Entwurfsbeschreibung

smartGWT

Was man zu verstehen gelernt hat, fürchtet man nicht mehr.
– Marie Curie

1 Allgemeines

smartGWT bildet das Application Programming Interface des Google Web Toolkit zu smartClient.

2 Produktübersicht

Eine vollständige und verständliche Entwurfsanalyse des smartGWT muss sich zuerst mit den allgemeinen Ideen und technischen Details der beiden zugrunde liegenden Frameworks - GWT und smartClient - beschäftigen, da smartGWT die Brücke zwischen diesen bildet. Diese Analyse kann sich, der Einfachheit willen, nur auf einer höheren Abstraktionsebene als die Analyse des smartGWT selbst befinden.

Dementsprechend wird im ersten Teil eine Analyse von GWT, smartClient und smartGWT auf Paketebene erfolgen. Im zweiten Teil wird anhand des Paketes com.smartGWT.client.widgets.tree.* eine Analyse der, für die Entwicklung einer Webanwendung mit smartGWT, relevanten Funktionen auf Klassenebene vorgenommen.

3 Grundsätzliche Struktur- und Entwurfsprinzipien für das Gesamtsystem

3.1 Google Web Toolkit(GWT)

Das von Google entwickelte Software Development Framework ermöglicht es Web-Entwicklern AJAX-Anwendungen vollständig in Java zu schreiben. Dies wird hauptsächlich durch den integrierten Java-to-JavaScript-Compiler ermöglicht.
Zusätzlich stehen GWT Remote Procedure Call (ein Interface ähnlich der Java Remote Method Invocation), User-Agent-basierte Optimierung, Internationalisierung, JavaScript-Unterstützung durch das JavaScript-Native-Interface (JSNI), JSON und XML Unterstützung, sowie verschiedene Methoden zur Vereinfachung der Entwicklung zur Verfügung.

3.1.1 Java-to-JavaScript-Compiler

Das Kernstück des GWT bildet der Java-to-JavaScript-Compiler. Dieser übersetzt jeden client-seitigen Quelltext in JavaScript und ermöglicht somit die Ausführung im Webbrowser. Bei der Übersetzung wird der Quelltext optimiert und verschleiert.

Eine besonders erwähnenswerte Technik der Optimierung ist das späte Binden. Da dynamisches Binden in JavaScript-Umgebungen nicht zur Verfügung steht, werden zur Kompilations-Zeit Klassen „dynamisch“ geladen, anstatt zur Ausführungszeit.
Damit werden mehrere browserspezifische Versionen des JavaScript-Quelltextes erstellt. Dies ermöglicht jedem User nur den benötigten Quelltext herunterzuladen und lässt den Compiler verschiedene Inline-Optimierungen vornehmen, die ansonsten polymorphe Methodenaufrufe wären. Dadurch kommt es zu einem Geschwindigkeitsvorteil von bis zu 10% gegenüber herkömmlichen JavaScript.

Wichtig ist, dass GWT nicht alle Javakonstrukte und Bibliotheken unterstützt. Die meiste Bedeutung kommt dabei der „GWT Emulated JRE Library“ zu.

3.1.2 GWT Remote Procedure Call (RPC)

RPC ist ein Mechanismus, der es AJAX-Anwendungen ermöglicht, neue Daten von einem Server abzurufen, um diese dann client-seitig zu verwenden.

