Orientation in Objects

Entwicklung eigener Vaadin-Komponenten

Tutorial: Vaadin 7 um eigene Widgets erweitern mit der Hilfe von GWT und HTML5

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

AUTOR

Roland Krüger
Orientation in Objects GmbH

Veröffentlicht am: 1.3.2013

ABSTRACT

Unter den zahlreichen Frameworks für Rich Internet Applications (RIAs) auf dem Markt erfreut sich das aus Finnland stammende Vaadin Toolkit [1] zunehmender Beliebtheit. Ein einfaches Programmiermodell zusammen mit einem großen Angebot flexibler UI-Komponenten ermöglichen eine reibungslose und zügige Entwicklung von webbasierten RIA-Anwendungen. Wer unter den angebotenen Komponenten eine bestimmte Funktionalität vermisst, kann für Vaadin sehr leicht eigene Erweiterungen auf Basis des Google Web Toolkits (GWT) entwickeln. Dieser Artikel beschreibt die dazu erforderlichen Schritte und stellt das notwendige Hintergrundwissen bereit. Es wird gezeigt, wie man sich die Möglichkeiten von GWT in Verbindung mit HTML5 zunutze machen kann, um performancekritische Teile seiner Vaadin Applikation zu beschleunigen oder sein Komponentenrepertoire um moderne Funktionen zu erweitern.

Der folgende Artikel basiert auf der Version 7 des Vaadin Toolkits. Gezeigt wird die Entwicklung einer eigenen Vaadin-Komponente, die bestimmte Features von HTML5 und des Google Web Toolkits verwendet.

Der Beispielcode für das beschriebene Projekt kann auf GitHub [6] heruntergeladen und ausprobiert werden.

Orientation in Objects GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.de info@oio.de

EINLEITUNG

Während das Vaadin Toolkit zwischen den beiden Platzhirschen JSF und GWT derzeit noch eher eine Außenseiterrolle einnimmt, weiß das Framework mit einem einfachen Programmiermodell und einer großen Zahl funktionsreicher UI-Komponenten zu überzeugen. Insbesondere seine Eigenschaft als serverseitiges Framework, das weitestgehend von dem Request-Response-Modell des HTTP-Protokolls abstrahiert, erlaubt eine Entwicklungsweise, die stark an die Oberflächenprogrammierung mit Swing oder SWT angelehnt ist. Diese Eigenschaft bringt neben vielen Vorteilen allerdings auch einige Nachteile mit sich. Die erhöhten Latenzzeiten bei Benutzeraktionen und der größere serverseitige Speicherverbrauch sind die zwei bedeutendsten. Nahezu jede Interaktion des Anwenders auf dem Client führt zu einem Datenaustausch mit dem Server. Dieser wiederum muss den kompletten Zustand sämtlicher Komponenten der Benutzeroberfläche einer Session lokal im Speicher verwalten. Man bekommt also auch hier, wie so oft im Leben, nichts geschenkt.

Was man dafür geschenkt bekommt, ist eine sehr mächtige und vielfältige UI-Komponentenbibliothek, die nur wenige Wünsche offen lässt. Unter der Haube verwendet Vaadin das Google Web Toolkit (GWT) als clientseitige Rendertechnologie. Damit stehen für den Client sämtliche Möglichkeiten offen, die man auch als GWT-Entwickler hat.

Der Mechanismus, mit dem das Vaadin Toolkit GWT-Komponenten in die eigene Architektur integriert, ist keine schwarze Magie und kann mit ein wenig GWT Kenntnis auch für eigene Erweiterungen genutzt werden. Die große Menge der verfügbaren Vaadin Addons aus der Community zeigt, dass das Schreiben eigener Vaadin Widgets kein Hexenwerk sein muss. Über diesen Weg hat man als Entwickler immer die Möglichkeit, in einer Anwendung eigene Erweiterungen gezielt an denjenigen Stellen einzusetzen, bei denen die Besonderheiten der Vaadin Architektur den Schuh am meisten drücken lässt.

Ein anschauliches Beispiel soll verdeutlichen, auf welche Art eine typische Vaadin-Anwendung durch die Netzwerklatenz unnötigerweise ausgebremst werden kann. Man stelle sich ein Textfeld vor, dessen Inhalt der Anwender über eine dazugehörige Schaltfläche leeren kann. Die Leeren-Funktionalität wird – wie in Vaadin-Anwendungen üblich – in einem `Button.ClickListener` für die Schaltfläche angestoßen. Drückt der Anwender auf diese Schaltfläche, wird dadurch eine Nachricht zum Server geschickt, die die Aufforderung enthält, doch bitte ein `Button.ClickEvent` auszulösen. Der Handler für diesen Event wiederum funkt dem Textfeld zurück, dass es seinen Inhalt leeren möge. Alles in allem hat man hier also eine Kommunikation zwischen zwei clientseitigen Komponenten (Schaltfläche und Textfeld), die über den Server als Vermittler abgewickelt wird. Das Ergebnis davon ist eine rein clientseitige Änderung. Bei der Verwendung der Vaadin-Standardkomponenten hat somit die Latenz des Netzwerks einen direkten Einfluss darauf, wie lange bestimmte Zustandsänderungen im Client dauern.

GWT hat derartige Probleme nicht. Ein GWT-Programm läuft als JavaScript-Anwendung komplett im Browser und muss den Server nur zum Datenaustausch mit den Backend-Services kontaktieren. Sehr schnelle Reaktionszeiten bei Benutzeraktionen sind also eine der Stärken des GWT.

Diese Eigenschaft kann man sich auch für Vaadin zunutze machen. Indem man mit Hilfe von GWT eigene, auf seine speziellen Anforderungen hin optimierte Vaadin-Komponenten entwickelt, kann man von den Vorteilen des einen Frameworks profitieren, um bestimmte Schwächen des anderen zu umgehen.

Neben seiner sehr netzwerkfreundlichen Eigenschaft als clientseitiges Framework bringt GWT noch einige weitere nützliche Funktionen mit, von denen man auch als Vaadin-Entwickler profitieren kann. Beispielhaft seien hier die GWT Client Bundles und bestimmte Features aus der HTML5 Spezifikation, wie z. B. der Local Storage, genannt. Mit Hilfe des Local Storages kann eine Webanwendung lokal im Client-Browser beliebige Daten speichern. Dies kann als Erweiterung des Cookie-Konzepts gesehen werden. GWT Client Bundles sind ein Mechanismus, mit dem Ressourcen (Bilder, CSS und beliebige andere Dateien) zusammengefasst und unter einer einheitlichen Schnittstelle aus GWT Code heraus zugreifbar gemacht werden. Die Daten bestimmter Ressourcen werden dabei direkt in den clientseitigen Applikationscode hineinkompiliert, so dass diese nicht separat geladen werden müssen. Sobald die GWT-Anwendung selbst geladen wurde, stehen auch die Ressourcen aus den Client Bundles im Browser zur Verfügung.

Grundsätzlich hat man auch mit Vaadin Zugriff auf den kompletten Funktionsumfang des Google Web Toolkits. Naturgemäß wird man dabei allerdings nicht sämtliche GWT Funktionen verwenden wollen, da sich einige Problemstellungen, die von diesen adressiert werden, mit Vaadin gar nicht ergeben. So wird man den GWT RPC Mechanismus (*Remote Procedure Calls*), mit dem die Client-Seite Funktionen auf dem Server aufrufen kann, nicht benötigen, da in einer Vaadin-Anwendung das Backend ohnehin nur serverseitig angesprochen wird.

