

4 GWT unter der Lupe

In diesem Kapitel wird das Webframework GWT genauer unter die Lupe genommen. Dabei wird vor allem auf die Architektur und auf die speziellen Eigenschaften von GWT näher eingegangen.

4.1 Aufbau der Architektur

Eine Architektur beschreibt den strukturellen Aufbau einer Webanwendung mit all ihren Services und Diensten. Dabei erfolgt eine Zerlegung des Systems in Komponenten und deren Schnittstellen und Beziehungen die logisch «vertikal» und eventuell im Netzwerk »horizontal« aufgeteilt werden können. Im Folgenden wird konkret auf die Architektur und den Aufbau einer GWT-Anwendung eingegangen.

Die Architektur des GWT ist in Abbildung 4-1 dargestellt und besteht aus folgenden vier Komponenten, die in zwei Gruppen zusammengefasst werden können:

Class Libraries	JRE emulation library	GWT Web UI class library
Development Tools	GWT Java-to-JavaScript Compiler	GWT Hosted Web Browser

Abb. 4-1
GWT-Architektur

4.1.1 Development Tools

Bei den Development Tools handelt es sich um Hilfsmittel, die dem Entwickler zur Verfügung stehen und den Softwareentwicklungsprozess vereinfachen sollen.

*Java-to-JavaScript-
Compiler*

Der GWT-Java-to-JavaScript-Compiler stellt das Herzstück des GWTs dar. Dieser Compiler wandelt Java-Quelltext in äquivalenten JavaScript-Quelltext um. Bei dieser Umwandlung gibt es jedoch Einschränkungen beim erlaubten Java-Sprachumfang. Diese Einschränkungen werden in Abschnitt 4.3.1 erklärt.

Hosted Web Browser

Der GWT-Hosted-Webbrowser ist speziell für den in der Entwicklungsphase verwendeten Hosted Mode (siehe Abschnitt 4.4.1) gedacht. GWT stellt einen Webbrowser zur Verfügung, der im JVM-Prozess läuft. Dadurch können Interaktionen mit dem Browser abgefangen und in der JVM verarbeitet werden. So wird das Debuggen der Webanwendung ermöglicht.

4.1.2 Class Libraries

Die Class Libraries stellen notwendige Klassenbibliotheken dar, die die unterschiedlichen GWT-Betriebsarten (siehe Abschnitt 4.4) ermöglichen.

JRE-emulation-library

Die **JRE-emulation-library** beinhaltet eine Implementierung der Java-Standardklassenbibliothek, jedoch beschränkt auf die Möglichkeiten von JavaScript in den aktuellen Browsern. Die Implementierung ist im Quelltext in der Datei `gwt-user.jar` (`com.google.gwt.emul`) des GWT-Frameworks einsehbar. Die Klassenbibliothek beinhaltet den Großteil des `java.lang`-Packages, einen Teil der `java.util`-Packages und zusätzlich die Klasse `Serializable` des Package `java.io`. Die Einschränkungen werden in Kapitel 4.3.2 genauer erläutert.

*GWT Web UI class
library*

Die GWT Web UI class library stellt die eigentliche Benutzerschnittstellenbibliothek, die eine Sammlung von Standard-Widgets beinhaltet, dar. Eine detaillierte Beschreibung dieser Widgets erfolgt in Abschnitt 6.

4.2 Multi-Tier-Architektur

Bei einer Multi-Tier-Architektur erfolgt eine vertikale und horizontale Aufteilung der Basiskomponenten einer Webanwendung auf unterschiedlichen physikalischen Servern. Nach Abbildung 4-2 verwendet das GWT die für Webanwendungen übliche 3-Schichten-Architektur, wobei eine horizontale Aufteilung der vertikalen Schichten »Presentation-Layer«, »Business-Layer« und »Persistence-Layer« im

Netzwerk durchgeführt werden kann. Bei kleineren Webanwendungen ist jedoch nur der Presentation-Layer (Webbrowser) horizontal verteilt.

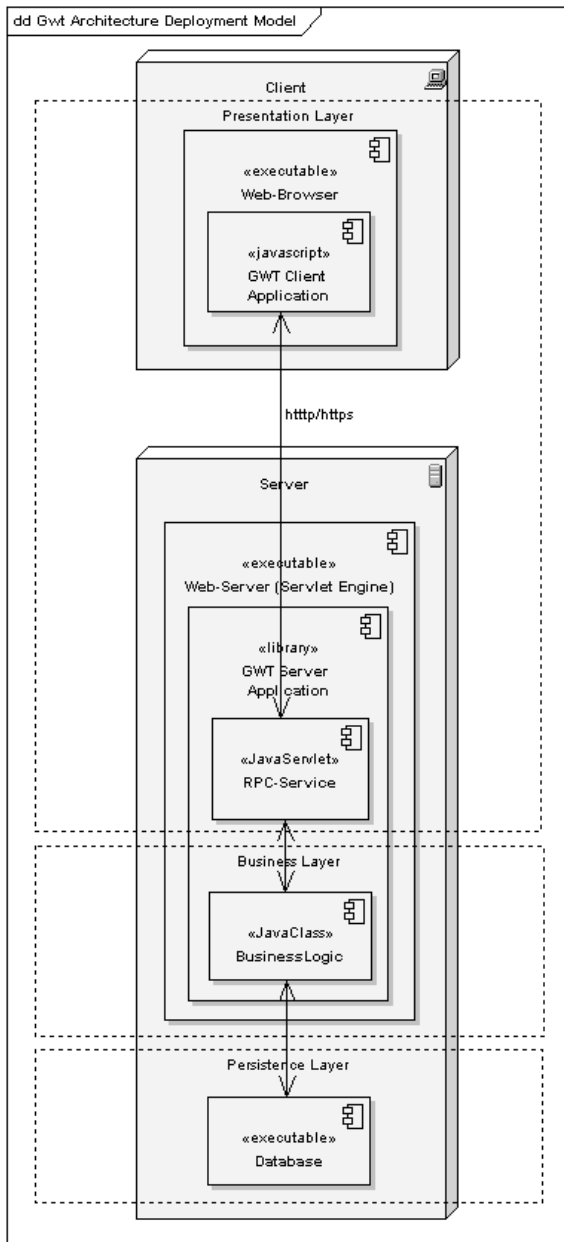


Abb. 4-2
GWT-multitier-
Architektur

Eine GWT-Webanwendung besteht aus einem clientseitigen und einem serverseitigen Applikationsteil, wobei der clientseitige Teil das in JavaScript übersetzte Frontend (»Presentation Layer«) darstellt. Das

Rendering des Frontends erfolgt vom Webbrowser. Falls Benutzerinteraktionen einen Aufruf der Anwendungslogik erfordern, wird mittels AJAX ein asynchroner Remote Procedure Call (RPC) zum RPC-Service, das als Java Servlet realisiert ist, durchgeführt. Innerhalb des RPC-Services kann dann ein Aufruf der Anwendungslogik (»Business-Layer«) im Java-Kontext durchgeführt werden. Falls die Anwendungslogik Daten von der Datenbank benötigt, können gegebenenfalls Daten vom »Persistence-Layer« angefordert werden.

4.3 JavaScript-Compiler

Der GWT-Compiler stellt, wie schon erwähnt, das Herzstück des Webframeworks dar und ist durch die Java-Klasse `GWTCompiler` realisiert. Dieser Compiler wandelt das in Java-Code geschriebene Frontend in JavaScript um und erfüllt somit eine ähnliche Aufgabe wie der herkömmliche Java-Compiler `javac`, der den Java-Code in Byte-Code kompiliert. Er kommt dann zum Einsatz, wenn die Webanwendung im Web Mode (siehe Abschnitt 4.4.2) oder in der eigentlichen »Produktionsumgebung« betrieben wird.

