Einführung in das Google Web Toolkit

== Entwicklungszyklus ==
Das GWT arbeitet mit einer Standard-Verzeichnisstruktur, die entweder manuell oder unterstützt durch die mitgelieferten Skripte erstellt werden kann.
Für Eclipse wird erst ein Projektverzeichnis zum Import in den Workspace erstellt und dann im zweiten Schritt die Anwendung angelegt.

Die Schritte sollten in einem leeren, eigenen Verzeichnis durchgeführt werden, da die Verzeichnisstruktur direkt im ausführenden Verzeichnis abgelegt wird.

Quellcode

1. projectCreator -eclipse MeineErstesProjekt
2. applicationCreator -eclipse MeineErstesProjekt \
3. com.gwt.client.MeineErsteAnwendung

Geeignete Entwicklungsumgebungen können das Projektverzeichnis dann als normales Java-Projekt importieren und die eigentiche Programmierung kann beginnen.
Die Entwicklungsumgebung kann dabei mit Debugging-Möglichkeit, Refactoring-Tools und einer Unit-Testing-Integration unterstützen.
Die Standard-Verzeichnisstruktur enthält bereits ein einfaches 'Hello World', das im nächsten Kapitel zu einer Anwendung wachsen wird.

Während des Testens wird öfter in den Hosted-Mode gewechselt, der einen Browser innerhalb einer Java VirtualMachine emuliert.
Die Anwendung kann hier live getestet und - entsprechende Break-Points vorausgesetzt - auch debuggt werden.

Skriptsprachen-Programmierer müssen ihr 'Try and Run' Konzept nicht verwerfen,
da der Hosted-Mode-Browser sie weder durch umständliche Entwicklungsumgebungs-Prozesse,
noch durch lange Kompilierzeiten ausbremst.
Änderungen im Java Quelltext Code werden 'Just in time' übernommen.

Wurde entschieden den fertig konfigurierten Tomcat-Server des Toolkits zur Prototype-Backend-Implementierung zu verwenden,
kann der komplette Anwendungszyklus debuggt werden.
Dadurch kann ein GUI Klick von vorne, bis hinten debuggt werden, wie es sonst mit AJAX nicht möglich gewesen wäre.

Wenn die Anwendung fertig erstellt ist und sie mit dem Code im Hosted-Mode zufrieden sind,
kann die Anwendung mit dem GWT-Compiler in JavaScript- und HTML-Code umgewandelt werden.

Am Ende müssen die Dateien nur noch auf dem Webserver deployed werden.

== Compiler ==
Der Compiler ist das Herzstück des GWT. Er verarbeitet den programmierten Java-Code und erstellt daraus einen neuen Code aus HTML und JavaScript -
so wie ein klassischer Java Compiler aus Java-Code einen Bytecode erstellt.
Der Unterschied ist, dass der erstellte Code ohne Java Runtime Umgebung und mit jedem aktuellen Browser ausgeführt werden kann.

Beim Compilieren wird entschieden welche Funktionen des GWT überhaupt genutzt werden.
Für eine simple 'Hello World' werden also keine hundert Kilobytes an Bibliotheken mitgeliefert, sondern nur die Funktionen die tatsächlich benötigt werden.

Außerdem wird der Quelltext komprimiert und komplett in JavaScript Dateien ausgelagert.
Dadurch kann die gesamte Anwendung vom Browser zwischengespeichert werden.

Für jede Kombination aus Gebietsschema und Browsertyp wird eine einzelne JavaScript Datei erstellt.
Das erzeugt zwar mehr Daten auf dem Server (meist in einer Größenordnung von Kilobytes),
aber durch ein automatisches Bootstrap-Skript wird für den Benutzer nur die Datei geladen,
die er tatsächlich benötigt.

Wurde vor einen Framework-Einsatz vorher noch zwischen sauberem Code und Mengen an unbenutztem Framework-Code abgewägt,
kann mit dem GWT fast nur gespart werden.