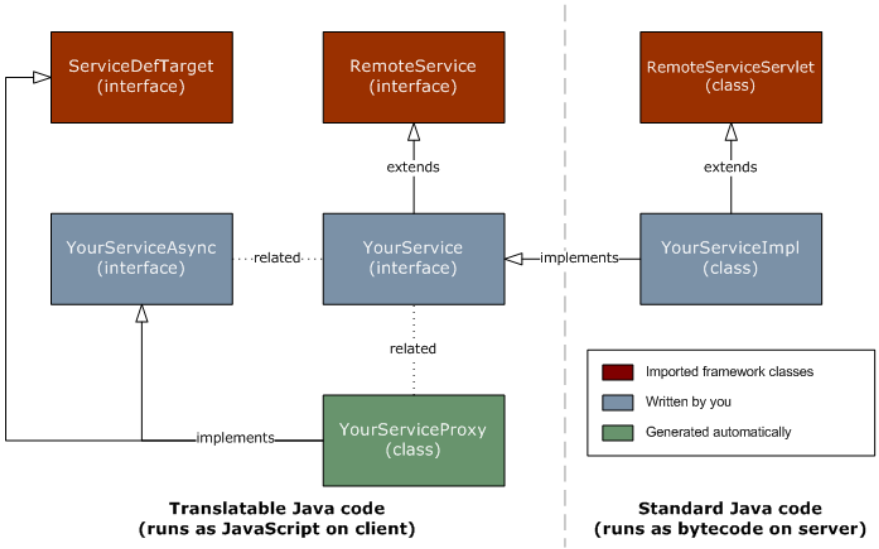

Abbildung 1: Anatomy Of Services

In dieser Architektur wird der Service durch einen Aufruf mit einem Callback-Objekt als Parameter angefordert. Die Service Implementation arbeitet dann serverseitig den Aufruf ab und benachrichtigt das Callback-Objekt über den Abschluss.
Dies ist nötig um den Kontrollfluss augenblicklich zum Client zurückkehren zu lassen. Diese Notwendigkeit hat mehrere praktische Gründe: Zum einen sind die meisten JavaScript-Engines in Browser single-threaded und eine Anwendung würde durch das Warten auf einen Server zum Stillstand kommen.

Zusätzlich kann Asynchronität als eine Form der Parallelisierung betrachtet werden, da Server und Client gleichzeitig unterschiedliche Funktionen ausführen können. Zudem können mehere RPCs zu unterschiedlichen Servern gleichzeitig stattfinden und somit kann Rechenzeit besser genutzt werden.

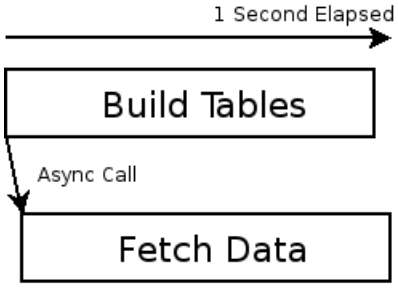

Abbildung 2: Getting Used To Async Calls

Ein wichtiges Konzept im Zusammenhang mit RPCs ist Fehlerbehandlung. GWT RPC benutzt hierfür das Java-eigene Exception Konzept mit der Ausnahme, dass alle eigenen Exceptions aus Exceptions der von GWT emulierten JRE Bibliothek aufgebaut sein müssen.

3.1.3 Internationalisierung

Web-Anwendungen zielen oft auf eine möglichst große Nutzerbasis ab und benötigen dementsprechend eine angepasste Lokalisierung an die Herkunft des Nutzers.
String-Internationalisierung ist eine bekannte Methode um diesen Effekt zu erreichen, ohne Einfluss auf die Leserlichkeit des Quelltextes zu nehmen und somit die Wartbarkeit und Erweiterbarkeit zu erhalten.

Da Webanwendungen sehr ressourcensparend arbeiten müssen, ist es hinderlich, jedem Nutzer alle Lokalisierungen zuzusenden, um dann die richtige durch clientseitiges JavaScript auszuwählen.

Dies wird im GWT durch drei Methoden ermöglicht:

- Statische String-Internationalisierung,
- Dynamische String-Internationalisierung und
- Implementierung von „Localizable“.

Die erste Methode bedient sich des bereits anfangs erwähnten Mechanismus des späten Bindens. Es werden alle Tags mit den entsprechenden Strings zur Kompilationszeit ersetzt. Um diese Variante zu nutzen muss, je nach Situation, eines der Interfaces Constants, ConstantsWithLookup oder Messages implementiert werden. Statische Internationalisierung ermöglicht eine tiefe Compileroptimierung und häufiges Inlining. Die Performanz gleicht also der von statisch übergebenen Strings.

Dynamische Internationalisierung kann verwendet werden, um existierende, internationalisierte Applikationen um GWT Funktionalität zu erweitern.
Die Klasse „Localizable“ kann verwendet werden, um mehrsprachige Bibliotheken zu implementieren. Beide Methoden sind für das Projekt nicht von weiterer Bedeutung und werden daher hier nicht erläutert.