Der Compiler beherrscht drei Betriebsarten, die die Lesbarkeit des übersetzten JavaScripts beeinflussen.

Bei der Betriebsart `obfuscate` wird der kompilierte JavaScript-Code »verschleiert«, was soviel bedeutet, dass der endgültige JavaScript-Code einer unlesbaren »Buchstabensuppe« gleichkommt (Variablen- und Funktionsnamen bestehen aus zufälligen Zeichen). Etwaige Rückschlüsse auf die eigentliche Semantik des Codes werden dadurch unmöglich, jedoch ist er syntaktisch korrekt und erfüllt die funktionale Aufgabe. Diese Betriebsart ist die Standardbetriebsart (siehe Listing 4.1).

Listing 4.1

GWT-Compiler-Mode:
`obfuscated`

```
1 function r(){r = a;s = t('N',[0],[10],[0],null);return window;}
```

Die Betriebsart `pretty` kompiliert den JavaScript-Code als »leserlich«, was sich durch sprechende Variablen- und Funktionsnamen und einer entsprechenden Formatierung (Einrückungen) äußert. Der kompilierte JavaScript-Code ist in Listing 4.2 ersichtlich.

Listing 4.2

GWT-Compiler-Mode:
`pretty`

```
1 function _$clinit(){
2   _$clinit = _nullMethod;
3   _NO_STACK_TRACE = _initDims('N', [0], [10], [0], null);
4   return window;
5 }
```

Die letzte Betriebsart *detailed* ist mit der Betriebsart *pretty* vergleichbar, jedoch wird zusätzlich der gesamte Klassenname der übersetzten Java-Klasse bei der Bezeichnung der dazugehörigen JavaScript-Methode angegeben. Dadurch wird nach Listing 4.3 das Debugging erleichtert und es werden Rückschlüsse zur übersetzten Java-Klasse möglich.

```
1 function java_lang_Throwable_$clinit__(){
2   java_lang_Throwable_$clinit__ = nullMethod;
3   java_lang_Throwable_NO_1STACK_1TRACE = com_google_
4     gwt_lang_Array_initDims__Ljava_lang_String_
5     2Ljava_lang_Object_2Ljava_lang_Object_2Ljava_
6     lang_Object_2Ljava_lang_Object_2 ('[N',
7     [0], [10], [0], null);
```

Listing 4.3
GWT-Compiler-Mode:
detailed

4.3.1 Spracheinschränkungen für Java

Abhängig davon, welche GWT-Version verwendet wird, gibt es unterschiedliche Einschränkungen.

Die größte Einschränkung des Compilers besteht bei der Kompatibilität der verwendeten Java-Version. In der GWT-Version 1.4 wird die JSE-Version 1.4 unterstützt und ab der GWT-Version 1.5 die JSE-Version 5.0. Das in Java verfasste Frontend kann also dann erfolgreich kompiliert werden, wenn der Java-Source-Code zur unterstützten JSE-Version oder einer früheren Version kompatibel ist. Wenn die mächtigen Spracherweiterungen der JSE 5.0 benutzt werden sollen – wie Generics, Enums, Annotations, erweiterte Schleifen und statische Imports –, muss die GWT-Version 1.5 eingesetzt werden.

Zusätzlich zur Java-Version bestehen noch folgende Einschränkungen bei der Verwendung der Sprache Java im von GWT zu JavaScript kompilierten Code.

Die primitiven Datentypen `byte`, `char`, `short`, `int`, `long`, `float`, `double`, die Objekttypen `java.lang.Object`, `java.lang.String` und die Arrays werden ohne Einschränkungen unterstützt.

Intrinsische Typen

Leider gibt es keinen ganzzahligen 64-bit-Typ `long` in JavaScript. In GWT 1.4 wird also ein Mapping zum Double-Precision Floating Point JavaScript-Typ `Number` durchgeführt. Es wird jedoch empfohlen, in diesem Fall den primitiven Datentyp `int` zu verwenden. In GWT 1.5 ist die Lösung besser: Es wird ein Paar von 32-bit Integer-Zahlen verwendet, um einen `long` zu simulieren. Das klappt allerdings nicht, wenn `long`-Werte nativ in JSNI-Code verwendet werden sollen. Hier sollte unabhängig von der Version `double` verwendet werden. Es kann auch der Wrapper `LONG` eingesetzt werden – bei der Verwendung des Wrappers gibt es keine Einschränkung bezüglich JSNI. Zusätzlich kann in GWT

1.5 die Annotaton `com.google.gwt.core.client.UnsafeNativeLong` verwendet werden, um `long`-Werte auch in JSNI benutzen zu können. Der Wert darf dann aber nur übergeben und nicht geändert werden.

Exceptions Die Schlüsselwörter `try`, `catch`, `finally` und selbst definierte Exceptions werden ohne Einschränkungen unterstützt. Die Ausgabe des Stack Traces mit der Methode `Throwable.getStackTrace()` wird im Web Mode nicht unterstützt.

Assertions Der Parser des Compilers berücksichtigt zwar das Java-Schlüsselwort `assert`, es wird aber kein entsprechender JavaScript-Code generiert.

Multithreading und Synchronization Die JavaScript-Interpreter (Webbrowser) sind `single-threaded`. Dadurch werden Multithreading-Mechanismen wie Objektsynchronisation ignoriert. Das betrifft das Schlüsselwort `synchronized` und die Methoden `Object.wait()`, `Object.notify()` und `Object.notifyAll()`.

Reflection Aus Gründen der Effizienz wird der zu übersetzende Java-Code in einen monolithischen JavaScript-Codeblock kompiliert. Dadurch ist auf der JavaScript-Seite die Reflection ausgeschlossen, jedoch kann der Name der Klasse von einer konkreten Objektinstanz mit Hilfe der Utility-Klasse `GWT.getTypeName(Object)` ermittelt werden. Ab der Version 1.5 soll diese Methode nicht mehr verwendet werden und wird in zukünftigen Versionen nicht mehr unterstützt. Als Alternative werden die Methode `Object.getClass` und `Class.getName()` angeboten. Diese gehören zum normalen Java-Sprachumfang und ermöglichen zusammen das Gleiche wie die Methode `GWT.getTypeName(Object)`.

Finalization Die Skriptsprache JavaScript unterstützt den Aufruf von Objektdestruktoren während der Garbage Collection nicht. Dadurch wird die Methode `Object.finalize()` im Web Mode bei der Garbage Collection nicht aufgerufen.

Strict Floating-Point Die Java-Sprachspezifikation definiert eine sehr präzise Floating-Point-Unterstützung. Diese umfasst auch Single-Precision- und Double-Precision-Zahlen, sowie das `strictfp`-Schlüsselwort, das GWT leider nicht unterstützt. Es kann daher nicht garantiert werden, dass Floating-Point-Variablen eins zu eins übersetzt werden. clientseitige Berechnungen sollten also nach Möglichkeit reduziert werden.