Die eigentliche HTML Datei sorgt nur für die Einbindung des JavaScripts.

== ImageBundles ==
Ein weiterer Grundprozess des GWT ist es die Anzahl an HTTP-Anfragen so gering wie möglich zu halten.
Auch hierfür sorgt der Compiler.
In den meisten Websites sind es Bilder, die die meisten HTTP-Anfragen verursachen.
Daher haben sich die GWT Entwickler das Konzept der ImageBundles überlegt.
Vielleicht wird der Vorteil anhand eines Beispiels bewusster:

Typische WYSIWYG-Editoren bringen etwa 50 Buttons mit, die dem Anwender eine Office-ähnliche Oberfläche vermitteln sollen.
Ein Browser würde an dieser Stelle 50 Bilder-Tags im HTML-Quelltext sehen und alle 50 Bilder per HTTP-Anfrage laden.
Selbst wenn der Besucher die Seite bereits besucht hat und alle Bilder nur aus dem Zwischenspeicher des Browsers geladen werden müssen,
müssen 50 Verbindungen aufgebaut werden um nachzufragen, ob sich eines der Bilder geändert hat.
Das GWT macht nun aus den 50 Bildern nur ein einziges Bild, das die selbe Datenmenge mitbringt, nur einmalig geladen werden muss und trotzdem zwischengespeichert wird.
Eingebunden werden die Bilder weiterhin über die 50 Bilder-Tags in ihrer Originalgröße.
Per CSS werden Sie aber so verschoben, dass der Endanwender, bis auf den Geschwindigkeitsvorteil keinen Unterschied mehr merkt.
easy-coding.de/Attachment/496/…8be23648cae18e15f88161a29

ImageBundles werden als Interface geschrieben.
Sollen die Bilder zentral verwaltet werden, werden sie per Ableiten zusammengefasst und der Compiler erstellt ein einziges Bild.

== Remote Procedure Call ==
Bisher wurden die optimierten Frontend-Prozesse des GWT vorgestellt.
Für die Anbindung an ein Backend nutzt das GWT 'Remote Procedure Calls' (kurz RPC),
über die eine Model-View-Controller Architektur umgesetzt werden kann.

Für die Prototyp-Entwicklung des Backends bringt das Google Web Toolkit einen Tomcat Server mit.
Das Toolkit fasst also alles benötige in einem Paket zusammen.
Falls kein Applikation-Server zur Verfügung steht,
stellt dies durch die Standardisierung von Remote Procedure Calls auch kein Problem dar und
es können andere Programmiersprachen wie PHP, Python oder C\# im Backend genutzt werden.

Eine Besonderheit der RPC Implementierung des GWT,
die das Debuggen erleichtern,
sind die polymorphen Klassenhierarchien und das Durchreichen von Exceptions.

Die Implementierung von RemoteProcedure Calls erstreckt sich über mehrere Klassen und Interfaces.
Aufbauend auf dem 'Hello World' wird nun die Beispielanwendung der AJAX-Programmierung implementiert.

Nach Konvention sollten dazu neben dem existierenden Client-Paket noch 'rpc' und 'server' Pakete angelegt werden.
Das rpc-Paket wird die folgende Klassenstruktur abbilden, die zuständig für die Remote Procedure Calls sind.
Komplexe Datentypen sind erlaubt.
Daher wird das rpc-Paket auch die Klassen enthalten, die über HTTP ausgetauscht werden.
easy-coding.de/Attachment/497/…8be23648cae18e15f88161a29

Status ist ein Interface das RemoteService erweitert und die tatsächliche Implementierung beschreibt.
Es enthält die Java-Signaturen und wird später vom Backend implementiert.

