend kommunizieren. Dank der vollen Java-Mächtigkeit auf dem Client ist es keine unüberwindliche Hürde, auf entfernte Dienste oder Webservices zuzugreifen. Dabei kommt es aber zum Technologie-Bruch: Das Server-Backend musste bisher ein klassischer JEE-Server à la JBoss sein, egal wie groß die Anforderungen an den Server wirklich sind.

Die Idee zur Lösung des Dilemmas ist simpel: Client-Software kann mit Hilfe der Eclipse-RCP entwickelt werden und der Server einfach auch. Dies hat nicht nur den Charme der einheitlichen Technologie, sondern darüber hinaus noch den unmittelbaren Vorteil, dass die Geschäftslogik auf dem Server aus den gleichen Bundles stammen kann, die auch auf dem Client eingesetzt werden. Damit ließen sich für bestimmte Anwendungsfälle Fat Clients bauen, die vieles selbst machen, aber auch Thin Clients, bei denen die Bundles mit der Geschäftslogik auf dem Server laufen und die Dienste „remote“ als Webanwendung anbieten.



Abbildung 1 zeigt eine solche Architektur. Auf diese Weise lässt sich eben auch die Geschäftslogik aus den Bundles in Webanwendungen nutzbar machen und es muss für die Webclients keine Logik mehrfach implementiert werden, um sie auf einem Webserver deployen zu können.

Eigentlich war schon immer alles da ...

So könnte man wohl sagen, denn im Bauch der Eclipse-Generation 3 werkelt schon immer ein Webserver, um das Hilfesystem, das aus HTML-Dateien besteht, darstellen zu können. Zunächst wurde Tomcat installiert, mittlerweile handelt es sich aber um Jetty [Jetty]. Dieser Server steht Tomcat in nichts nach und kann neben den rein statischen HTML-Seiten selbstverständlich auch JSPs und Servlets hosten.

In den frühen Versionen der Eclipse konnte man den Server aber nicht ohne weiteres für andere Zwecke verwenden. Erst seit dem Europa-Release von Eclipse werden nun alle nötigen Bundles ausgeliefert, mit denen man den Server komfortabel benutzen kann. Damit lassen sich anspruchsvolle Webanwendungen realisieren, denen alle Möglichkeiten der Eclipse-RCP zur Verfügung stehen und die natürlich auch selbst als abgeschlossenes Bundle daherkommen.

Vorbereitung des Servers

Da der eigentliche Server-Container bereits Teil der Eclipse-RCP ist, gestaltet sich die Inbetriebnahme denkbar einfach – allerdings nicht ganz ohne Fallstricke, handelt es sich doch eher um ein „Hidden Feature“.

Es empfiehlt sich, für den Server ein separates Bundle anzulegen – dies ist kein Muss, ermöglicht aber eine saubere Trennung der restlichen Anwendung vom Webserver. Erledigt wird dies, wenn man mit dem Wizard (*File – New – Project... – Plug-In-Project*) ein neues Plug-In-Projekt erzeugt. Im Wizard können alle vorgeschlagenen Einstellungen übernommen werden. Auf der zweiten Seite ist allerdings wichtig, dass die Option *Generate an activator* gesetzt ist, denn dann wird eine Klasse angelegt, mit deren Methoden ins Start- und Stoppverhalten des Bundles eingegriffen werden kann. Außerdem sollte im Wizard noch die Option *Rich Client Application* auf *No* gesetzt werden.

Um aus dem neuen Bundle nun einen vollwertigen Webserver zu machen, müssen noch einige Bundles der Plattform eingebunden werden. Diese Abhängigkeiten werden im Manifest definiert. Abbildung 2 zeigt den Dependencies-Dialog im Manifest-Editor. Neben dem Bundle `org.eclipse.core.runtime` werden folgende Bundles aus dem Equinox-Umfeld benötigt,