Ähnlich zum Bereich der Internationalisierung ist die Barrierefreiheit. Hier gilt das gleiche, wie für die letzten beiden Internationalisierungsmethoden.

3.1.4 GWT User Interface Design

Klassische AJAX-Anwendungen manipulieren das Document Object Model (DOM) eines Browsers direkt, um ein User Interface zu erstellen. GWT erlaubt zwar die Manipulation des DOM durch die Klasse com.google.gwt.user.client.DOM, bietet hierfür jedoch einfachere Methoden an. Z.B. die Klassen aus dem Paket com.google.gwt.user.ui.*.

Die GWT User Interface Klassen sind wie Bibliotheken modelliert und verhalten sich ähnlich (wie z.B. „Swing“ oder „SWT“). Der Unterschied entsteht hauptsächlich durch die Verwendung von HTML-Elementen und das Rendern durch den Browser, anstatt der Verwendung von pixelorientierten Grafiken. Daher baut sich jedes User Interface zu allererst aus einem der unterschiedlichen, verfügbaren Panels auf. Dieses kann wiederum mit Widgets, wie z.B. Button, TextBox, Tree und anderen Panels befüllt werden. Widgets können als Ausgangspunkt für Eigenkompositionen verwendet werden.

3.1.5 Event Handler

Mit Version 1.5 wurde das Event Systems des GWT vom herkömmlichen Listener-Pattern auf ein Handlersystem umgestellt. Es ist nun möglich jedem Widget einen spezifischen, möglicherweise anonymen Handler hinzuzufügen und in diesem die entsprechenden Aktionen zur Behandlung des Elements vorzunehmen.

3.2 smartClient

smartClient ist ein JavaScript-Framework, dass eine Vielzahl von vorgefertigten User Interface Elementen bietet. Essentiell ist dabei die Möglichkeit Metadaten-Formate, wie z.B. XML, an Komponenten zu binden und somit ein Client-Server-Datenmodell mit erweiterbaren Typen, Validierungsregeln, Editierbeschränkungen, sowie den CRUD-Operationen zu ermöglichen.

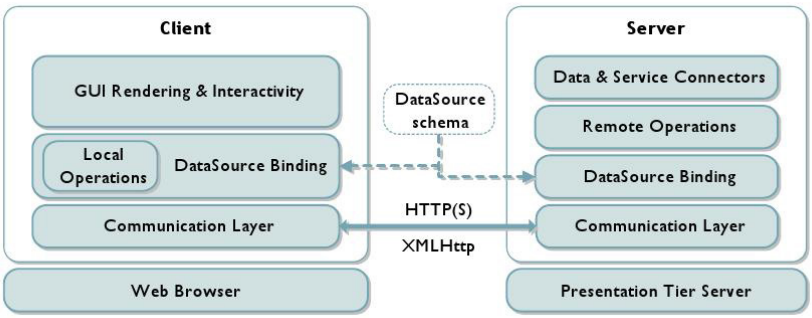

Abbildung 3: smartClient-Architektur

smartClient erlaubt die Verwendung mehrerer Sprachen, um Anwendungen zu erstellen:

- XML zur deklarativen Definition des User Interface und/oder der Datenquellen,
- JavaScript für clientseitige Logik, eigene Komponenten und prozedurale User-Interface-Definition und
- Java zur Datendefinition (benötigt den smartClient Java Server).

Die wichtigsten Bestandteile des smartClient-Frameworks im Zusammenhang mit GWT sind die „Visuellen Komponenten“ und der Vorgang der „Datenbindung“ an diese. Eine exemplarische Erläuterung dieser Konzepte wird an späterer Stelle des Dokumentes erfolgen.

3.3 smartGWT

smartGWT bildet die Synthese zwischen smartClient und GWT. Dabei werden die Vorteile beider Frameworks vereint:

- die Entwicklung ist vollständig in Java möglich,
- es existiert eine Vielzahl von Widgets,
- die Verwendung des DataSource-Konzeptes aus smartClient wird im GWT möglich und
- die Optimierungen des GWT-Compilers bleiben erhalten.