Um die angesprochenen Konzepte an einem Beispiel zu demonstrieren, soll im Folgenden exemplarisch eine Vaadin-Komponente entwickelt werden, die einige der angesprochenen Ideen vereint. Es werden dabei die notwendigen Schritte zum Schreiben eigener Vaadin-Komponenten skizziert.

VERLUSTFREIES TEXTFELD MIT LEEREN-FUNKTION UND ICON

Die zu entwickelnde Komponente soll eine spezielle `TextArea` werden, die die folgende Funktionalität anbietet: Das Textfeld ist mit einer eigenen Schaltfläche verbunden, mit der sich der bisher eingegebene Text leeren lässt. Um den zuvor beschriebenen Umweg über den Server zu vermeiden, soll das Leeren des Textfeldes rein clientseitig funktionieren. Eine Verzögerung durch die Netzwerklatenz entfällt damit.

Der Reset-Button zum Leeren des Textfeldes soll mit einem eigenen Icon dekoriert werden, welches über den GWT Client Bundle-Mechanismus eingebunden wird. Dadurch wird diese Grafik gleich zusammen mit dem JavaScript-Code des Widgets geladen und muss nicht erst nachträglich vom Server abgerufen werden.

Die Text-Komponente soll verlustfrei sein. Das heißt, bisher eingegebener Text soll bei einem Browserabsturz oder nach dem Schließen des Browserfensters nicht verloren gehen. Bereits geschriebener aber noch nicht gespeicherter Text wird daher in regelmäßigen Abständen im lokalen Speicher des Browsers zwischengespeichert. Beim Laden der Seite wird der Inhalt des Textfeldes mit dem gesicherten Inhalt aus dem lokalen Browserspeicher initialisiert. Dadurch gehen keine Benutzereingaben verloren.

Die neue Komponente soll den Namen `ResettableTextArea` erhalten.

DAS HOPP DESIGN PATTERN

Die Grundidee des Vaadin Frameworks besteht darin, dass der Entwickler einer Webanwendung das clientseitige Verhalten der Anwendung über serverseitige Proxy-Objekte ansteuern kann. Er muss sich damit nicht mehr um netzwerkbezogene Belange kümmern, wie z. B. die Synchronisierung von Serveranfragen oder die Übermittlung der Frontend-Daten an den Client.

Hinter diesem Prinzip steckt das Design Pattern *Half Object Plus Protocol* (HOPP) [2]. Bei diesem Entwurfsmuster geht es darum, dass ein Objekt mit zwei unterschiedlichen Adressräumen interagieren muss. Man teilt dieses Objekt dafür auf zwei getrennte Implementierungen auf, die jede für sich in einem der beiden Adressräume lebt. Diese beiden "halben" Objekte synchronisieren sich dabei über ein vordefiniertes Protokoll.

Die Rollen der beiden Adressräume im Falle von Vaadin werden durch den Browser auf der einen und das Server-Backend auf der anderen Seite wahrgenommen. Eine Vaadin-Komponente, wie z. B. ein Textfeld, teilt sich immer in zwei Implementierungen auf: ein clientseitiges Widget und die dazugehörige serverseitige Komponente.

Die Kommunikation zwischen diesen beiden Objekten findet über ein auf JSON basierendes Protokoll statt. Zustandsänderungen der Komponenten auf dem Server werden den dazugehörigen Client Widgets über UIDL-Nachrichten (*User Interface Description Language [3]*) mitgeteilt, damit diese ihren Status entsprechend aktualisieren können. Wenn also beispielsweise durch eine Benutzeraktion die Beschriftung eines Labels geändert werden soll, weist der serverseitige Teil des Labels sein Gegenstück im Browser an, sich mit dem aktualisierten Text neu zu zeichnen.

Das Entwickeln eigener Vaadin-Komponenten besteht also im Kern immer aus zwei Hauptschritten: dem Schreiben eines clientseitigen Widgets und der Programmierung von dessen serverseitigem Gegenstück. Das clientseitige Widget kann mit dem Google Web Toolkit oder basierend auf einem vorhandenen Vaadin-Widget entwickelt werden. Es ist ebenfalls möglich, ein rein JavaScript-basiertes Widget zu schreiben. Sämtliche Möglichkeiten des GWTs stehen einem also offen.

Der serverseitige Teil einer eigenen Vaadin-Komponente fügt sich in die bekannte Klassenhierarchie von Vaadin ein. Hier kann auf die gesamte API von Vaadin zurückgegriffen werden.

DIE BESTANDTEILE

Bevor wir in die Implementierungsdetails eintauchen, wollen wir zunächst einen allgemeinen Blick auf die Bestandteile werfen, die wir für unsere neue Komponente benötigen. Eine Komponente von Vaadin besteht immer mindestens aus den folgenden drei Teilen:

- **Client Widget:** Die clientseitige GWT-Komponente, die später im Browser dargestellt wird.
- **Serverkomponente:** Das serverseitige Gegenstück, das in einer Vaadin-Anwendung als Repräsentant für das Client Widget verwendet wird.
- **Connector-Klasse:** Eine Klasse, die den Kommunikationskanal zwischen Client Widget und Serverkomponente definiert. Über den Connector als Mittler können die beiden Komponentenbestandteile miteinander interagieren.

Die Gesamtheit aller Vaadin-Komponenten wird in einem Widget Set zusammengefasst. Was es damit auf sich hat, wird weiter unten noch im Detail beschrieben.

Eine Neuerung von Vaadin 7 gegenüber älteren Versionen des Frameworks ist die Tatsache, dass ein clientseitiges Widget sich nicht mehr selbst um die Datensynchronisierung mit seinem Gegenstück auf dem Server kümmern muss. Es verhält sich nun mehr wie eine klassische GWT-Komponente ohne spezielle Anbindung an ein bestimmtes Backend-Framework. Damit wurden die Zuständigkeiten besser getrennt und die Kopplung verringert. Das Widget wird vollständig von außen gesteuert.

Die Aufgabe des Vermittlers zwischen Client- und Serverteil einer Vaadin-Komponente übernimmt das Connector-Objekt. Über dieses wird der gemeinsame Zustand der beiden Komponentenbestandteile synchronisiert und die Möglichkeit bereitgestellt, entfernte Methodenaufrufe jeweils auf Client und Server auszuführen. Jede Komponente ist daher mit einem eigenen Connector-Objekt verbunden. Der Connector verwaltet von außen den Zustand des ihm zugeordneten Client Widgets.

WIDGET SETS

Ein zentraler Begriff für das Vaadin Toolkit ist das Widget Set. Als Widget Set wird die Gesamtheit aller clientseitigen Komponenten bezeichnet, die von einer Vaadin-Anwendung verwendet werden kann. Sämtliche Komponenten, die das Framework von Hause aus mitbringt, werden in Vaadins `DefaultWidgetSet` definiert. Ein Widget Set ist im Kern nichts anderes als ein ganz normales GWT Programm mit dem dazugehörigen Einstiegspunkt (Entry Point). Im Unterschied zu einer herkömmlichen GWT-Anwendung stellt ein Vaadin Widget Set nur den JavaScript-Code der Vaadin Widgets und die dazugehörigen Kommunikationsmechanismen bereit. Ein Widget Set ist somit generisch und stellt noch keine eigenständig lauffähige Applikation dar.

Beschrieben wird ein Widget Set wie jedes andere GWT Programm über eine GWT-Modulbeschreibungsdatei. So findet man in der Datei `DefaultWidgetSet.gwt.xml` aus der Vaadin Bibliothek die GWT-Konfiguration und den GWT Entry Point von Vaadin selbst.