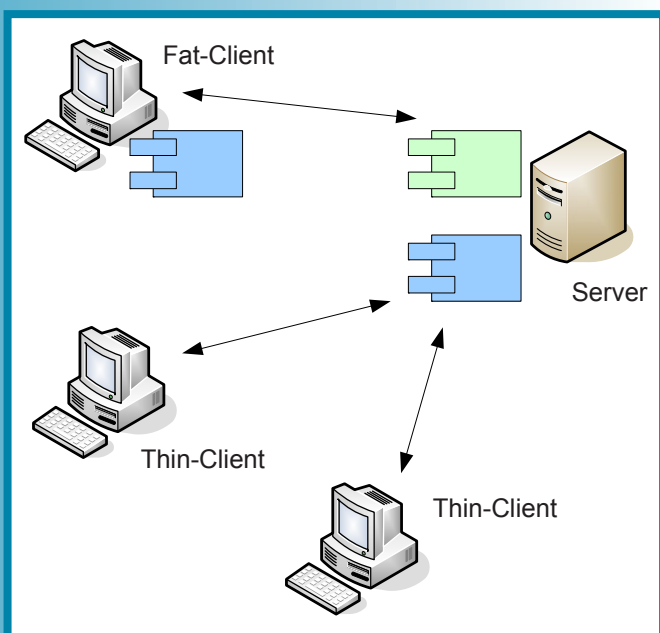


Abb. 1: Client und Server mit Hilfe der Eclipse-RCP

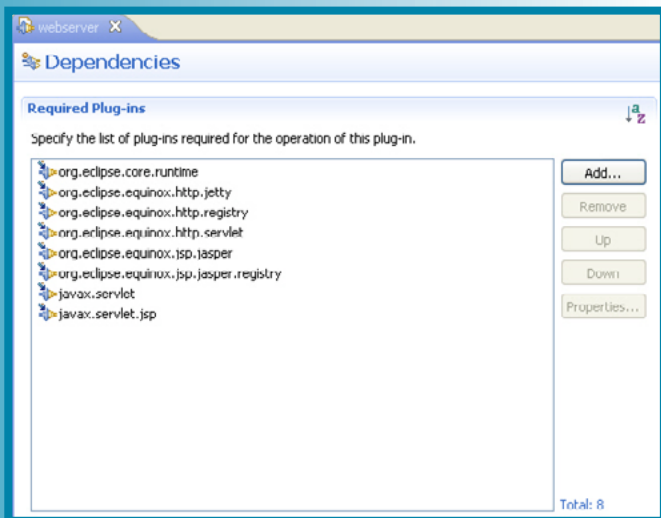


Abb. 2: Dependencies-Dialog im Manifest-Editor

die zur Steuerung und Konfiguration des Servers gebraucht werden:

- ▼ org.eclipse.equinox.http.jetty,
- ▼ org.eclipse.equinox.http.registry,
- ▼ org.eclipse.equinox.http.servlet,
- ▼ org.eclipse.equinox.jsp.jasper,
- ▼ org.eclipse.equinox.jsp.jasper.registry.

Damit Servlets und JSP-Seiten übersetzt werden können, braucht man zusätzlich `javax.servlet` und `javax.servlet.jsp`.

Die vom Projekt-Wizard angelegte Klasse `Activator` enthält die zunächst leeren Methoden `start` und `stop`. Diese werden von der Workbench aufgerufen, wenn das Bundle gestartet bzw. gestoppt werden soll. In unserem Fall müssen die Methoden entsprechend gefüllt werden, damit sie den Jetty-Server korrekt starten und wieder beenden. Dies ist gewissermaßen auch die einzige Schwierigkeit und der Bruch im Eclipse-Konzept, bei dem Bundles normalerweise automatisch geladen und gestartet werden, sobald sie gebraucht werden. Beim Jetty-Server muss dies leider von Hand erledigt werden.

Listing 1 zeigt die Implementierung der Klasse `Activator`: In den Methoden werden zunächst Referenzen auf die beiden Bundles, die manuell gestartet werden müssen, erzeugt und durch Aufruf ihrer `Activator`-Methoden werden die Bundles explizit gestartet bzw. später wieder gestoppt.