3.4 Widget-Adaption

GWT unterstützt nur eine kleine Menge leicht erweiterbarer Widgets. smartGWT gliedert die bereits in smartClient vorhandenen JavaScript-Widgets in die GWT Vererbungshierarchie ein und ermöglicht deren Verwendung in GWT Anwendungen. Zusätzlich werden die vom GWT bereitgestellten Event-Mechanismen erweitert und integriert.

4 Grundsätzliche Struktur- und Entwurfsprinzipien für die einzelnen Pakete

Im folgenden wird die Umsetzung der Daten in die zugehörigen visuellen Komponenten beispielhaft am Paket com.smartGWT.client.widgets.tree erläutert.

4.1 Package com.smartGWT.client.widgets.tree

Das Tree-Package ist Bestandteil des smartGWT und befindet sich im Widget-Package.

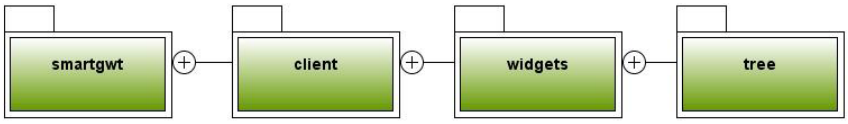

Abbildung 4: Hierarchie com.smartGWT.client.widgets.tree

Es besteht aus 5 Klassen und 2 Interfaces sowie einem Package Event.

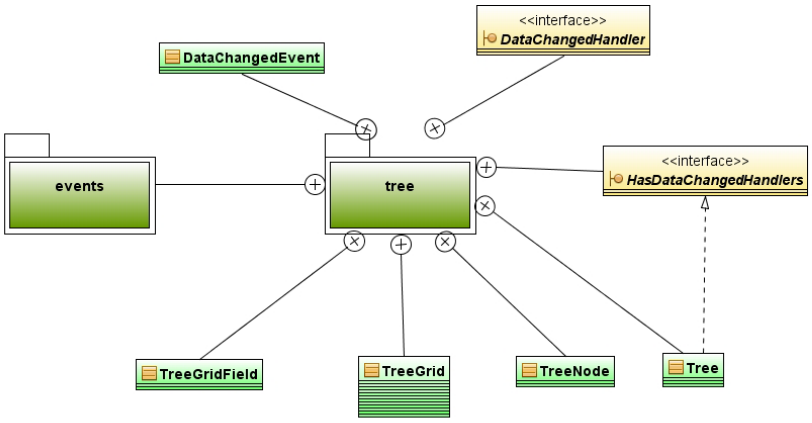

Abbildung 5: Package com.smartGWT.client.widgets.tree

4.1.1 Klasse Tree

Die Klasse Tree implementiert eine Datenstruktur, welche eine Menge von hierarchisch angeordneten Objekten aufnehmen kann. Dabei werden Objekte, welche auf Unterobjekte verweisen, als Knoten bezeichnet. Die Verzweigung in Unterobjekte kann auf mehrere Ebenen ausgeweitet werden. Objekte, welche keine Unterobjekte besitzen heißen Blätter. Erstellen lässt sich ein Tree aus einer Liste von Objekten, welche durch IDs aufeinander verweisen oder aus Objekten, deren hierarchisch untergeordnete Objekte explizit gegeben sind. Ein Tree an sich hat keine grafische Darstellung, er wird entweder als ColumnTree oder als TreeGrid angezeigt.

Tree ist direkt abgeleitet von BaseClass. Von BaseClass erbt Tree eine ID, welche den Tree eindeutig in der smartGWT-Instanz identifiziert, und ein JavaScript-Objekt. Die Klasse enthält zwei Konstruktoren, diverse Methoden zum Einfügen, Entfernen und Manipulieren von Daten, diverse Getter und Setter und einige JavaScript-to-Java Methoden.

Konstruktoren

- *Tree()*

  Erstellt einen neuen leeren Baum.

- *Tree(JavaScriptObject jsObj)*

  Dieser Konstruktor benötigt ein JavaScript-Objekt. Dies ist der Standardfall, da ein Tree meist durch die statische Methode getOrCreateRef aufgerufen wird.