Möchte man neben den Vaadin-Standardkomponenten auch selbstgeschriebene Widgets oder Vaadin Addons verwenden, muss man ein eigenes Widget Set definieren, welches das Vaadin Widget Set erweitert. Dazu legt man eine neue GWT Modulbeschreibungsdatei an und bindet darin alle zusätzlichen Vaadin Module (z. B. Addons) ein, die man verwenden möchte. Zwingend erforderlich ist hierbei, von Vaadins `DefaultWidgetSet` zu erben.

Das folgende Listing zeigt die Modulbeschreibungsdatei `ResettableTextAreaWidgetset.gwt.xml` für unsere selbstgeschriebene Vaadin-Komponente. Hier wird lediglich das `DefaultWidgetSet` eingebunden (geerbt). Weitere GWT-spezifische Konfigurationen sind an dieser Stelle nicht notwendig. Wir erweitern damit den Standardkomponentensatz von Vaadin um unsere eigenen Komponenten.

```
<module>
   <inherits name="com.vaadin.DefaultWidgetSet" />
</module>
```

Anhand dieser Beschreibungsdatei kann der GWT Compiler nun das komplette Widget Set für unsere Anwendung kompilieren. Dieses besteht aus den Vaadin-Standardkomponenten und zusätzlich aus unserer eigenen Komponente. Er wird dazu den Quellcode des `DefaultWidgetSets` und den Code unserer Komponente heranziehen, um diese in den clientseitigen JavaScript-Code zu kompilieren. Der zu übersetzende Code wird von dem GWT Compiler standardmäßig in einem Package mit dem Namen `client` erwartet, welches parallel zu der Modulbeschreibungsdatei liegen muss. Dies kann man bei Bedarf anpassen. Für Details dazu sei auf die GWT Dokumentation verwiesen.

Damit eine Vaadin-Anwendung unser neues Widget Set - und damit unsere eigenen Komponenten - verwenden kann, muss die neue Modulbeschreibungsdatei in der `web.xml` der Anwendung bekannt gemacht werden. Das geschieht über den Servlet-Parameter `widgetset`.

```
<servlet>
   <servlet-name>ResettableTextArea Demo
   Application</servlet-name>
   <servlet-class>com.vaadin.server.VaadinServlet</servlet-class>
   ...
   <init-param>
      <description>Application widgetset</description>
      <param-name>widgetset</param-name>
      <param-value>
         de.oio.vaadin.ResettableTextAreaWidgetset
      </param-value>
   </init-param>
</servlet>
```

Damit weiß das Framework beim Anwendungsstart, welche GWT-Anwendung, sprich welches Widget Set, vom Browser geladen werden muss. Zu beachten ist hier, dass der Name des Widget Sets ohne die Dateiendung `gwt.xml` und mit dem korrekten Package-Pfad (hier `de.oio.vaadin`) angegeben wird.

DAS CLIENTSEITIGE VAADIN WIDGET

Wir beginnen mit der Programmierung des clientseitigen Vaadin Widgets mit Hilfe von GWT. Sämtliche Klassen, die wir für die Clientseite schreiben, werden wir unter einem Package `client` ablegen. Dadurch werden die Klassen für die Kompilierung durch den GWT Compiler bereitgestellt. Der GWT Compiler soll nur diejenigen Klassen nach JavaScript übersetzen, die auch tatsächlich auf Browserseite verwendet werden. Solche Klassen werden daher unterhalb eines bestimmten Packages erwartet – in unserem Fall das standardmäßige `client` Package.

Unser clientseitiges Widget soll den Namen `VResettableTextArea` erhalten. Wir halten uns damit an die Vaadin-Konvention, dass clientseitige Widget-Klassen immer ein V an den Klassennamen vorangestellt bekommen. Die Klasse erbt von `com.google.gwt.user.client.ui.Composite` und implementiert das Interface `com.vaadin.client.ui.Field`:

```
public class VResettableTextArea extends Composite implements
Field {
   public static final String CLASSNAME =
   "v-resettabletextarea";
   private VerticalPanel panel;
   private ResetButton resetButton;
   private TextArea textArea;
   private Storage localStorage;
   private String textAreaUID = "";
   // ...
}
```

Mit `com.google.gwt.user.client.ui.Composite` als Superklasse haben wir die Möglichkeit, mehrere GWT Widgets zu einem einzelnen, zusammenhängenden Widget zu vereinen. Das Interface `com.vaadin.client.ui.Field` ist ein Marker-Interface ohne eigene Methoden, welches dem Vaadin Framework nur mitteilt, dass hier eine Eingabekomponente vorliegt. Das Widget wird dadurch einheitlich zu den anderen Eingabekomponenten behandelt.

Wir definieren eine eigene CSS Klasse `v-resettabletextarea` für unser Widget, damit wir dessen Aussehen später mit Hilfe eines CSS Stylesheets anpassen können. Damit gehen wir konform zu der Vorgehensweise des Vaadin Frameworks, das ebenfalls jeder Komponente eine eigene CSS-Klasse spendiert.

Für den Aufbau unseres Textfeldes benötigen wir drei UI-Komponenten: Ein Layout vom Typ `com.google.gwt.user.client.ui.VerticalPanel`, um die beiden anderen Komponenten vertikal zueinander auszurichten. Wir können hier auf die Layout-Klassen des GWT zurückgreifen und müssen nicht das äquivalente Layout von Vaadin verwenden. Die zusätzliche Backend-Anbindung, die Vaadins `VerticalLayout` mitbringt, wird in unserem Fall nicht benötigt und würde das Widget nur unnötig aufblähen.

Weiter benötigen wir ein Textfeld zusammen mit einer Schaltfläche, die den Inhalt des Textfeldes zurücksetzt. Als Textfeld verwenden wir wieder eine GWT Klasse: `com.google.gwt.user.client.ui.TextArea`. Für die Schaltfläche schreiben wir eine eigene Klasse mit dem Namen `ResetButton`. Die Implementierung der Schaltfläche werden wir uns später genauer anschauen.

Unser Textfeld soll verlustfrei sein. Das heißt, wird das Browserfenster geschlossen, bevor der Benutzer seinen eingegebenen Text an den Server geschickt hat, soll die noch nicht gespeicherte Eingabe automatisch im Local Storage des Browsers gesichert werden. Das Google Web Toolkit stellt für den Zugriff auf diesen Speicher die Klasse `com.google.gwt.storage.client.Storage`zurVerfügung. Wir werden eine Referenz auf diese Klasse verwenden, um die Benutzereingaben in regelmäßigen Abständen zu sichern.

Der Local Storage des Browsers ist ein recht einfacher Key-Value-Speicher für Strings. Das heißt, man kann beliebige Zeichenketten in den Speicher eintragen. Mit frei wählbaren Strings als Schlüssel können diese dann später wieder ausgelesen werden. Die Daten bleiben auch nach dem Schließen des Browserfensters erhalten.