4.3.2 Klassenbibliothekseinschränkungen für Java

Zusätzlich zu den Einschränkungen bei der Sprache Java gibt es auch Einschränkungen bei der Klassenbibliothek (JRE-emulation-library, siehe Abschnitt 4.1.2). Für das clientseitige Frontend können daher nur folgende Klassen und Interfaces aus dem Package `java.lang` verwendet werden:

■ **Klassen:**

ArrayStoreException, AssertionError, Boolean, Byte, Character, Class, ClassCastException, Double, Error, Exception, Float, IllegalArgumentException, IllegalStateException, IndexOutOfBoundsException, Integer, Long, Math, NegativeArraySizeException, NullPointerException, Number, NumberFormatException, Object, RuntimeException, Short, String, StringBuffer, StringIndexOutOfBoundsException, System, Throwable, UnsupportedOperationException

■ **Interfaces:**

CharSequence, Clonable, Comparable

In der GWT-Version 1.5 werden zusätzlich

■ **Klassen:**

StringBuilder, Enum

unterstützt.

Aus dem Utility-Package `java.util` können folgende Klassen und Interfaces verwendet werden.

■ **Klassen:**

AbstractCollection, AbstractList, AbstractMap, AbstractSet, ArrayList, Arrays, Collections, Date, EmptyStackException, EventObject, HashMap, HashSet, MissingResourceException, NoSuchElementException, Stack, TooManyListenersException, Vector

■ **Interfaces:**

Collection, Comparator, EventListener, Iterator, List, Map, RandomAccess, Set

In der GWT-Version 1.5 werden zusätzlich

■ **Klassen:**

EnumSet, EnumMap, PriorityQueue, HashMap, TreeMap, TreeSet

unterstützt.

Zu guter Letzt bietet das Package `java.io` auch noch folgendes Interfaces.

■ **Interface:**

Serializable

Das Interface `java.io.Serializable` hat eine Sonderstellung bei der Verwendung des Remote Procedure Calls (RPCs), das genauer im Kapitel 5 erläutert wird.

Zusätzlich zu den Einschränkungen gibt es noch Abweichungen der GWT-JRE-emulation-library zur Standard Java-JRE.

Regular Expressions

Die Syntax der »Java Regular Expressions« ist jener der »JavaScript Regular Expressions« sehr ähnlich, jedoch nicht identisch. Man sollte daher bei der Verwendung sehr vorsichtig sein, und nur jene Konstrukte verwenden, die in Java und in JavaScript dieselbe Bedeutung haben.

Serialization

Die Serialisierung von Objekten ist in JavaScript nicht möglich, da Mechanismen wie *Dynamic Class Loading* und *Reflection* nicht zur Verfügung stehen. GWT verwendet einen eigenen Mechanismus der im Rahmen des RPCs, siehe Kapitel 5 zur Anwendung kommt.

4.3.3 JavaScript Native Interface

Ein sehr wichtiger Aspekt des GWT-Compilers ist das JavaScript Native Interface (JSNI), das es ermöglicht, selbst geschriebenes JavaScript in den Java-Code zu mischen. Um dies zu gewährleisten, erfolgt die Implementierung des JSNI ähnlich dem Java Native Interface (JNI)-Konzepts. So können Low-Level-Funktionalitäten, die mit Java nicht realisiert bzw. nur schwer realisiert werden können, in die native Programmiersprache der Ausführungsumgebung »ausgelagert« werden. Im Falle von JNI ist es die native Programmiersprache C/C++ für die unterschiedlichen Betriebssysteme¹, bei JSNI ist es JavaScript für die unterschiedlichen Webbrowser². Bei einer Implementierung einer Funktionalität in einer native Programmiersprache erkaufte man sich jedoch die erhöhte Flexibilität durch eine mögliche Inkompatibilität gegenüber den unterschiedlichen Ausführungsumgebungen.

JSNI bietet folgenden Funktionsumfang:

- Implementierung einer Java-Methode direkt in JavaScript
- »Wrappen« von typischeren Java-Methodensignaturen um existierendes JavaScript
- Aufruf von JavaScript- und Java-Methoden und umgekehrt
- Auslösen von Exceptions über die Grenzen von Java/JavaScript hinweg
- Lesen und Schreiben von Java-Klassenattributen aus JavaScript
- Debuggen von Java- und JavaScript-Methoden mit dem Hosted Mode (siehe Abschnitt 4.4.1)

¹Betriebssysteme: Microsoft Windows, sämtliche Unix-Derivate wie Linux, Solaris, AIX, usw. und Apples MacOS

²Webbrowser: Mozilla Firefox, Microsoft Internet Explorer, Opera, Konquerer, Safari,...

Um JSNI richtig einsetzen zu können, müssen folgende vier Aspekte beachtet werden:

1. Implementierung von native JavaScript-Methoden
2. Zugriff auf Java-Methoden und Attribute via JavaScript
3. »Teilen« von Ojekten zwischen Java und JavaScript
4. Exceptions und JSNI

Im Folgenden wird auf die einzelnen Punkte näher eingegangen.

Implementierung von native JavaScript-Methoden

JSNI-Methoden werden `native` deklariert und enthalten den JavaScript-Code in einem speziell formatierten Kommentarblock zwischen dem Ende der Parameterliste und dem abschließenden Semikolon ; am Ende der Methode. Dieser »JSNI Kommentarblock« beginnt mit `/*-{` und endet mit `}-*/`, wobei der eigentliche Methodenrumpf in Java kommentiert werden muss, da gemäß der JNI-Spezifikation `native`-Methoden nur deklariert werden dürfen. Die eigentliche Implementierung wird ja »extern« innerhalb einer Dynamic Link Library (DLL) unter Windows oder eines Shared Objects (SO) unter Unix durchgeführt, die mit Hilfe des Befehls `System.loadLibrary(String)` geladen werden. Eine beispielhafte Implementierung zeigt der in Listing 4.4 dargestellte Java-Code, der ein Dialogfenster mit einer Meldung öffnet.

```
1 public class MyFirstNativeClass {
2     public static native void alert(String msg)/*-{
3         $wnd.alert(msg);
4     }-*/;
5 }
```

Listing 4.4

*JSNI native
JavaScript-Methode*

JSNI bietet zwei implizit definierte Variablen, die innerhalb der `native` Methode verwendet werden können:

- `$wnd`: Eine Referenz auf das Browserfenster
- `$doc`: Referenz auf das aktuelle Dokument, um auf dessen DOM zugreifen zu können.

Zugriff auf Java-Methoden und Attribute via JavaScript

Um Java-Methoden aus JavaScript aufzurufen, ist die Vorgehensweise ähnlich dem Aufruf von Java-Methoden aus C/C++-Code in JNI. Der Aufruf hat daher nach Listing 4.5 folgende Syntax:

Listing 4.5
 JSNI-Syntax für den
 Aufruf von
 Java-Methoden aus
 JavaScript

- ```

1 [instance-expr.]@class-name::
2 method-name(param-signature) (arguments)

```
- [instance-expr.]: Handelt es sich bei der aufzurufenden Methode um eine static-Methode, so muss dieser Ausdruck weggelassen werden. Ist jedoch die Methode an eine konkrete Objektinstanz gebunden, so muss dieser Ausdruck angegeben werden.
  - class-name: Das ist der voll qualifizierte Name der Klasse, in der die Methode deklariert ist (oder eine Subklasse dieser Klasse)
  - method-name: der eigentliche Methodename
  - param-signature: Das entspricht der internen Java-Methodensignatur, aber ohne Signatur des Rückgabewerts, da dieser nicht erforderlich ist.

**Tab. 4-1**  
 Interne Java-  
 Methodensignatur

| Typsignatur              | Java-Typ              |
|--------------------------|-----------------------|
| Z                        | boolean               |
| B                        | byte                  |
| C                        | char                  |
| S                        | short                 |
| I                        | int                   |
| J                        | long                  |
| F                        | float                 |
| D                        | double                |
| L fully-qualified-class; | fully-qualified-class |
| [type                    | type[]                |
| ( arg-types ) ret-type   | method type           |

Das folgende Beispiel zeigt die Übersetzung der Signatur der Java-Methode in die korrespondierende Typsignatur (siehe Listing 4.6).