Quellcode

```
1. import com.google.gwt.user.client.rpc.*;
2. public interface Status extends RemoteService {
3. Person[] getActive();
4. void setStatus(Person name, boolean status);
5. }
```

StatusAsync ist ein Interface das später im GWT-Frontend-Code instanziiert wird.
Die Methoden haben keinen Rückgabetyp, erhalten jedoch das parametrisierte AsyncCallback Objekt, das auf die readyState Änderungen reagiert.

Quellcode

```
1. import com.google.gwt.user.client.rpc.*;
2. public interface StatusAsync {
3. void getActive(AsyncCallback callback);
4. void setStatus(Person name, boolean status, AsyncCallback callback);
5. }
```

Der Status-Proxy beschreibt die Instantiierung im Frontend.
Er ist eine StatusAsync Instanz, wird aber vom GWT initialisiert.
Hier wird er die URL des RPC Moduls des GWT genutzt.
Alternativ kann man aber auch einen Pfad zur eigenen RPC Implementierung eintragen.

Quellcode

```
1. StatusAsync status =
2. (StatusAsync) GWT.create(Status.class);
3. ServiceDefTarget endpoint = (ServiceDefTarget) status;
4. String moduleRelativeURL = GWT.getModuleBaseURL() + "rpc";
5. endpoint.setServiceEntryPoint(moduleRelativeURL);
6. AsyncCallback callback = new AsyncCallback() {
7. public void onSuccess(Object result) {
8. Person[] list = (Person[]) result;
9. for(int i=0; i<list.length; i++) {
10. Window.alert(list[i].getName());
11. }
12. }
13. public void onFailure(Throwable caught) {
14. Window.alert(caught.getMessage());
15. }
16. };
```

Alles anzeigen

Wird die Anwendung mit dem mitgelieferten Tomcat-Server getestet,
muss das Backend das Status-Interface implementieren und der Pfad zum Servlet in der gwt-Modulkonfiguration eingefügt werden.
In der public/MeineErsteAnwendung.gwt.xml wird dazu ein neuer XML-Knoten unter '<module>' eingefügt.

Quellcode

1. <servlet path="/rpc" class="com.gwt.server.StatusImpl"/>

== Elemente des GWT ==
Neben den behandelten Performance-Vorteilen, durch die erst eine Reaktionszeit ähnlich der von Desktop- möglich wurde, bringt das GWT Basiselemente für die GUI-Programmierung und Konzepte um die Usability von Web-Anwendungen zu erhöhen mit.

== Panels ==
Wer grafische Oberflächen bereits mit Java erstellt hat, kennt das Layout-Manager-Konzept von Swing und AWT.
Abgesehen von ein paar Besonderheiten von HTML-Elementen wird das Konzept beliebig kombinierbarer Panels auch im GWT angewandt.

Panels sind Container um andere Elemente aufzunehmen.
Es gibt verschiedene Panels für Layout und Content.
Ein typisches Beispiel für ein Layout-Panel ist das DockPanel.
Es kann als Grundlayout verwendet werden und ähnelt dem BorderLayout von AWT.
easy-coding.de/Attachment/498/…8be23648cae18e15f88161a29

== Widgets ==
Hinter Widgets versteckt sich der deutsche Begriff 'grafisches Bedienelement'.
Das GWT versteht sich auf Webanwendungen, und so wird es auch die gewohnten Browser-Elemente aus Usability-Aspekten behalten.
Neben den für HTML Entwickler bekannten Elementen wie Eingabefeld, Button und Auswahlbox bringt das GWT aber noch weitere, aus Desktop-Applikationen bekannte Widgets mit.
Dazu zählen der Tree für eine Baumdarstellung eines Verzeichnisbaums,
die MenuBar für eine Menüleiste,
die TabBar für das Container-Wechsel über eine Reiterleiste,
das StackPanel oder die RichTextArea für WYSIWYG-Editoren.

Außerdem gibt es neben Kalender- und Zeichenwidgets auch 'Drag \& Drop' Elemente aus der aktiven OpenSource Community hinter dem GWT.