Für den Zugriff auf den gesicherten Wert des Textfeldes spendieren wir unserer Komponente eine eigene ID, die wir in der Variable mit dem Namen `textAreaUID` ablegen. Die ID wird als Schlüssel für den Local Storage verwendet. Diese ID werden wir später von der Serverkomponente aus setzen. Wie wir das erreichen können, werden wir uns später im Detail anschauen. Zunächst werfen wir einen Blick auf den Konstruktor von `VResettableTextArea`:

```
public VResettableTextArea() {
   super();
   resetButton = new ResetButton("Reset");
   addResetButtonClickHandler();
   panel = new VerticalPanel();
   textArea = new TextArea();
   panel.add(textArea);
   panel.add(resetButton);
   initWidget(panel);
   setStyleName(CLASSNAME);
   localStorage = Storage.getLocalStorageIfSupported();
   startSnapshotTimer();
   getCachedValueFromLocalStorage();
   VConsole.log("Local storage supported: " +
      Storage.isLocalStorageSupported());
}
```

Zuerst erzeugen wir uns die beiden Komponenten `TextArea` und `ResetButton` und legen diese auf das `VerticalLayout`. Anschließend müssen wir die Methode `initWidget()` von `Composite` aufrufen, um das Layout-Element als die Hauptkomponente für unser Widget zu definieren. Danach setzen wir unseren CSS Klassennamen für das Widget.

Als letzte Aktion im Konstruktor unseres Widgets richten wir den Zugriff auf den Local Storage des Browsers ein. Zuerst holen wir uns dazu eine Referenz auf die GWT Storage-Klasse, über die wir den Browserspeicher ansprechen können. Leider wird der lokale Browserspeicher noch nicht von jedem Browser unterstützt. Das Interface der Storage-Klasse bietet daher Funktionen an, die auf diesen Umstand entsprechend eingehen. Mit der Funktion `isLocalStorageSupported()` können wir feststellen, ob der Local Storage von dem Zielbrowser überhaupt unterstützt wird. Wir geben das Ergebnis davon mit Hilfe der Vaadin-Klasse `VConsole` auf die Debug-Konsole aus. Log-Ausgaben, die mit `VConsole.log()` ausgegeben werden, erscheinen in Vaadins Debug-Fenster und in der Debug-Konsole des Browsers, wenn eine solche unterstützt wird (bspw. über das Firebug-Plugin für Firefox).

Die Referenz auf das Storage-Objekt holen wir uns mit `getLocalStorageIfSupported()`. Die Funktion liefert null zurück, wenn der Local Storage nicht durch den Browser unterstützt wird. Wir müssen daher vor jedem Zugriff auf die Storage-Referenz eine Nullprüfung durchführen.

Nachdem wir uns die Referenz auf das Storage-Objekt geholt haben, gibt es noch zwei weitere Dinge zu tun. Zum einen müssen wir einen Timer konfigurieren, der den aktuellen Inhalt des Textfeldes in regelmäßigen Abständen in den Browserspeicher sichert. Zum anderen müssen wir das Textfeld mit dem aktuellen Inhalt des Local Storages initialisieren. Befindet sich für die ID des Textfelds ein Wert im Browserspeicher, dann bedeutet dies, dass dieser Wert zuvor noch nicht von der Anwendung zum Server geschickt werden konnte. Möglicherweise ist ein Browserabsturz dazwischengekommen, oder der Benutzer hat die Seite verlassen, ohne vorher auf Speichern zu drücken. Mit diesem Wert werden wir das Textfeld initialisieren und somit die alte Eingabe für den Benutzer wiederherstellen.

Schauen wir uns die beiden Schritte im Detail an. Den Timer starten wir in der Methode `startSnapshotTimer()`:

```
private void startSnapshotTimer() {
   if (localStorage != null) {
      Timer timer = new Timer() {
         public void run() {
            saveContentToLocalStorage(textArea.getText());
         }
      };
      timer.scheduleRepeating(5000);
   }
}
```

Als Timer verwenden wir die GWT-Klasse `com.google.gwt.user.client.Timer`. Wir dürfen an dieser Stelle keine Timer-Implementierung verwenden, die mit Threads arbeitet. Da der Code, den wir hier schreiben, später im Browser laufen wird und in diesem Threads nicht direkt unterstützt werden, müssen wir eine spezielle Timer-Implementierung benutzen, die auf diesen Umstand entsprechend Rücksicht nimmt. Die GWT Timer-Klasse stellt eine solche Implementierung zur Verfügung.

Wir konfigurieren den Timer dergestalt, dass alle fünf Sekunden dessen `run()` -Methode aufgerufen wird. In dieser Methode speichern wir einfach den aktuellen Inhalt des Textfeldes im Local Storage des Browsers. Das lässt sich ganz einfach mit der Methode `setItem()` der Storage-Klasse bewerkstelligen.

```
public void saveContentToLocalStorage() {
   localStorage.setItem(textAreaUID, textArea.getText());
}
```

Als nächstes werfen wir einen Blick auf die Methode `getCachedValueFromLocalStorage()`, mit der wir während der Initialisierung des Widgets den gesicherten Wert für das Textfeld auslesen.

```
private void getCachedValueFromLocalStorage() {
   Scheduler.get().scheduleDeferred(new ScheduledCommand()
   {
      public void execute() {
         if (localStorage != null) {
            setText(localStorage.getItem(textAreaUID));
         }
      }
   });
}
```

Den Inhalt des Browserspeichers lesen wir mit `getItem()` und der ID des Textfelds als Schlüssel aus. Mit dem ausgelesenen Text initialisieren wir anschließend den Inhalt des Textfelds. Auf diese Weise haben wir unser Ziel erreicht, dass noch nicht gesicherte Texte wiederhergestellt werden können.

An dieser Stelle müssen wir uns eines Mechanismus von GWT bedienen, mit dem wir den Ausführungszeitpunkt von bestimmten Anweisungen festlegen können. Die Methode `getCachedValueFromLocalStorage()` wird von uns im Konstruktor des clientseitigen Widgets aufgerufen. Innerhalb dieser Methode verwenden wir die ID des Textfeldes, um den gesicherten Text aus dem Browserspeicher auszulesen. Das Problem hierbei ist, dass während des Konstruktoraufrufes die ID, die wir serverseitig mit `setTextAreaUID()` setzen, noch gar nicht an das Client Widget geschickt wurde und die Variable `textAreaUID` somit noch null ist. Das Auslesen des Browserspeichers kann also erst später geschehen. Und zwar erst nachdem die `textAreaUID` an das Client Widget geschickt worden ist.

Das Google Web Toolkit gibt uns mit der `Scheduler` -Klasse eine Schnittstelle an die Hand, mit der wir die Ausführung von Code in die Zukunft verlagern können. Dieser Code wird in der `execute()` -Methode eines `ScheduledCommand` -Objekts definiert und mit den verschiedenen `schedule*()` -Methoden der Scheduler-Klasse für die Ausführung zu einem späteren Zeitpunkt vorgemerkt. Wir verwenden hier die Methode `scheduleDeferred()`, die unseren Code in dem Moment ausführt, in dem die Event Loop des Browsers aus der Event-Verarbeitung zurückkehrt. In unserem Fall ist das der Zeitpunkt, zu dem die `textAreaUID` bereits an unser Client Widget geschickt worden ist. In der `execute()` -Methode können wir daher auf die ID des Textfeldes zugreifen.

DER RESET-BUTTON

Wir wollen unserer Komponente eine eigene Schaltfläche spendieren, die den Inhalt des Textfeldes zurücksetzt, ohne dafür den Server kontaktieren zu müssen. Die Schaltfläche ist mit einem entsprechenden Icon dekoriert, das wir über ein GWT Client Bundle laden möchten. Das folgende Listing zeigt die Implementierung des Client Bundles.

```
public interface IconResources extends ClientBundle {
   public static final IconResources INSTANCE =
   GWT.create(IconResources.class);
   @Source("reset.png")
   ImageResource resetIcon();
}
```

Unser Client Bundle wird durch das Interface `IconResources` definiert, das von dem GWT-Interface `com.google.gwt.resources.client.ClientBundle` abgeleitet wird. Jede Ressource, die man in seiner Anwendung verwenden möchte, wird in diesem Interface in Form einer annotierten Methode deklariert. Die verwendete Annotation und der Rückgabetyp der Methode richten sich dabei nach der Dateiart der einzubindenden Ressource. In unserem Fall ist dies eine Bilddatei `reset.png` mit dem dazugehörigen Rückgabetyp `ImageResource`. Die Datei wird vom GWT-Compiler im gleichen Package gesucht, in dem auch das Client Bundle Interface liegt.