Normalerweise lauscht der Jetty-Server gleich auf dem Port 80. Ist für die Anwendung ein anderer Port erforderlich, kann die Port-Nummer natürlich auch geändert werden. Bewerkselligt wird dies über den folgenden VM-Parameter, der beim Start der Anwendung angegeben wird und beispielsweise auf den Port 8080 umschaltet: `-Dorg.eclipse.equinox.http.jetty.http.port=8080`.

Statische Inhalte

Auf einem Server sind stets statische Inhalte am unproblematischsten, verursachen sie doch kaum Belastung und erfordern auch keine Compiler oder Interpreter. So ist dies natürlich auch bei diesem Webserver. Die HTML-Seiten und sonstigen Ressourcen können in mehreren Anwendungskontexten auf dem Server deployed werden. Die Trennung hierbei ist wesentlich sauberer möglich, als dies bei anderen Webservern der Fall ist, da die Webanwendungen in Bundles ver-

```
public class Activator extends Plugin {
    public static final String PLUGIN_ID = "webserver";
    private static Activator plugin;

    public Activator() {
    }

    public void start(BundleContext context) throws Exception {
        super.start(context);
        plugin = this;

        Bundle jettyBundle = Platform
            .getBundle("org.eclipse.equinox.http.jetty");
        Bundle httpRegistryBundle = Platform
            .getBundle("org.eclipse.equinox.http.registry");
        try {
            jettyBundle.start();
            httpRegistryBundle.start();
        } catch (BundleException e) {
            e.printStackTrace();
        }
    }

    public void stop(BundleContext context) throws Exception {
        plugin = null;
        super.stop(context);

        Bundle jettyBundle = Platform
            .getBundle("org.eclipse.equinox.http.jetty");
        Bundle httpRegistryBundle = Platform
            .getBundle("org.eclipse.equinox.http.registry");
        try {
            httpRegistryBundle.stop();
            jettyBundle.stop();
        } catch (BundleException e) {
            e.printStackTrace();
        }
    }

    public static Activator getDefault() {
        return plugin;
    }
}
```

Listing 1: Die Klasse `Activator`

packt sind und nicht etwa nur auf verschiedene Verzeichnisse verteilt sind.

In der Datei `plugin.xml` der Beispielanwendung muss zunächst ein Kontext angelegt werden, der definiert, wo die Dateien zu finden sind. Listing 2 zeigt den entsprechenden Eintrag zur Erweiterung des Extension-Points `org.eclipse.equinox.http.registry.httpcontexts`. Über den Wert des Attributs `id` kann der Kontext später referenziert werden, wenn das öffentliche Mapping definiert wird. Mit dem Attribut `path` des Elements `resource-mapping` wird der Pfad zum Wurzelverzeichnis der Webanwendung angegeben. Dieser Pfad ist absolut zur ersten Ebene des Bundles, in dem die Dateien liegen. Zusätzlich könnte mit einem Attribut `bundle` noch angegeben werden, in welchem Bundle die statischen Inhalte liegen. Lässt man es weg, werden die Ressourcen im aktuellen Bundle gesucht.

Die öffentliche URL, über welche die Dateien im Verzeichnis `/src/webfiles/statical` angesprochen werden, wird über die

```
<extension point="org.eclipse.equinox.http.registry.httpcontexts">
    <httpcontext id="html">
        <resource-mapping path="/src/webfiles/statical"/>
    </httpcontext>
</extension>
```

Listing 2: Extension-Point für den Kontext statischer Inhalte



```
<extension point="org.eclipse.equinox.http.registry.resources">
  <resource alias="/webApp" httpcontextId="html"/>
</extension>
```

Listing 3: Extension-Point für das URL-Mapping auf den Kontext

Erweiterung des Extension-Points `org.eclipse.equinox.http.resources` angelegt. Listing 3 zeigt den vollständigen Eintrag, der das Mapping der URI `/webApp` auf den Kontext mit dem obigen Pfad herstellt. Die komplette URL der statischen Inhalte lautet somit: `http://localhost:80/webApp/index.html`.