**Listing 4.6**  
 JSNI Typsignatur

```

1 //Java-Method:
2 long f (int n, String s, int[] arr);
3 //Type signature:
4 (Ljava/lang/String;[I)J

```

- arguments: Liste der tatsächlichen Parameter

Soll der Zugriff auf Java-Attribute aus JavaScript erfolgen, so hat der Aufruf nach Listing 4.7 folgende Syntax:

```
1 [instance-expr.]@class-name::field-name
```

- [instance-expr.]: Erfolgt der Zugriff auf ein Klassenattribut (static-Attribut), so muss dieser Ausdruck weggelassen werden. Repräsentiert das Attribut jedoch den internen Zustand des Objekts, so muss dieser Ausdruck angegeben werden.
- class-name: Das ist der voll qualifizierte Name der Klasse, in der das Attribut deklariert ist (oder eine Subklasse dieser Klasse).
- field-name: der eigentliche Attributname

Das folgende Beispiel soll die unterschiedlichen Mechanismen zeigen. Die Klasse in Listing 4.8 soll als Basis für die folgenden Beispiele dienen.

```
1 package at.mycompany.myapp;
2
3 public class MyJsniClass {
4 private String myInstanceField;
5 private static int myStaticField;
6
7 private void myInstanceMethod(String s) {
8 // use s
9 }
10 private static void myStaticMethod(String s) {
11 // use s
12 }
13 }
```

Im ersten Beispiel soll die Instanzmethode `myInstanceMethod(String)`, deren Aufruf nur mit einer gültigen Objektinstanz möglich ist, innerhalb der JNI-Methode aufgerufen werden (siehe Listing 4.9).

```
1 public native void bar(MyJsniClass x, String s) /*- {
2 // Call instance method myInstanceMethod() on this
3 this.@at.mycompany.myapp.
4 MyJsniClass::myInstanceMethod(Ljava/lang/String;) (s);
5 }
```

#### Listing 4.7

*JSNI-Syntax für den Zugriff auf Java-Attribute aus JavaScript*

#### Listing 4.8

*JSNI-Beispiel: Klasse*

#### Listing 4.9

*JSNI-Beispiel: Aufruf Instanzmethode 1*

Das nächste Beispiel (siehe Listing 4.10) zeigt den Aufruf der Instanzmethode `myInstanceMethod(String)` vom Objekt `x`, das als Parameter der JNI-Methode übergeben wird.

**Listing 4.10**

JSNI-Beispiel: Aufruf  
Instanzmethode 2

```
1 public native void bar(MyJsniClass x, String s) /*-{
2 // Call instance method myInstanceMethod() on x
3 x.@at.mycompany.myapplication.
4 MyJsniClass::myInstanceMethod(Ljava/lang/String;)(s);
5 }
```

Das in Listing 4.11 dargestellte Beispiel zeigt den Aufruf der Klassenmethode `myStaticMethod(String)`.

**Listing 4.11**

JSNI-Beispiel: Aufruf  
Klassenmethode

```
1 public native void bar(MyJsniClass x, String s) /*-{
2 // Call static method myStaticMethod()
3 @at.mycompany.myapplication.
4 MyJsniClass::myStaticMethod(Ljava/lang/String;)(s);
5 }
```

Soll der lesende Zugriff auf das Instanzattribut `myInstanceField` erfolgen, so muss der Zugriff wie in Listing 4.12 beschrieben, erfolgen.

**Listing 4.12**

JSNI-Beispiel: Aufruf  
Klassenmethode

```
1 public native void bar(MyJsniClass x, String s) /*-{
2 // Read instance field on this
3 var val = this.@at.mycompany.myapplication.
4 MyJsniClass::myInstanceField;
5 }
```

Der schreibende Zugriff auf das Instanzattribut `myInstanceField` wird anhand des Parameters `x` der JNI-Methode nach Listing 4.13 gezeigt.

**Listing 4.13**

JSNI-Beispiel: Aufruf  
Klassenmethode

```
1 public native void bar(MyJsniClass x, String s) /*-{
2 // Write instance field on x
3 x.@at.mycompany.myapplication.
4 MyJsniClass::myInstanceField = val + " and stuff";
5 }
```

Zu guter Letzt zeigt das Listing 4.14 noch den Zugriff auf das Klassenattribut `myStaticField`.

**Listing 4.14**

JSNI-Beispiel: Aufruf  
Klassenmethode

```
1 public native void bar(MyJsniClass x, String s) /*-{
2 // Read static field (no qualifier)
3 @at.mycompany.myapplication.
4 MyJsniClass::myStaticField = val + " and stuff";
5 }
```

### »Teilen« von Objekten zwischen Java und JavaScript

Die Parameter und Return-Typen sind in JSNI-Methoden als Java-Typen deklariert. Dabei muss ein Typ-Mapping zwischen der Java-Welt und der Welt des JavaScripts stattfinden.

Die Übergabe von Java-Werten an JavaScript ist in Tabelle 4-2 dargestellt.

| Eingehender Java-Typ             | JavaScript-Typ                                                                                                                          |
|----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| numerischer primitiver Datentyp  | ein JavaScript numerischer Wert, wie in <code>var x = 42;</code>                                                                        |
| String                           | ein JavaScript String, wie in <code>var s = "my string";</code>                                                                         |
| boolean                          | ein JavaScript boolescher Wert, wie in <code>var b = true;</code>                                                                       |
| JavaScriptObject                 | Ein JavaScriptObject, das von JavaScript-Code erzeugt wurde. Dabei handelt es sich üblicherweise um den Return-Wert einer JSNI-Methode. |
| Java-Array                       | Ein nicht interpretierbarer Wert, der nur zum Java-Code zurückgegeben werden kann.                                                      |
| irgendein beliebiges Java-Objekt | Ein nicht interpretierbarer Wert, der über die Typsignatur (siehe Listing 4.5) referenziert werden kann.                                |

**Tab. 4-2**  
Übergabe von  
Java-Werten an  
JavaScript

Die Übergabe von JavaScript-Werten an Java ist in Tabelle 4-3 dargestellt.

| Ausgehender Java-Typ                            | JavaScript-Typ                                                                                               |
|-------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| numerischer primitiver Datentyp                 | ein JavaScript numerischer Wert, wie in <code>return 19;</code>                                              |
| String                                          | ein JavaScript String, wie in <code>return "boo";</code>                                                     |
| boolean                                         | ein JavaScript boolescher Wert, wie in <code>return false;</code>                                            |
| JavaScriptObject                                | ein natives JavaScript-Objekt, wie in <code>return document.createElement("div");</code>                     |
| irgendein beliebiges Java-Objekt (inkl. Arrays) | Ein Java-Objekt, das innerhalb von Java-Code erzeugt worden ist. Es kann in JavaScript nicht erzeugt werden. |

**Tab. 4-3**  
Übergabe von  
JavaScript-Werten an  
Java

### Wichtige Anmerkungen

- Ein Java primitiver Datentyp ist vom Typ `byte`, `short`, `char`, `int`, `long`, `float`, `double`. Es muss sichergestellt werden, dass der Wert für den angegebenen Typ passend ist. Die Rückgabe von 3.7 für den Typ `int` ruft unvorhersehbares Verhalten hervor.