Um unser Client Bundle verwenden zu können, benötigen wir zuerst eine Instanz davon. Diese lassen wir uns über den Deferred Binding [4] Mechanismus von GWT als statische Variable erzeugen:

```
public static final IconResources INSTANCE =
GWT.create(IconResources.class);
```

Daraus wird während der GWT-Kompilierung JavaScript-Code generiert, der sich um das Laden der referenzierten Ressourcen kümmert. Die Instanz-Variable kann nun wie ein Factory-Objekt für Ressourcendaten verwendet werden.

Das Client Bundle wird in unserer eigenen Klasse `ResetButton` verwendet, welche wir von der Standard Vaadin-Klasse `VButton` ableiten. Das folgende Listing zeigt die Implementierung von `ResetButton`:

```
public class ResetButton extends VButton {
   public ResetButton() {
      Image resetIcon = new
      Image(IconResources.INSTANCE.resetIcon());
      resetIcon.setStylePrimaryName("v-icon");
      super.wrapper.insertBefore(resetIcon.getElement(),
      super.captionElement);
   }
}
```

Wir können hier nicht einfach `VButton` direkt verwenden, da wir mit der Vaadin-Standardimplementierung das Icon nicht über ein Client Bundle setzen können. Um das zu erreichen, leiten wir von `VButton` ab und sorgen selbst für das Platzieren des Icons. Dazu erzeugen wir ein `Image` -Objekt, das wir mit der `ImageResource` aus unserem Client Bundle initialisieren. Damit werden dem Image-Objekt die eigentlichen Bilddaten übergeben. Danach weisen wir dem Bild die CSS-Klasse für Vaadin-Icons zu, damit sich die Grafik optisch in den Vaadin Standard einfügt. Das Bild wird schließlich per DOM-Manipulation vor die Beschriftung des Buttons in den HTML Code eingefügt. Hervorzuheben ist an der Stelle, dass wir bei diesen Schritten ausschließlich GWT Programmierung betreiben. Wir benötigen dafür noch keinerlei Vaadin Mechanismen.

Das Ergebnis des Ganzen ist, dass wir nun einen Reset-Button haben, der genauso aussieht und sich genauso verhält, wie ein herkömmlicher Vaadin-Button mit Icon. Der entscheidende Unterschied ist, dass die Bilddaten nun nicht mehr über einen separaten GET-Request vom Server geladen werden müssen. Der Inhalt des Bildes wird statt dessen von GWT direkt in die Anwendung hineinkompiliert und gleich beim ersten Laden der Anwendung zusammen mit dem JavaScript-Code der Anwendung geladen. Man kann sich mit Hilfe eines Tools, wie z. B. Firebug, das Ergebnis davon anschauen. Die Bilddaten liegen Base64-kodiert als Data-URL [5] direkt in der CSS-Style-Definition des zu dem Icon gehörenden Image-Tags ab.

Diese Vorgehensweise hat den Vorteil, dass häufig gebrauchte Bildressourcen, wie z. B. Anwendungsicons, Gestaltungselemente oder Standardgrafiken, nicht separat geladen werden müssen. Dies kann abhängig von der Menge der einzelnen zu ladenden Bilddateien den Ladevorgang einer Webseite erheblich verzögern. Da laut HTTP-Spezifikation ein Browser immer nur maximal zwei gleichzeitige Verbindungen zu einem Server aufrecht erhalten darf, muss das Laden der einzelnen Ressourcendateien serialisiert werden. GWTs Client Bundles helfen dabei, den Overhead durch das Nachladen von Ressourcen zu verringern.

Zu guter Letzt müssen wir dem Reset-Button nur noch einen `ClickHandler` spendieren. Ein `ClickHandler` ist das GWT-Pendant zu Vaadins `Button.ClickListener`. In dessen `onClick()` -Methode leeren wir den Inhalt des Textfeldes und entfernen den Text aus dem Browser-Speicher.

```
private void addResetButtonClickHandler() {
   resetButton.addClickHandler(new ClickHandler() {
      public void onClick(ClickEvent event) {
         setText("");
         if (localStorage != null) {
            localStorage.removeItem(textAreaUID);
         }
      }
   });
}
```

An dieser Stelle ist hervorzuheben, dass das gesamte Event-Handling für die Schaltfläche rein im Browser abläuft. Wir verwenden für die Behandlung des Click Events den dazugehörigen GWT Event Handler und nicht den serverseitigen Vaadin-Mechanismus. Die Verarbeitung des Events geschieht im Browser, und es muss kein expliziter Request an den Server geschickt werden. Damit haben wir unser Ziel erreicht, eine Schaltfläche zum Zurücksetzen eines Textfeldes zu schreiben, die völlig ohne Serverkommunikation auskommt.

CONNECTOR UND DIE SERVERKOMPONENTE

Für die Verbindung von Client- und Serverteil einer Vaadin-Komponente sind eigene Connector-Objekte zuständig. Um unsere clientseitige Widget-Klasse `VResettableTextArea` mit ihrem serverseitigen Gegenstück namens `ResettableTextArea` zu verbinden, schreiben wir eine Connector-Klasse, die von der abstrakten Klasse `com.vaadin.client.ui.AbstractComponentConnector` erbt. Wir nennen unseren Connector passenderweise `ResettableTextAreaConnector` und legen diesen ebenfalls unter das `client` -Package, in dem auch unsere Widget-Klasse liegt. Auch der Connector muss vom GWT Compiler übersetzt werden können.

```
@Connect(ResettableTextArea.class)
public class ResettableTextAreaConnector extends
  AbstractComponentConnector implements BlurHandler {
   @Override
   public VResettableTextArea getWidget() {
      return (VResettableTextArea) super.getWidget();
   }
   @Override
   protected Widget createWidget() {
      return GWT.create(VResettableTextArea.class);
   }
   //...
}
```

Die Verknüpfung zu dem Client Widget wird über die Methode `getWidget()` hergestellt, die man für einen Connector überschreiben muss. Den Rückgabetyp der Methode schränkt man auf die Klasse seines Widgets ein, um dem Framework diese bekannt zu machen. In der Connector-Klasse greift man nun ausschließlich über diese Methode auf die Instanz des Widgets zu. Eine eigene Instanz für das Widget muss von uns nicht erzeugt und vorgehalten werden. Das Framework kümmert sich selbständig um die Erzeugung und Verwaltung eines Widget-Objekts.

Den Algorithmus zum Erzeugen der Widget-Instanz geben wir jedoch vor. Das tun wir, indem wir die Methode `createWidget()` überschreiben. Hier erzeugen wir ein neues Objekt unserer Widget-Klasse mit dem GWT-Mechanismus für Deferred Binding. Die statische Funktion `GWT.create()` weist den GWT Compiler an, eine Instanz der angegebenen Klasse bereitzustellen. Welche konkrete Implementierung der Compiler dann zurückgibt und wie er das genau anstellt, liegt vollkommen in dessen Verantwortung. Als Entwickler brauchen wir uns darum nicht zu kümmern. Wir haben später die Möglichkeit, über die Konfiguration auf diesen Prozess Einfluss zu nehmen. Über den Deferred Binding-Mechanismus von GWT kann man mit der GWT Modul-Konfiguration bestimmen, unter welchen Bedingungen der GWT Compiler welche Implementierung mit `GWT.create()` zurückgibt. Standardmäßig wird die Klasse erzeugt, die wir der `create()` -Methode als Parameter übergeben.