JavaScript-zu-Java Methoden

- *TreeNode[] convertToTreeNodeArray(JavaScriptObject nativeArray)*

  Diese Methode konvertiert ein JavaScript-Array in ein Java-Array mit Elementen vom Typ TreeNode.

- *ListGridRecord[] convertToTreeNodeRecordArray(JavaScriptObject nativeArray)*

  Diese Methode konvertiert ein JavaScript-Array in ein Java-Array mit Elementen vom Typ ListGridRecord.

- *Tree getOrCreateRef(JavaScriptObject jsObj)*

  Diese Methode wird aufgerufen, wenn eine neue Referenz auf einen bestehenden Tree benötigt wird. Existiert das JavaScript-Objekt nicht, wird ein neuer Tree erzeugt.

- *TreeNode nodeForRecord(ListGridRecord record)*

  Da TreeNode von ListGridRecord abgeleitet ist, ist es nur folgerichtig, dass es eine Methode gibt, die ein ListGridRecord in ein TreeNode casten kann.

Handlermethoden

- *HandlerRegistration addDataChangedHandler(DataChangedHandler handler)*

  Methode in welcher ein Handler an den Tree gebunden wird, welcher Änderungen in der Struktur des Trees überwacht. Dabei wird zuerst die private Methode setupDataChangedEvent() aufgerufen, bevor der Handler im HandlerManager mit den gerade gesetzten Daten angemeldet wird.

- *SetupDataChangedEvent()*

  Hier werden in Java-Script die Attribute des JavaScript-Objekts (über den Konstruktor gesetzt) des Tree manipuliert. Wenn der Tree bereits einen Handler hat, wird die dataChanged-Section aktualisiert, sonst neu angelegt.

- *JavaScriptObject create()*

  Erstellt aus der in BaseClass erstellten Konfiguration des Trees ein JavaScript-Objekt des Tree.

- *ListGridRecord[] getData()*

  Erstellt aus den Daten des Tree ein ListGridRecord.

Datenmanipulation

- *Element einfügen:*

  Tree bietet verschiedene add(...) Methoden. Es können einzelne Elemente eingefügt werden, immer mit VaterID, optional mit Position. Das Hinzufügen mehrerer Objekte zu einem Vaterobjekt verwendet die addList(...) Methoden.

- *Element verschieben:*

  Hier gibt es zwei Methoden, es müssen der Knoten, der neue Vaterknoten und optional die Position bekannt sein.

- *Element entfernen:*

  Auch hier bietet Tree zwei Methoden: remove(TreeNode) zum Entfernen eines Elements und removeList(TreeNode[]) für mehrere Elemente.

4.1.2 Klasse TreeGrid

Die Klasse TreeGrid ist für die grafische Darstellung des Trees in einem Widget zuständig. Sie ist von ListGrid abgeleitet, welche die Darstellung einer einfachen Liste in einem Widget realisiert, und erweitert diese um das Feld „treeField“, welches einen hierarchischen Tree darstellen kann. Ein Tree wird in Zeilen und Spalten dargestellt.

4.1.3 Klasse TreeGridField

Die Klasse TreeGridField ist von ListGridField abgeleitet. ListGrid ist ein JavaScript-Objekt, welches für die Anzeige der Spalten und die Interaktion mit ihnen verantwortlich ist. TreeGridField implementiert Methoden zur Manipulation der Attribute des JavaScript-Objekts. Insbesondere enthält sie die Methode setTreeField, welche den Tree anzeigt oder ausblendet.

4.1.4 Klasse TreeNode

Die Darstellung eines Knoten in einem Tree wird über ein Objekt vom Typ TreeNode, welches vom JavaScript-Objekt ListGridRecord abgeleitet ist, realisiert. Die Klasse implementiert im wesentlichen Methoden zur Manipulation dieses Objekts, wie z.B.: Title, ID, Name oder isEnabled.

Für die Interaktion des Trees mit seinen Elementen erweitert smartGWT das Event-Konzept des GWT. Es wird ein Handler am HandlerManager angemeldet, welcher dann informiert wird, wenn sich am zugehörigen Objekt etwas geändert hat.

4.1.5 Klasse DataChangedEvent

Die Klasse DataChangedEvent ist abgeleitet von GwtEvent aus dem GWT. Sie implementiert eine fire-Methode, welche an alle im HandlerManager registrierten Handler „feuert“, wenn sich die Daten des Tree geändert haben, also wenn Element geändert, gelöscht oder hinzugefügt wurden.