- Java null und JavaScript null sind identisch und legale Werte für Objekttypen. JavaScript undefined hingegen entspricht nicht null. Deshalb sollte von einer JSNI-Methode nie undefined zurückgeliefert werden.
- Um auch native JavaScript-Objekte im Java-Code darstellen zu können, wird das Objekt JavaScriptObject verwendet. Dieses Objekt wird vom GWT-Compiler und vom Hosted Browser (siehe Hosted Mode Abschnitt 4.4.1) speziell behandelt.

### Exceptions und JSNI

Exceptions können sowohl in Java als auch in selbst geschriebenem JavaScript ausgelöst werden. Eine Exception, die in dieser JSNI-Methode ausgelöst und innerhalb vom Java-Code abgefangen wird, kann als JavaScriptException behandelt werden. Wenn eine JSNI-Methode eine Java-Methode aufruft, wird ein komplexer Aufruf-Stack abgearbeitet. Eine Exception die innerhalb der Java-Methode ausgelöst wird, kann »sicher« durch die JSNI-Methode zurück zum eigentlichen Java-Aufrufspunkt gelangen.

## 4.4 Hosted Mode vs. Web Mode

Für die Entwicklung einer Webanwendung mit dem GWT-Framework bietet Google zwei unterschiedliche Modi. Der Entwickler hat die Möglichkeit, seine Applikation im Hosted Mode über eine grafische Konsole zu testen, wobei in diesem Modus der Java-Code der Anwendung direkt ausgeführt wird. Im Web Mode wird der clientseitige Teil der Anwendung (das Frontend) gänzlich zu JavaScript kompiliert (siehe Abschnitt 4.4.2) und ausgeführt. Dieser Modus entspricht der eigentlichen Ausführungsumgebung, in der das Deployment der Anwendung durchgeführt wird.

### 4.4.1 Hosted Mode

Jeder Softwareentwickler einer GWT-Webanwendung wird die meiste Zeit der Implementierung im Hosted Mode arbeiten. Bei diesem Modus findet keine clientseitige Kompilierung des Java-Codes in JavaScript statt. Die gesamte Webanwendung wird innerhalb einer Java Virtual Machine (JVM) wie eine »normale« Java-Anwendung ausgeführt. Das zeitraubende Deployment der Webanwendung entfällt und der sich wiederholende Entwicklungszyklus von »Codierung-Testen-Debuggen« kann möglichst effizient durchgeführt werden.

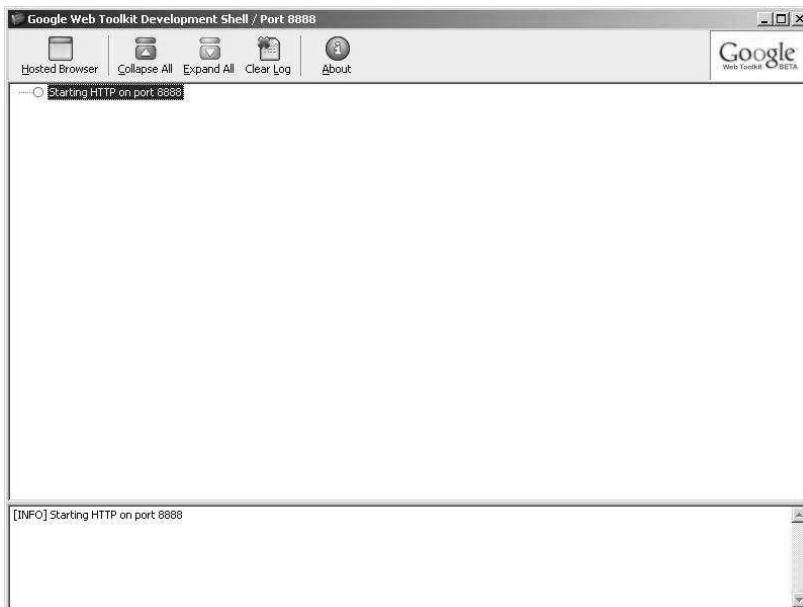
Um eine GWT-Anwendung im Hosted Mode zu starten, muss die GWT-Shell mit Hilfe der Java-Klasse `GWTShell` vom Java Application Archive (JAR) `gwt-dev-windows.jar` bzw. `gwt-dev-linux.jar` gestartet werden. Zusätzlich muss noch die eigentliche GWT-Bibliothek `gwt-user.jar` im Klassenpfad gesetzt sein. Der in Listing 4.15 angegebene Befehl zum Aufruf der GWT-Shell bezieht sich auf die beschriebene Demoapplikation.

```
1 @java -cp
2 "%-dp0\src;%-dp0\bin;
3 %GWT_HOME%/gwt-user.jar;%GWT_HOME%/gwt-dev-windows.jar"
4 com.google.gwt.dev.GWTShell -out "%-dp0\www" %*
5 at.tw.bicss.gwt.MyApp/MyApp.html
```

**Listing 4.15***Starten der GWT-Shell*

Die Möglichkeiten der Parametrisierung der GWT-Shell werden in Tabelle 4-4 dargestellt.

Der in Listing 4.15 angegebene Befehl führt zum Fenster der Google-Web-Toolkit-Development-Shell, die in Abbildung 4-3 zu sehen ist. Dieses Fenster ist in zwei Teile geteilt, wobei der obere Teil für Ausgaben von Statusinformationen der GWT-Shell dient, im unteren Teil werden die Logging-Informationen gemäß dem eingestellten Log-Level ausgegeben. Alle Statusinformationen können entweder zugeklappt oder aufgeklappt werden, die Logging-Informationen gelöscht und der Hosted Browser gestartet werden.

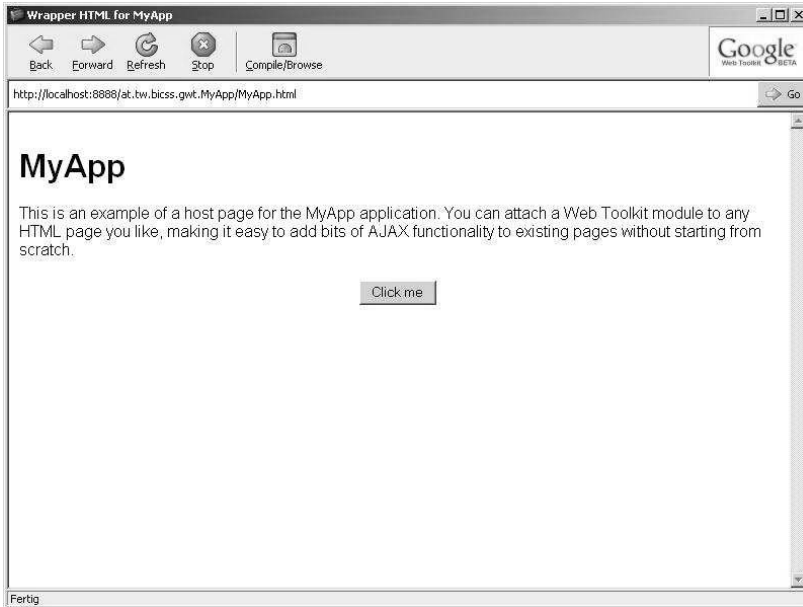
**Abb. 4-3***Hosted Mode:  
Google-Web-Toolkit-  
Development-Shell*