Man sollte daher sein clientseitiges Widget nicht mit dem `new` -Operator erzeugen, sondern auf die `GWT.create()` -Methode zurückgreifen.

Mit dem Überschreiben der beiden Methoden `getWidget()` und `createWidget()` wurde die notwendige Verbindung des Connectors mit dem clientseitigen Widget geschaffen. Nun muss dem Framework auch noch mitgeteilt werden, mit welcher serverseitigen Komponente der Connector verbunden werden soll. Dies geschieht über die Annotation `com.vaadin.shared.ui.Connect`, die wir an unsere Connector-Klasse hängen. Der Annotation geben wir die Klasse unserer serverseitigen Komponente mit.

Mit diesem letzten Schritt ist nun die Verbindung zwischen clientseitigem Widget und serverseitiger Komponente vollständig hergestellt worden. Das Connector-Objekt dient ab jetzt als Vermittler zwischen diesen beiden Objekten.

Die Serverkomponente selbst ist in unserem einfachen Beispiel eine einfache Subklasse von `AbstractField<String>`. Eine Field-Komponente ist im Vaadin-Framework ein UI-Element, das einen einzelnen, vom Benutzer eingegebenen Wert verwaltet. Das kann ein Datum, ein numerischer Wert oder eben, wie in unserem Beispiel, ein Text sein. `AbstractField` implementiert das `com.vaadin.data.Property` -Interface. Somit fügen sich Field-Klassen nahtlos in das Data Binding-Konzept von Vaadin ein.

KOMPONENTENZUSTAND

Eine UI-Komponente hat üblicherweise einen bestimmten Zustand, der an unterschiedlicher Stelle geändert und abgefragt werden muss. Der Benutzer muss den Zustand im Browser ändern können, indem er z. B. einen Wert über die Komponente eingibt. Auf der anderen Seite muss der Zustand auch serverseitig angepasst werden können. Wird eine Benutzereingabe bspw. in einem Label angezeigt, muss der Wert des Labels von serverseitigem Code geändert werden können.

Vaadin hat in der Version 7 für die Verwaltung und Synchronisierung dieses Zustands das Konzept des Shared State eingeführt. Zustandsinformationen, die Client- und Serverteil einer Komponente gemeinsam haben, werden über ein Objekt verwaltet, das von der Klasse `com.vaadin.shared.communication.SharedState` erbt. Für eigene Vaadin-Komponenten, die einen eigenen Zustand haben, legt man eine State-Klasse an, die von einer passenden Unterklasse von `SharedState` abgeleitet ist. Diese Unterklassen bringen für den jeweiligen Einsatzzweck schon bestimmte standardmäßige Zustandsinformationen mit, wie z. B. die Maße einer Komponente oder deren Beschreibungs- und Überschriftstexte.

Für unser Textfeld sieht die Shared State-Klasse wie folgt aus:

```
package de.oio.vaadin.client.connectors;
import com.vaadin.shared.AbstractFieldState;
import com.vaadin.shared.annotations.DelegateToWidget;
public class ResettableTextAreaState extends
AbstractFieldState {
   public String text;
   @DelegateToWidget
   public String textAreaUID;
}
```

Wir leiten von `com.vaadin.shared.AbstractFieldState` ab und erben damit sämtliche Standardzustandsinformationen für Field-Komponenten. Zusätzlich definieren wir zwei eigene Zustandswerte `textAreaUID` und `text`. Die beiden String-Werte sollen vom Server aus geändert werden können. Die Variable `text` enthält einfach den Inhalt unseres Textfeldes. Der zweite Wert ist eine textuelle ID, die wir unserer Komponente zuweisen können. Mit dieser ID wird die Komponente ihrem jeweiligen Eintrag im Local Storage des Browsers eindeutig zugeordnet. Dies ist wichtig, da man die Komponente ja auch mehrmals auf derselben Seite verwenden kann und sich die gesicherten Werte dann nicht gegenseitig überschreiben sollen. Die `textAreaUID` wird später als Schlüssel in den Browserspeicher verwendet.

Die Klasse legen wir in das gleiche Package wie unseren Connector, so dass auch der Shared State durch den GWT Compiler übersetzt werden kann. Das ist notwendig, da wir mit unserem Client Widget lesend auf das State Objekt zugreifen müssen. Die beiden String-Felder können wir als `public` deklarieren, da das Shared State-Objekt ausschließlich als funktionsloser Datencontainer verwendet wird.

Wie wird nun der geteilte Zustand in Client Widget und Serverkomponente verwendet? Schauen wir uns dafür zunächst den clientseitigen Teil an. Die Connector-Klasse ist, wie der Name schon sagt, Vermittler zwischen Client- und Serverseite. Unser Connector verwaltet also clientseitig den Zugriff auf das State-Objekt. Um dem Framework unser eigenes Shared State-Objekt bekannt zu machen, müssen wir die Methode `getState()` im Connector überschreiben.

```
// In ResettableTextAreaConnector
@Override
public ResettableTextAreaState getState() {
   return (ResettableTextAreaState) super.getState();
}
```

Wir tun dies, indem wir einfach den Rückgabetyp dieser Methode auf unsere State-Klasse einschränken. Mehr müssen wir nicht machen, um diese Klasse bekannt zu geben. Insbesondere müssen wir nicht dafür sorgen, eine neue Instanz der State-Klasse anzulegen. Darum kümmert sich das Framework für uns. Der Zugriff auf den Komponentenzustand geschieht für uns ab jetzt ausschließlich indirekt über die `getState()` -Methode.

Es gibt nun zwei Möglichkeiten, wie die Informationen aus dem Zustandsobjekt in die Widget-Klasse übertragen werden. Hat die Widget-Klasse eine Property mit dem gleichen Namen, wie ein Wert aus dem Zustandsobjekt, und die dazugehörige öffentliche Setter-Methode, dann kann man das Framework dafür sorgen lassen, dass Änderungen an diesem Zustandswert automatisch an diese Property weitergereicht wird. Dies erreicht man mit der Annotation `com.vaadin.shared.annotations.DelegateToWidget`. Ein Zustandswert, der mit dieser Annotation markiert ist, wird automatisch an das Widget weitergegeben, sobald sich dieser geändert hat.

In unserem Fall lassen wir den Wert für die `textAreaUID` automatisch an das Widget übermitteln. In der Widget-Klasse gibt es dafür die entsprechende Setter-Methode:

```
// In VResettableTextArea
public void setTextAreaUID(String newTextAreaUID) {
   if (!this.textAreaUID.equals(newTextAreaUID)) {
         localStorage.removeItem(this.textAreaUID);
   }
   this.textAreaUID = newTextAreaUID == null ? "" :
   newTextAreaUID;
}
```

In diesem Setter sorgen wir zusätzlich dafür, dass der Browserspeicher aufgeräumt wird, wenn man die ID des Textfeldes ändert.

Bei der zweiten Möglichkeit, geänderte Zustandswerte an das Widget weiterzureichen, kümmert man sich selber um das Weiterleiten der aktualisierten Zustandsinformation. Dazu überschreibt man die Methode `onStateChanged()` in der Connector-Klasse.

```
// In ResettableTextAreaConnector
@Override
public void onStateChanged(StateChangeEvent stateChangeEvent)
{
   super.onStateChanged(stateChangeEvent);
   getWidget().setText(getState().text);
}
```

Diese Methode wird immer dann aufgerufen, wenn sich der Inhalt des Zustandsobjekts geändert hat. Wir reichen dann die geänderten Werte einfach an das Widget-Objekt weiter. In unserem Fall ist das der Inhalt des Textfelds.