JSPs und Servlets

Ähnlich wie für die statischen Seiten wird auch für JSP-Seiten erst einmal ein Kontext benötigt, der angibt, wo die JSP-Dateien auf dem Server liegen. Dies leistet auch hier der Extension-Point `org.eclipse.equinox.http.registry.httpcontexts`, dessen Erweiterung Listing 4 zeigt. Über den Wert von Attribut `id` wird später der Pfad innerhalb des Bundles zu den JSP-Dateien referenziert. Hierzu ist anzumerken, dass es zwar laut Definition des Extension-Points möglich sein sollte, mehrere Elemente vom Typ `httpcontext` für den Extension-Point anzugeben. Allerdings hat dies bei mir nicht funktioniert, weshalb der Extension-Point zweimal erweitert werden musste, um zwei Kontexte zu registrieren.

```
<extension point="org.eclipse.equinox.http.registry.httpcontexts">
  <httpcontext id="jsp">
    <resource-mapping path="/src/webfiles/dynamical"/>
  </httpcontext>
</extension>
```

Listing 4: Extension-Point für den Kontext dynamischer Inhalte

Nachdem der Kontext definiert wurde, muss noch das Mapping vom öffentlichen Aufruf zum Kontext vorgenommen werden. Dies wird über den Extension-Point `org.eclipse.equinox.http.registry.servlets` gemacht. Innerhalb der Extension werden JSP-Seiten genauso wie Servlets behandelt, d. h. dass die JSPs über das Element `servlet` an einen Kontext gebunden werden, der angibt, wo die Dateien liegen. Bei den JSP-Dateien hat man den Vorteil, dass nicht jede JSP-Seite separat genannt werden muss, sondern der Wildcard `*.jsp` genügt. Außerdem wird als implementierende Klasse eine JSP-Factory angegeben, die sich automatisch um das Laden der JSP-Seiten kümmert. Listing 5 zeigt die Erweiterung des Extension-Points für JSP-Seiten und für ein Servlet. Statt eines Kontextes ist für ein Servlet nur das eindeutige Mapping der Servlet-URL auf die implementierende Klasse notwendig.

```
<extension point="org.eclipse.equinox.http.registry.servlets">
  <servlet alias="/webApp/*.jsp"
    class="org.eclipse.equinox.jsp.jasper.registry.JSPFactory"
    httpcontextId="jsp">
  </servlet>
  <servlet alias="/webApp/Login"
    class="webfiles.servlets.Login">
  </servlet>
</extension>
```

Listing 5: Extension-Point für das URL-Mapping auf Servlets

Und jetzt die Beispielanwendung

Als Beispiel soll eine kleine Anwendung dienen, die auf einer statischen HTML-Seite einen Anmeldedialog implementiert und die Eingaben an ein Servlet schickt. Das Servlet prüft die Daten und leitet entweder mit einer Fehlermeldung an den Anmeldedialog zurück, falls Benutzername und Passwort nicht passen, oder es leitet an einen Begrüßungsdialog weiter, wenn die Anmeldung erfolgreich war.

Der HTML-Quelltext wird in Listing 6 gezeigt und die fertige Seite im Browser in Abbildung 3. Daran ist zu beachten, wie der Aufruf des Servlets im Attribut `action` des Tags `form` definiert ist.

```
<html>
  <head>
    <title>RCP-Webserver-Test</title>
  </head>
  <body>
    <h3>Anmelde-Dialog</h3>
    <form action="/webApp/Login" method="post">
      Name: <input type="text" name="name"><br/>
      Passwort: <input type="password" name="password">
      <input type="submit" value="Anmelden">
    </form>
  </body>
</html>
```