**Tab. 4-4**  
GWT-Shell:  
Aufrufparameter

| Parameter         | Beschreibung                                                                                                                                                                                                                                                                                                                                    |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -noserver         | Bei der Angabe dieses Parameters wird das Starten des built-in Tomcats von GWT verhindert. Dieser Parameter ist vor allem dann relevant, wenn man seine eigene Servlet-Engine nutzen möchte.                                                                                                                                                    |
| -whitelist "list" | Wird im Hosted Mode eine URL außerhalb des Applikationsbereichs aufgerufen, wird standardmäßig eine Browser-Fehlermeldung angezeigt. Soll dieses Verhalten verändert werden, so müssen die relevanten URLs in der Form einer Regular Expression zur »White List« hinzugefügt werden: <code>-whitelist "http[:] [/] [/]www[.]google[.]at"</code> |
| -blacklist "list" | Die Black List stellt eine Steigerungsstufe der White List dar und verhindert gänzlich eine Navigation zu einer externen URL, falls diese in der Black List vorhanden ist. Es wird lediglich eine Security-Fehlermeldung in der GWT-Shell angezeigt.                                                                                            |
| -logLevel level   | Der standardmäßige Log-Level für den Hosted Mode ist INFO. Insgesamt sind sieben Log-Level möglich (ERROR, WARN, INFO, TRACE, DEBUG, SPAM, ALL), wobei ALL die Ausgabe aller Logging-Informationen bedeutet.                                                                                                                                    |
| -gen "location"   | Ausgabeverzeichnis des GWT-Generators, der es ermöglicht, eine GWT-Rumpfapplikation zu erzeugen.                                                                                                                                                                                                                                                |
| -out "location"   | Ausgabeverzeichnis für alle erzeugten GWT-Dateien.                                                                                                                                                                                                                                                                                              |
| -style value      | Dieser Parameter steuert die Betriebsart des GWT-Compilers (siehe Abschnitt 4.4.2). Mögliche Werte sind: OBFUSCATED, PRETTY, DETAILED.                                                                                                                                                                                                          |

Der Hosted Browser, der in Abbildung 4-4 dargestellt ist, ist eine Art »Browser-Emulator«, der einerseits direkt den kompilierten Java-Code des Frontends interpretieren kann und andererseits Browser-Konzepte, wie z.B. Hyperlinks, Navigationshistorie (vorwärts, zurück), Neuladen und Abbruch des Ladens einer Webseite simuliert. Dadurch wird ein einfaches Debugging der Applikation möglich.

Soll nun der Hosted Mode-Emulationsmodus verlassen und die Anwendung im Web Mode ausgeführt werden, so kann mit »Compile/-Browse« die Webanwendung in einem Standard-Browser angezeigt werden.



**Abb. 4-4**  
 Hosted Mode: Google-  
 Web-Toolkit-Hosted  
 Browser

### Starten einer Applikation im Hosted Mode

Wird eine GWT-Webapplikation im Hosted Mode gestartet, sind sieben Schritte bis zur lauffähigen Applikation notwendig:

1. Starten der GWT-Shell von einer Entwicklungs-IDE oder nach Listing 4.15 von einer Shell eines Betriebssystems, sodass die GWT-Shell die eigentliche GWT-Applikation, in der die Startseite (in diesem Fall `MyApp.html`) geladen wird, startet.
2. Die Startseite `MyApp.html` lädt die kompilierte JavaScript-Datei `gwt.js` des Frontends, die mit Hilfe des HTML-`<script>`-Tags eingebettet ist.
3. Die JavaScript-Datei `gwt.js` sucht innerhalb der Startseite `MyApp.html` nach `<meta name="gwt:module" content="com.xyz.MyApp">` um den Modulnamen auszulesen.
4. Basierend auf dem Modulnamen wird die Moduldatei `MyApp.gwt.xml` geladen, um den Namen der Entry-Point-Klasse zu erhalten. In diesem Fall ist dieser Name `MyApp`.
5. Die Entry-Point-Klasse `MyApp` wird instanziiert und die Methode `onModuleLoad()` aufgerufen, die die eigentliche GWT-Applikation startet.
6. Die eigentliche Applikation tätigt etliche Aufrufe der GWT-Bibliothek `gwt-user.jar`.

- Die GWT-Bibliothek `gwt-user.jar` manipuliert mit Hilfe von DHTML das DOM des Browsers, um die UI-Komponenten darzustellen.

#### 4.4.2 Web Mode

Irgendwann kommt der Zeitpunkt, ab dem die GWT-Webanwendung in eine »Produktivumgebung« (Standard-Browser, eigene Servlet-Engine,...) integriert werden soll. Um eine GWT-Anwendung im Web Mode ausführen zu können, muss der clientseitige Java-Code des Frontends mit Hilfe des GWT-Compilers in JavaScript kompiliert werden (siehe Abschnitt 4.3). Der GWT-Compiler wird mit Hilfe der Java-Klasse `GWTCompiler` vom JAR-Archiv `gwt-dev-windows.jar` bzw. `gwt-dev-linux.jar` gestartet. Zusätzlich muss noch die eigentliche GWT-Bibliothek `gwt-user.jar` im Klassenpfad gesetzt sein. Der in Listing 4.16 angegebene Befehl zum Aufruf des GWT-Compilers nach Listing 4.16 bezieht sich auf die Demoapplikation.

##### Listing 4.16

Starten des  
GWT-Compilers

```

1 @java -cp
2 "%-dp0\src;%-dp0\bin;
3 %GWT_HOME%/gwt-user.jar;%GWT_HOME%/gwt-dev-windows.jar"
4 com.google.gwt.dev.GWTCompiler
5 -out "%-dp0\www" %* at.tw.bicss.gwt.MyApp

```

Wie auch die GWT-Shell (siehe Abschnitt 4.4.1) kann der GWT-Compiler durch die in Tabelle 4-5 möglichen Aufrufparameter parametrisiert werden.

#### Starten einer Applikation im Web Mode

Wird eine GWT-Webapplikation im Web Mode gestartet, sind sechs Schritte bis zur lauffähigen Applikation notwendig:

- Der Webbrowser lädt die Startseite der Applikation `MyApp.html`.
- Die Startseite `MyApp.html` lädt die kompilierte JavaScript-Datei `gwt.js` des Frontends, die mit Hilfe des HTML-`<script>`-Tags eingebettet ist.
- Die JavaScript-Datei `gwt.js` sucht innerhalb der Startseite `MyApp.html` nach `<meta name="gwt:module" content="com.xyz.MyApp">`, um den Modulnamen auszulesen.
- Die JavaScript-Datei `gwt.js` modifiziert die Startseite, um einen HTML-`<iframe>` zu inkludieren. Dieser lädt die HTML-Datei `[module-name].nocache.html` (in diesem Fall `com.xyz.MyApp.nocache.html`).

| Parameter       | Beschreibung                                                                                                                                                                                                                                                                       |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -logLevel level | Der standardmäßige Log-Level für den Web Mode ist INFO. Insgesamt sind sieben Log-Level möglich (ERROR, WARN, INFO, TRACE, DEBUG, SPAM, ALL), wobei ALL die Ausgabe aller Logging-Informationen bedeutet.                                                                          |
| -treeLogger     | Formatiert den Output des Compilers im GWT-TreeLogger-Format. Dabei erfolgt eine hierarchische Ausgabe der Statusinformationen des Compilers im oberen Bereich des GWT-Shell-Fensters (siehe Abbildung 4-3), wodurch eine einfache Überprüfung des Kompilierprozesses möglich ist. |
| -gen "location" | Ausgabeverzeichnis des GWT-Generators, der es ermöglicht, eine GWT-Rumpfapplikation zu erzeugen.                                                                                                                                                                                   |
| -out "location" | Ausgabeverzeichnis für alle erzeugten GWT-Dateien.                                                                                                                                                                                                                                 |
| -style value    | Dieser Parameter steuert die Betriebsart des GWT-Compilers (siehe Abschnitt 4.3). Mögliche Werte sind: OBFUSCATED, PRETTY, DETAILED.                                                                                                                                               |

**Tab. 4-5**

GWT-Compiler:  
Aufrufparameter

- Das eingebettete JavaScript innerhalb der Datei `com.xyz.MyApp.nocache.html` versucht den Browser-Hersteller zu bestimmen, um die jeweilig spezifische HTML-Code-Datei (in diesem Fall `A14B3452E214.cache.html`) innerhalb vom HTML-`<iframe>` über einen HTTP-Redirect zu laden.
- Die übersetzte JavaScript-Methode des in Java geschriebenen Entry-Points `onModuleLoad()` wird aufgerufen, und die eigentliche Applikation gestartet.