SERVERSEITIGE ZUSTANDSÄNDERUNG

Der clientseitige Zustand einer Vaadin-Komponente wird über serverseitigen Code geändert. Dies wollen wir auch für unsere Komponente erreichen. Auch hier brauchen wir dafür Zugriff auf das State-Objekt. In der Serverkomponente `ResettableTextArea` müssen wir, wie zuvor in der Connector-Klasse, die Methode `getState()` überschreiben und den Rückgabetyp auf unsere State-Klasse einschränken.

```
// In ResettableTextArea:
@Override
protected ResettableTextAreaState getState() {
   return (ResettableTextAreaState) super.getState();
}
```

Sämtliche Zugriffe auf die Werte des Komponentenzustands müssen auch hier wieder über diese Methode abgewickelt werden:

```
// In ResettableTextArea:
public void setTextAreaUID(String uid) {
   getState().textAreaUID = uid;
}
public String getTextAreaUID() {
   return getState().textAreaUID;
}
@Override
public void setValue(String newFieldValue)
  throws ReadOnlyException, ConversionException {
   super.setValue(newFieldValue);
   getState().text = newFieldValue;
}
```

Für das Setzen des Inhaltes unseres Textfelds überschreiben wir die Methode `setValue()`, die über das `com.vaadin.data.Property`-Interface geerbt wird. Im Kontext einer Vaadin Field-Komponente steht dieser Property-Wert für den Inhalt unseres Textfelds. Den neuen Text reichen wir entsprechend an das State-Objekt weiter, damit er an das Client Widget übertragen werden kann. Um diesen Schritt kümmert sich jetzt das Vaadin Framework von alleine. Mehr müssen wir als Entwickler nicht tun, um serverseitig den Zustand des Clients zu ändern.

ZUSTANDSÄNDERUNG AUF DEM CLIENT

An diesem Punkt haben wir die Synchronisierung des Komponentenzustands vom Server auf den Client abgeschlossen. Der Mechanismus für die Gegenrichtung sieht ein wenig anders aus. Man könnte versucht sein, den Komponentenzustand clientseitig auf die gleiche Weise zu ändern, wie wir es für die Serverkomponente gesehen haben; das heißt über das Setzen eines Zustandswertes direkt im State-Objekt. Das wird allerdings nicht funktionieren. Greift der Client schreibend auf das Zustandsobjekt zu, wird eine solche Änderung einfach ignoriert und beim nächsten Update durch den Server überschrieben. Der Grund hierfür ist, dass der Server die Verwaltungshoheit über das State-Objekt hat. Könnten Client und Server den Komponentenzustand gleichberechtigt ändern, stünde man vor dem Problem, entscheiden zu müssen, welche Seite Vorrang bekommt. Es würde in bestimmten Situationen unweigerlich zu Datenverlust kommen.

Aus diesem Grund hat man sich dazu entschlossen, ausschließlich den Server den Komponentenzustand ändern zu lassen. Das Client Widget muss seine Änderungen also über einen anderen Mechanismus an den Server weiterreichen, als über das Shared State-Objekt. Für diesen Zweck wurde mit Vaadin 7 ein Server-RPC-Mechanismus eingeführt. Ein Client Widget kann damit direkt Methoden auf der Serverkomponente aufrufen und dadurch unter anderem den Komponentenzustand auf dem Server ändern.

AUFRUF VON SERVER-METHODEN ÜBER SERVER RPC

Damit der clientseitige Teil einer Vaadin Komponente entfernte Methoden auf dem Server aufrufen kann, müssen diese Methoden in einem eigenen RPC-Interface deklariert werden. Dieses Interface muss von dem Marker-Interface `com.vaadin.shared.communication.ServerRpc` erben. Für unsere Textfeld-Komponente sieht das wie folgt aus:

```
import com.vaadin.shared.communication.ServerRpc;
public interface ResettableTextAreaServerRpc extends
ServerRpc {
   @Delayed(lastOnly = true)
   public void setText(String text);
}
```

Wir definieren eine Methode `setText()`, mit der wir den aktuellen Inhalt des Textfeldes an die Serverkomponente schicken können. Die Methode soll immer dann aufgerufen werden, wenn das Textfeld den Eingabefokus verliert - sobald also der Benutzer das Textfeld verlässt.

Die RPC-Methode haben wir mit der Annotation `com.vaadin.shared.annotations.Delayed` erweitert. Damit weisen wir das Framework an, den entfernten Methodenaufruf nicht sofort auszuführen, sondern ihn zurückzuhalten, bis die nächste reguläre Anfrage an den Server geschickt wird. Das ist zum Beispiel immer dann der Fall, wenn ein Button geklickt wurde. Das Attribut `lastOnly` gibt an, dass immer nur der jeweils letzte Aufruf einer Reihe von zurückgehaltenen RPC-Methoden an den Server geschickt werden soll. Wir benötigen diese Annotation in unserem Fall, da wir mit unserer Komponente so wenig Netzwerkverkehr wie möglich verursachen wollen. Würden wir unsere RPC-Methode nicht mit `Delayed` annotieren, so würde der aktuelle Inhalt des Textfeldes sofort an den Server geschickt werden, sobald das Textfeld den Fokus verliert. Das wollen wir ja gerade vermeiden.

Wir müssen zwei Dinge erledigen, um das RPC-Interface benutzen zu können: Die RPC-Methode muss serverseitig implementiert werden, und auf Clientseite muss die Methode aufgerufen werden können.

Auf Serverseite implementieren wir das Interface im Konstruktor der Serverkomponente als anonyme Klasse und registrieren diese Implementierung als RPC-Endpunkt.

```
public class ResettableTextArea extends AbstractField<String>
{
   public ResettableTextArea() {
      registerRpc(new ResettableTextAreaServerRpc() {
         @Override
         public void setText(String text) {
            setValue(text);
         }
      });
   }
   @Override
   public void setValue(String newFieldValue)
     throws ReadOnlyException, ConversionException {
      super.setValue(newFieldValue);
      getState().text = newFieldValue;
   }
   //...
}
```

Mit `registerRpc()` legen wir fest, dass das übergebene Objekt die vom Framework zu verwendende Implementierung eines bestimmten RPC-Interfaces darstellt und für entsprechende Methodenaufrufe vom Client verwendet werden soll. In unserer Implementierung reichen wir einfach den übergebenen Text des Textfeldes an die `setValue()` -Methode weiter, die ihrerseits das Shared State-Objekt aktualisiert. Hier sieht man, wie die Serverkomponente die alleinige Verantwortung für die Verwaltung des Komponentenzustands übernimmt.