4.1.6 Interface DataChangedHandler

Das Interface DataChangedHandler enthält die Methode, welche nach dem DataChangedEvent ausgeführt wird.

4.1.7 Interface HasDataChangedHandlers

Das Interface DataChangedHandler enthält die Methode, welche den Tree am HandlerManager anmeldet.

4.1.8 Package Event

Das Package Event enthält alle Operationen, um die Interaktion mit den Elementen eines Trees zu ermöglichen. Die vorhandenen Klassen „feuern“ bei Ereignissen an die registrierte Handler.

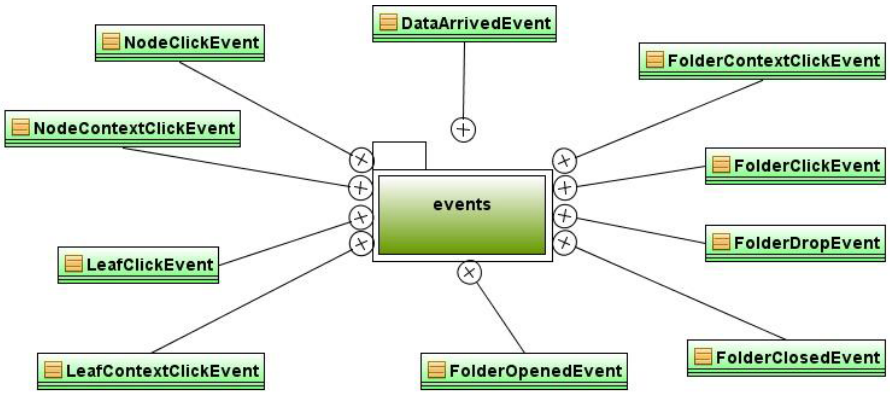

Abbildung 6: Package Event

Die enthaltenen Interfaces enthalten Methoden, welche nach dem Eintreten des Events ausgeführt werden (alle Interfaces, die auf *Handler enden) bzw. Methoden, um die Objekte am HandlerManager anzumelden (alle Interfaces die mit Has* beginnen).

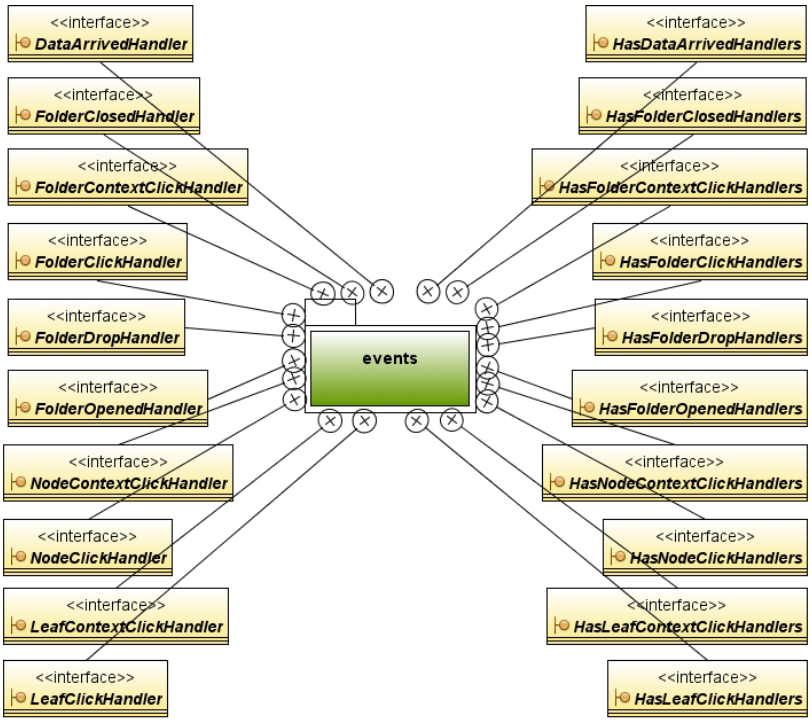

Abbildung 7: Interfaces Package Event