Listing 6: HTML-Code des Login-Dialogs

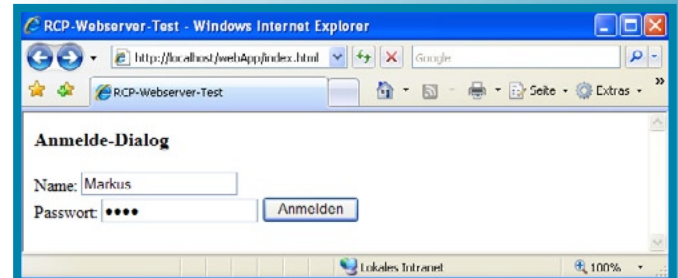


Abb. 3: RCP-Webserver-Test

```
public class Login extends HttpServlet {
  @Override
  protected void doPost(HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, IOException {

    String name = (String) req.getParameter("name");
    String password = (String) req.getParameter("password");

    if (password.equals("1234")) {
      RequestDispatcher dispatcher = getServletContext()
        .getRequestDispatcher("/webApp/start.jsp");
      dispatcher.forward(req, res);
    } else {
      PrintWriter out = res.getWriter();

      out.println(
        "<html><head><title>RCP-Webserver-Test</title></head><body>");
      out.println("<h3>Anmeldung fehlgeschlagen</h3><br/>");
      out.println("<a href='\"index.html\"'>Zurück</a>");
      out.println("</bod></html>");
    }
  }
}
```

Listing 7: Implementierung des Login-Servlets

Die Klasse des Login-Servlets befindet sich im Package `webfiles.servlets`, ihre Implementierung ist in Listing 7 zu sehen. Das Servlet reagiert nur auf POST-Requests; aus dem Request werden die Parameter aus dem HTML-Formular mit Benutzernamen und Passwort extrahiert und geprüft. Ist die Prüfung positiv, wird der Request an die Willkommenseite, die als JSP realisiert ist, weitergeleitet. Im Falle einer negativen Prüfung generiert das Servlet eine einfache HTML-Antwort, die eine Fehlermeldung enthält.

Die JSP-Seite mit dem Begrüßungsdialog ist in Listing 8 zu sehen. Neben dem eingegebenen Benutzernamen zeigt sie das aktuelle Datum und die Uhrzeit an.

```
<html>
<head>
<title>RCP-Webserver-Test</title>
</head>
<body>
<h3>Hallo <%= (String) request.getParameter("name") %></h3>
Es ist jetzt: <%= new java.util.Date() %>
</body>
</html>
```

Listing 8: JSP-Seite mit Begrüßung bei erfolgreicher Anmeldung

Fazit

Der Ansatz, Webanwendungen auch auf Basis der Eclipse-Plattform zu entwickeln, ist quasi nur der nächste logische Schritt, den die Plattform auf ihrem Siegeszug machen musste. Beispielsweise existiert mit der Rich-Ajax-Plattform [RAP] bereits seit einiger Zeit die Möglichkeit, komplexe SWT-Anwendungen einfach in Webanwendungen zu transformieren.

Ein alternativer Weg wurde nun in diesem Artikel aufgezeigt. Er eignet sich insbesondere für erfahrene Webentwick-

ler, da das Programmierkonzept der Oberfläche erhalten bleibt und weiterhin JSPs und Servlets verwendet werden – dies natürlich professioneller als beispielhaft im Artikel gezeigt. Die gängigen Frameworks wie Struts oder Java-Server-Faces lassen sich hierfür problemlos einsetzen, da Equinox einen vollwertigen Web-Container anbietet.

Die gezeigte Architektur ermöglicht, dass Webanwendungen als Bundles gekapselt werden können. Damit geht einher, dass gekapselte Logik mit den etablierten Eclipse-Mechanismen für andere Webanwendungen aber auch für RCP-Anwendungen verfügbar ist bzw. bestehende Logik von RCP-Anwendungen als Webanwendung angesprochen werden kann.

Alles in allem ist die Idee, Eclipse auch auf dem Server zu verwenden, absolut gelungen, auch wenn an manchen Stellen weitere Detailverbesserungen nötig sind. Insbesondere muss die Konfiguration des Jetty-Servers noch verfeinert und erleichtert werden.

Links

[EquinoxServ] Server-Side Equinox,

<http://www.eclipse.org/equinox/server/>

[Jetty] Jetty WebServer, <http://www.mortbay.org/>

[RAP] Rich Ajax Platform, <http://www.eclipse.org/rap/>



Markus Bach ist bei der Hydrometer GmbH als Java-Entwickler tätig und beschäftigt sich schwerpunktmäßig mit Software-Engineering und -Architekturen.
E-Mail: bach.markus@gmx.de.