### 4.4.3 Deployment

Die für den Web Mode (siehe Abschnitt 4.4.2) mit dem GWT-Compiler kompilierte Webanwendung kann ohne Probleme in die »Produktionsumgebung« integriert werden. Nachdem ja bereits im Web Mode ein Starten der Applikation mit einem Standard-Browser durchgeführt wird, sollte auch ein problemloser Betrieb in der gewünschten Zielumgebung möglich sein.

## 4.5 Bestandteile einer GWT-Applikation

Die Verzeichnisstruktur der generierten Demoapplikation mit den verwendeten GWT-Tools beinhaltet folgende Komponenten die in den Listings 4.17 bis 4.20 beschrieben werden.

**Listing 4.17**  
Verzeichnisstruktur  
einer GWT-Applikation

```

1 MyApp
2 |- .gwt-cache
3 | |- bytecode
4 | | |- ...
5 | |- bin
6 | |- ...

```

Die in Listing 4.17 dargestellten Verzeichnisse beinhalten den kompilierten Bytecode, wobei sich im Verzeichnis `.gwt-cache` der Bytecode der Java-Sourcen des GWT und im Verzeichnis `bin` die kompilierten Java-Sourcen des Projekts befinden.

**Listing 4.18**  
Verzeichnisstruktur  
einer GWT-Applikation

```

1 |- src
2 | |- at
3 | | |- tw
4 | | | |- bicss
5 | | | | |- gwt
6 | | | | | |- client
7 | | | | | | |- MyApp.java
8 | | | | | | |- res
9 | | | | | | | |- MyAppConstants.java
10 | | | | | | | | |- MyAppConstants.properties
11 | | | | | | | | |- public
12 | | | | | | | | |- MyApp.html
13 | | | | | | | | |- MyApp.gwt.xml
14 |- test
15 | |- at
16 | | |- tw
17 | | | |- bicss
18 | | | | |- gwt
19 | | | | | |- client
20 | | | | | | |- MyAppTest.java

```

Die Java-Sourcen des Projekts befinden sich nach Listing 4.18 im Verzeichnis `src`, wobei folgende wichtige GWT-Komponenten zu nennen sind:

- `MyApp.html`: Hosted GWT-Website
- `MyApp.gwt.xml`: GWT-Module
- `MyApp.java`: GWT-Entry-Point

Die Quellen der generierten JUnit-Test-Klassen befinden sich im Verzeichnis test.

```

1 |- tomcat
2 | |- conf
3 | | |- gwt
4 | | | |- localhost
5 | | | |- web.xml
6 | | |- webapps
7 | | | |- ROOT
8 | | | | |- WEB-INF
9 | | | | | |- web.xml
10 | | |- work
11 | | | |- gwt
12 | | | | |- localhost
13 | | | | | |- _
14 | | | | | | |- SESSIONS.ser
15 | | | | | | |- tldCache.ser

```

#### **Listing 4.19**

*Verzeichnisstruktur  
einer GWT-Applikation*

3

Der eingebettete Apache Tomcat [WebTomcat] des GWT ist nach Listing 4.19 im Verzeichnis tomcat zu finden.

```

1 |- www
2 | |- at.tw.bicss.gwt.MyApp
3 | | |- 0A32B1D6BDA01809DEBF3D4CB229A167.cache.html
4 | | |- 0A32B1D6BDA01809DEBF3D4CB229A167.cache.xml
5 | | |- 361FE139D156FCB61903FB4115EF7AD2.cache.html
6 | | |- 361FE139D156FCB61903FB4115EF7AD2.cache.xml
7 | | |- 41882F9AD3ACFD4ADAD7F846FEFAC0BA.cache.html
8 | | |- 41882F9AD3ACFD4ADAD7F846FEFAC0BA.cache.xml
9 | | |- at.tw.bicss.gwt.MyApp.nocache.html
10 | | |- C126719C459D4D97E527FD96C01F3E73.cache.html
11 | | |- C126719C459D4D97E527FD96C01F3E73.cache.xml
12 | | |- gwt.js
13 | | | - history.html
14 | | | |- MyApp.html
15 | | | |- tree_closed.gif
16 | | | |- tree_open.gif
17 | | | |- tree_white.gif
18 |- .classpath
19 |- .project
20 |- MyApp-compile.cmd
21 |- MyApp-shell.cmd
22 |- MyApp.ant.xml
23 |- MyApp.launch
24 |- MyAppConstants-i18n.cmd

```

#### **Listing 4.20**

*Verzeichnisstruktur  
einer GWT-Applikation*

4

```

25 |- MyAppConstants-i18n.launch
26 |- MyAppTest-hosted.cmd
27 |- MyAppTest-hosted.launch
28 |- MyAppTest-web.cmd
29 |- MyAppTest-web.launch

```

In Listing 4.20 ist das wichtigste Verzeichnis, das Ausgabeverzeichnis des GWT-Compilers `www`, zu finden. Die in diesem Verzeichnis vorhandenen in JavaScript kompilierten Java-Sourcen des Frontends können nun für den Web Mode (siehe Abschnitt 4.4.2) verwendet werden. Folgende wichtige GWT-Dateien sind dabei zu nennen:

- `at.tw.bicss.gwt.MyApp.nocache.html`: Diese HTML-Datei wird vom Script `gwt.js` geladen, und für die Bestimmung des Browsers (HTTP-User-Agent) verwendet.
- `[class-id].cache.html`: Konkrete proprietäre Implementierung der GWT-Applikation für den jeweiligen Browser, die von der HTML-Datei `at.tw.bicss.gwt.MyApp.nocache.html` geladen wird. Die Class-IDs bedeuten dabei für folgende HTTP-User-Agents der Browser-Implementierungen:  
`0A32B1D6BDA01809DEBF3D4CB229A167`: gecko1\_8 oder gecko (Firefox und Mozilla)  
`361FE139D156FCB61903FB4115EF7AD2`: safari (Safari Webbrowser)  
`41882F9AD3ACFD4ADAD7F846FEFAC0BA`: opera (Opera Webbrowser)  
`C126719C459D4D97E527FD96C01F3E73`: ie6 (Internet Explorer)
- `[class-id].cache.xml`: Konfigurationsdatei für die jeweilige proprietäre Implementierung `[class-id].cache.html` des Browsers.

Der Application Creator erzeugt die in Tabelle 4-6 dargestellte obligatorische Package-Struktur einer GWT-Applikation `[myDomain].[appDomain].[client|public|server]`.