Nun muss die Clientseite die Möglichkeit haben, die angebotenen RPC-Methoden auch aufzurufen. Das geschieht über ein Proxy-Objekt für das RPC-Interface, das man sich vom Framework geben lässt. Auch hier dient das clientseitige Connector-Objekt wieder als Mittler zwischen Client Widget und Serverkomponente. Das Proxy-Objekt kann im Connector mit der Methode `getRpcProxy()` geholt werden.

```
public class ResettableTextAreaConnector extends
  AbstractComponentConnector implements BlurHandler {
   public ResettableTextAreaConnector() {
      getWidget().getTextArea().addBlurHandler(this);
      getWidget().getResetButton().addClickHandler(new
      ClickHandler() {
         @Override
         public void onClick(ClickEvent event) {
            sendCurrentText();
         }
      });
   }
   // ...
   @Override
   public void onBlur(BlurEvent blurEvent) {
      sendCurrentText();
      getWidget().saveContentToLocalStorage();
   }
   private void sendCurrentText() {
      getRpcProxy(ResettableTextAreaServerRpc.class).setText(
        getWidget().getText());
   }
}
```

Wir wollen immer dann den an den Server zu schickenden Text aktualisieren, wenn das Feld den Eingabefokus verliert. Wir registrieren die Connector-Klasse daher als `BlurHandler` für das Textfeld und rufen in dem dazugehörigen `onBlur()` -Event Handler die RPC-Methode auf. Dieser Methode übergeben wir dann den aktuellen Inhalt des Textfeldes als Parameter. Anschließend aktualisieren wir den Text im Local Storage des Browsers.

Außerdem muss der an den Server zu schickende Text immer dann geleert werden, wenn der Benutzer auf den `ResetButton` geklickt hat. Aus diesem Grund registrieren wir für den `ResetButton` im Konstruktor des Connectors einen weiteren `ClickHandler`, in dem der aktuelle (geleerte) Inhalt des Textfeldes über die RPC-Methode an den Server geschickt wird.

Auf diese Weise haben wir einen Kommunikationskanal vom Client hin zum Server errichtet. Über diesen Kanal kann das Client Widget den aktuellen Komponentenzustand ändern.

Eine direkte Kommunikation in die Gegenrichtung ist ebenfalls möglich. Das heißt, der Server kann über denselben RPC-Mechanismus entfernte Methoden auf dem Client aufrufen. Das dafür notwendige Interface ist `com.vaadin.shared.communication.ClientRpc`. Damit könnte das Client Widget bspw. eine Methode zur Verfügung stellen, mit der der Server programmatisch den im Browserspeicher gesicherten Text des Textfeldes zurücksetzen kann. Das ist immer dann notwendig, wenn der Inhalt des Textfeldes erfolgreich vom Backend gespeichert wurde.

VERWENDUNG DER KOMPONENTE IN EINER VAADIN-ANWENDUNG

Nachdem wir schließlich die gesamte Vorarbeit zur Entwicklung einer eigenen Vaadin-Komponente geleistet haben, können wir jetzt die Früchte unserer Arbeit genießen. Der letzte Schritt, der uns noch fehlt, ist die Kompilierung unseres eigenen Widget Sets und die Verwendung unserer neuen Komponente in einer Vaadin-Applikation. Wir können unser Widget Set entweder über das Vaadin Plugin für Eclipse kompilieren, wenn wir unsere Komponente als Eclipse Projekt umsetzen. Eine andere Möglichkeit besteht darin, das Projekt als Maven Build aufzuziehen und das Widget Set mit dem Vaadin Maven Plugin zu kompilieren. Das Beispielprojekt verwendet letztere Variante.

Bei der Verwendung der Komponente in einer Vaadin-Anwendung unterscheidet sich unsere `ResettableTextArea` nicht von einer beliebigen anderen Vaadin-Komponente. Wir können wie im folgenden Listing das Textfeld zum Beispiel auf ein `VerticalLayout` legen und mit den angebotenen Setter-Methoden konfigurieren.

```
VerticalLayout layout = new VerticalLayout();
ResettableTextArea resettableTextArea = new
ResettableTextArea();
resettableTextArea.setTextAreaUID("MY_RESETTABLE_TEXTAREA");
layout.addComponent(resettableTextArea);
```

Mit der Methode `setTextAreaUID()` definieren wir die ID für das Textfeld. Über den Shared State-Mechanismus wird dieser Wert für uns automatisch an das Client Widget übermittelt, wo er als Schlüssel in den Local Storage des Browsers dient.

BEISPIELCODE

Für die in diesem Artikel vorgestellte Vaadin-Komponente existiert ein Beispielprojekt auf GitHub [6]. Das Projekt enthält den Code für die Komponente und eine Beispielanwendung, in der die `ResettableTextArea` verwendet wird. Gebaut und ausführt werden kann das Projekt mit Hilfe von Apache Maven [7]. Hat man Maven installiert und das `mvn` -Kommando entsprechend in den PATH gelegt, kann man mit dem folgenden Befehl, den man im Hauptverzeichnis des Projekts absetzt, das Projekt bauen lassen und starten:

```
mvn package jetty:run
```

Wurde der Jetty-Server gestartet, kann man die Beispielanwendung über die URL `http://localhost:8080/ResettableTextArea` in einem Browser ausprobieren.

FAZIT

Der vorliegende Artikel hat einen Überblick darüber gegeben, wie man bei der Entwicklung eigener Komponenten für Vaadin vorgeht. Wir haben dazu untersucht, wie man den client- und serverseitigen Teil einer Komponente implementiert und wie man mit Hilfe eines Connector-Objekts die Kommunikation zwischen diesen beiden Bestandteilen gestalten kann. Clientseitig haben wir auf die Klassen des Google Web Toolkits zurückgegriffen, um die Funktionen unseres Client Widgets umzusetzen.

In unserem Beispiel haben wir eine Komponente entwickelt, die für einen bestimmten Einsatzzweck hin optimiert wurde. Ein Ziel dabei war es, bestimmte Aktionen nicht über den Server gehen zu lassen, um die daraus resultierende Netzwerklatenz zu vermeiden. Auch hier gilt die Devise nach Donald E. Knuth: "*Premature optimization is the root of all evil*". Bevor man sich also an die Entwicklung besonders optimierter eigener Komponenten macht, sollte man zuerst diejenigen Stellen einer Anwendung identifizieren, bei denen man mit speziell angepassten Komponenten tatsächlich eine messbare Verbesserung der Leistungsfähigkeit erreichen kann. Man sollte es vermeiden, zu viele speziell angepasste Komponenten zu schreiben, um sich nicht die Vorteile des Vaadin-Frameworks zunichte zu machen. Stellt man fest, dass die zusätzliche Netzwerklatenz, die das Vaadin Framework mit sich bringt, an sehr vielen Stellen weh tut, sollte man sich sogar überlegen, ob man seine Anwendung nicht lieber gleich komplett mit dem Google Web Toolkit implementiert. Das GWT hat derartige Latenzprobleme nicht, kann dafür aber kein so einfaches Programmiermodell anbieten wie Vaadin.

Weiter empfiehlt es sich auch, zuerst einen Blick auf das Vaadin Addon Directory [8] zu werfen, bevor man eine eigene Komponente entwirft. Vielleicht gibt es die vermisste Funktionalität dort schon in Form eines Vaadin Addons.

REFERENZEN

- [1] Vaadin
  (https://www.vaadin.com)
- [2] Das Half Object Plus Protocol Pattern
  (http://c2.com/cgi/wiki?HalfObjectPlusProtocol)
- [3] Die User Interface Definition Language
  (http://www.uidl.net/)
- [4] Deferred Binding des GWT
  (https://developers.google.com/web-toolkit/doc/latest/DevGuideCodingBasicsDeferred)
- [5] RFC 2397, The "data" URL scheme
  (http://www.ietf.org/rfc/rfc2397.txt)
- [6] Beispielprojekt auf GitHub
  (https://github.com/rolandkrueger/vaadin-by-example/tree/master/de/erweiterungen/Artikel-EigeneKomponentenMitGWT)
- [7] Apache Maven
  (http://maven.apache.org/)
- [8] Vaadin Directory
  (https://vaadin.com/directory)