== Implementierung der Browser-History ==
Ein typischer Kritikpunkt und Stolperstein bei Web-Anwendungen ist die Browserhistorie,
die entweder nicht funktioniert oder von der kontextuell abgeraten wird.

Das asynchrone Konzept von AJAX macht ein Neuladen der Website unnötig, wodurch es für den Browser keinen Anlass gibt einen History-Eintrag zu erstellen.
Dadurch, dass sich für den Benutzer aber Inhalte ändern, nimmt er an, dass er den 'Zurück'-Button nutzen kann.

Es gibt verschiedene Workarounds für dieses Problem, die sich zu Quasi-Entwurfsmustern entwickelt haben, aber nicht einfach zu implementieren sind.

Das GWT hat diese Implementierung bereits vorgenommen und kann vom Programmierer abstrakt genutzt werden.
Hier zeigt sich wieder der Vorteil eines Frameworks:
Gibt es in der Zukunft eine bessere Technik braucht der Entwickler nur die Google JAR auszutauschen.

Die Implementierung funktioniert mit der Schnittstelle HistoryListener, die onHistoryChanged(String token) implementiert.
Das Token lässt sich wie ein Dateiname behandeln.
Dadurch kann jede GUI-Aktion durch ein Token abgebildet werden.

== Internationalisierung ==
Das GWT unterstützt den Entwickler mit verschiedenen Möglichkeiten um sowohl die Sprache,
als auch unterschiedliche Zahlen- oder Datumsformaten zu internationalisieren.

Die bevorzugte Art der Implementierung ist die statische bei der die Sprachvariablen schon zur Compilierzeit befüllt werden.
Wählt man stattdessen die dynamische Internationalisierung werden dem Benutzer alle Dateien mitgeliefert.

Bei der statischen Variante wird ein Interface mit je einer Methode pro Eigenschaft erstellt.

Quellcode

```
1. public interface MyLang extends Constants {
2. String helloWorld();
3. String sliceOfSpam();
4. }
```

Eine Instanz dieser Schnittstelle greift dann auf eine Property-Datei namens MyLang.properties zu.
Wurde eine weitere Sprache (z.B. 'en') über die globale Konfigurationsdatei gwt.xml bekannt gemacht,
wird auch für die Sprache 'en' und die Datei MyLang_en.properties ein JavaScript kompiliert.

Die Sprache wird je nach Browserkonfiguration automatisch gewählt.
Mit dem GET-Parameter 'locale' lässt sie sich aber auch erzwingen.

== Eigene Scripte ==
Es ist gängige Praxis neue Widgets zu bauen indem vorhandene Widgets und Panels des GWT kombiniert werden.
Das erleichtert die Pflege bei Versionsupdates und verhindert, dass der Programmierer wieder auf JavaScript Ebene herunter muss um Änderungen einzupflegen.

Falls aber doch individuelle Effekte, fertige Bildergallerien oder Ähnliches benötigt werden,
bietet das GWT über JSNI die Möglichkeit normalen JavaScript Code einzubinden.

Soll eine solche Methode geschrieben werden, wird das Schlüsselwort 'native' und eine spezielle Kommentarsyntax verwendet.
Wie im folgenden Beispiel zu sehen, ist selbst ein Zugriff auf Attribute und Methoden des GWT möglich.

Der Zugriff auf 'window' und 'document' ist gekapselt und nur über die Variablen \$wnd und \$doc möglich.

Quellcode

```
1. public class JSNIExample {
2. void foo(String s) {
3. private FlowPanel fp = new FlowPanel();
4. private Button b1 = new Button(s+" 1");
5. private Button b2 = new Button(s+" 2");
6. }
7. public native void foo(JSNIExample x, String s) /*-{
8. $wnd.alert(msg);
9. this.@book.JSNIExample::foo(Ljava/lang/String;)(s);
10. }-*/;
11. }
```

Alles anzeigen