**Tab. 4-6**  
GWT-Packagestruktur

| Package           | Beschreibung                                                                                                                                       |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| Client            | Java-Code für das Frontend, der mit Hilfe des GWT-Compilers (siehe Abschnitt 4.3) in JavaScript übersetzt wird.                                    |
| public            | Beinhaltet die eigentliche Webseite der Anwendung (inkl. der für das Layout erforderlichen CSS-Dateien), die über den Browser geladen werden soll. |
| server (optional) | Java-Code für das Backend, der die eigentliche Programmlogik beinhalten soll. Dieses Package wird vom GWT-Compiler nicht berücksichtigt.           |

Im Folgenden wird auf die wichtigsten Komponenten von GWT, Hosted GWT-Website, GWT-Module und GWT-Entry-Point näher eingegangen.

### 4.5.1 Hosted-GWT-Website

Die Hosted GWT-Website ist die HTML-Seite, in der die gesamte GWT-Applikation eingebettet wird. Im Web Mode erfolgt die Einbettung des in JavaScript kompilierten Frontends über `<script language="javascript"src="gwt.js"></script>`. Im Hosted Mode ist die Verwendung der JavaScript-Datei nicht erforderlich, da ja direkt der Java-Bytecode ausgeführt wird.

Die GWT-History wird über das HTML-Tag `<iframe id="__gwt_historyFrame" style="width:0;height:0;border:0"></iframe>` realisiert und über das HTML-Tag `<meta name="gwt:module"content="at.tw.bicss.gwt.MyApp">` kann dann das dazugehörige GWT-Module gefunden werden. Das Listing 4.21 zeigt den Aufbau der Hosted-Website.

```

1 <html>
2 <head>
3 <!-- Any title is fine -->
4 <title>Wrapper HTML for MyApp</title>
5 <!-- Use normal html, such as style -->
6 <style>
7 body,td,a,div,.p{font-family:arial,sans-serif}
8 div,td{color:#000000}
9 a:link,.w,.w a:link{color:#0000cc}
10 a:visited{color:#551a8b}
11 a:active{color:#ff0000}
12 </style>
13 <!-- The module reference below is the link -->
14 <!-- between html and your Web Toolkit module -->
15 <meta name='gwt:module' content='at.tw.bicss.gwt.MyApp'>
16 </head>
17
18 <!-- The body can have arbitrary html, or -->
19 <!-- you can leave the body empty if you want -->
20 <!-- to create a completely dynamic ui -->
21 <body>
22 <!-- This script is required bootstrap stuff. -->
23 <!-- You can put it in the HEAD, but startup -->
24 <!-- is slightly faster if you include it here. -->
25 <script language="javascript" src="gwt.js"></script>
26 <!-- OPTIONAL: -->

```

**Listing 4.21**

Hosted-GWT-Website

```

27 <!-- include this if you want history support -->
28 <iframe id="__gwt_historyFrame"
29 style="width:0;height:0;border:0"></iframe>
30 <h1>MyApp</h1>
31 <table align=center>
32 <tr>
33 <td id="slot1"></td><td id="slot2"></td>
34 </tr>
35 </table>
36 </body>
37 </html>

```

Zusätzlich zum HTML-`<meta>`-Tag für das GWT-Module sind noch folgende `<meta>`-Tags möglich, die in Tabelle 4-7 dargestellt sind.

**Tab. 4-7**

Hosted-GWT-Website:  
Meta-Tags

| Meta-Tag              | Beschreibung                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| gwt:module            | Definition des GWT-Modules                                                                                                                                                                                                                                                                                                                                                                                                       |
| gwt:property          | Spezifiziert ein verzögert gebundenes Client-Property, das für viele Aspekte verwendet werden kann. Als Beispiel könnte die Spracheinstellung (Locale) einer Applikation geladen werden, die wiederum andere spezifische Konstantendateien für Nachrichten laden könnte. Mit Hilfe dieses Elements werden Schlüssel/Wert-Paare mit der Notation <code>&lt;meta name="gwt:property" content="_name=_value_"&gt;</code> definiert. |
| gwt:onPropertyErrorFn | Spezifiziert eine Funktion, die bei einem ungültigen Wert einer Client-Eigenschaft aufgerufen wird. Eine beispielhafte Definition könnte folgendermaßen aussehen: <code>&lt;meta name="gwt:onPropertyErrorFn" content="_fnName_"&gt;</code>                                                                                                                                                                                      |
| gwt-onLoadErrorFn     | Diese Funktion wird aufgerufen, sobald eine Exception bei der Entry-Point-Methode <code>onModuleLoad()</code> geworfen wird. Die Definition erfolgt mit: <code>&lt;meta name="gwt:onLoadErrorFn" content="_fnName_"&gt;</code> .                                                                                                                                                                                                 |

## 4.5.2 GWT-Module

Ein GWT-Module ist eine Sammlung von clientseitigem Applikations-Code und etwaigen dazugehörigen Ressourcen, die wie folgt definiert werden können.

Mithilfe des XML-Tags `<inherits name="moduleName"/>` können GWT-Kernmodule definiert werden, von denen das projektspezifische Modul ableitet. Jedoch müssen alle GWT-Module zumindest vom Modul `User` ableiten.

*Abgeleitete Module*

Die Entry-Point-Klasse kann über das Tag `<entry-point class="className"/>` definiert werden.

*Entry-Point*

Befindet sich der Quellpfad außerhalb des `Client-Packages`, so muss dieser mit `<source path="path"/>` angegeben werden.

*Source-Pfad*

Auch dieser Pfad kann mit `<public path="path"/>` erweitert werden, wenn sich der Public-Pfad außerhalb des `public-Package` befindet.

*Public-Pfad*

Das Tag `<servlet path="url-path" class=" className"/>` definiert die URL für das Java Servlet, das das Remote Service implementiert.

*RPC-URL*

Externes, eventuell selbst geschriebenes JavaScript kann mit Hilfe von `<script src=" js-url">script ready-function body</script>` inkludiert werden.

*Externes JavaScript*

Auch ein externes CSS kann mit `<stylesheet src=" css-url"/>` inkludiert werden.

*Externes CSS*

Das XML-Tag `<extend-property name=" client-property-name" values="comma-separated-values"/>` definiert einen Wertebereich des Client-Properties, das mit dem HTML-Tag `<meta name="gwt:property" content=" _name=_value_">` innerhalb der Hosted-Website definiert wird.

*Wertebereichsdefinition für Client-Property*

Das Listing 4.22 zeigt den Aufbau der XML-Konfigurationsdatei, der schon vorher erwähnten Datei `MyApp.gwt.xml`.

```

1 <module>
2 <!-- Inherit the core Web Toolkit stuff. -->
3 <inherits name="com.google.gwt.user.User"/>
4
5 <!-- Specify the app entry point class. -->
6 <entry-point class="at.tw.bicss.gwt.client.MyApp"/>
7 </module>

```

**Listing 4.22**  
*GWT-Module*

### 4.5.3 GWT-Entry-Point

Der GWT-Entry-Point stellt den eigentlichen Einsprungspunkt einer GWT-Applikation dar. Die Java-Klasse des GWT-Entry-Points muss dabei das Java-Interface `EntryPoint` implementieren und die Methode `onModuleLoad()` überschreiben (siehe Listing 4.23). Innerhalb dieser Methode kann dann der projektspezifische Client-Code codiert werden. Nach Abschnitt 4.4.2 wird der Entry-Point mit Hilfe des GWT-Compilers in JavaScript übersetzt.

**Listing 4.23**  
GWT-Entry-Point

```
1 package at.tw.bicss.gwt.client;
2
3 import com.google.gwt.core.client.EntryPoint;
4
5 /**
6 * Entry point classes define onModuleLoad().
7 */
8 public class MyApp implements EntryPoint {
9
10 /**
11 * This is the entry point method.
12 */
13 public void onModuleLoad() {
14 // Client Application Stuff ...
15 }
16 }
```