3.– 6. September 2012
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Der vergessene Code

JavaScript in Java-Projekten

Stefan Hildebrandt

consulting.hildebrandt.tk

@hildebrandttk

Agenda

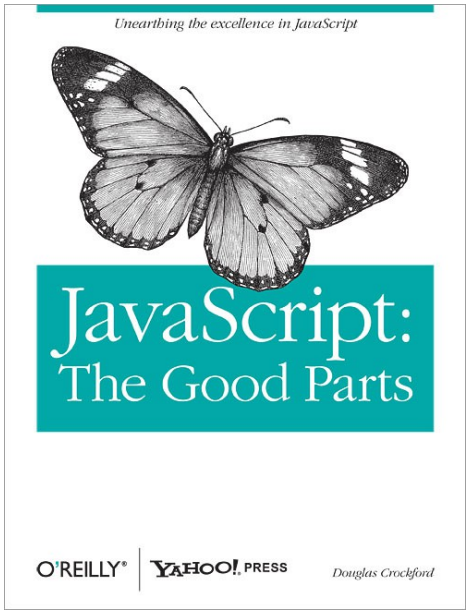
- Motivation
- JSLint
- Depenenmanagement
- Testausführung
- Zusammenfassen und Komprimieren
- Einbinden in Webseiten
- Entwicklungsvorgehen
- Und bei Gradle?

Motivation

- JavaScript wird immer häufiger und umfangreicher in Java-Webprojekten eingesetzt
- Java und Javascript haben nicht viel mehr als einen Teil des Namens gemeinsam
  → Java-Entwickler beherrschen es nicht immer richtig.
- Das Tooling für JavaScript-Code ist nicht auf dem Nivea von Java-Code
- In der Regel gibt es keine Architekturvorgaben, die den Einsatz steuern

Schwächen der Sprache

- JavaScript The Good Parts von Douglas Crockford
  - Diese Good Parts stellen einen stabilen Grundstock für professionelle Entwicklung dar
  - Enthält eine Übersicht der „Awful Parts“ und der „Bad Parts“
  - Zur Überprüfung hat er jslint definiert



Schwächen der Sprache (Beispiele)

- Globale Variablen
  - Implizite Deklaration
- Scopes von lokalen Variablen
  - Gilt von Begin der Methode
- Semikolon Einsetzung
  - Es wird versucht diese hinzuzufügen, wenn diese nicht vorhanden waren
- typeof
  - typeof null → object !?
- parseInt
  - Arbeitet stillschweigend bis zum 1. nicht Ziffer
  - Arbeitet mit führender „0“ im oktalen System

Schwächen der Sprache (Beispiele)

- Vergleiche

  " == '0' //false

  0 == " // true

  0 == '0' //true

  false == 'false' //false

  false == '0' //true

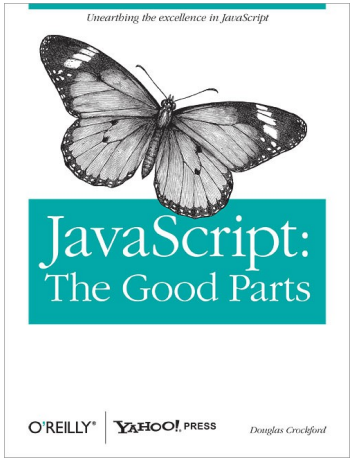
  false == undefined //false

  false == null //flase

  null == undefined // true

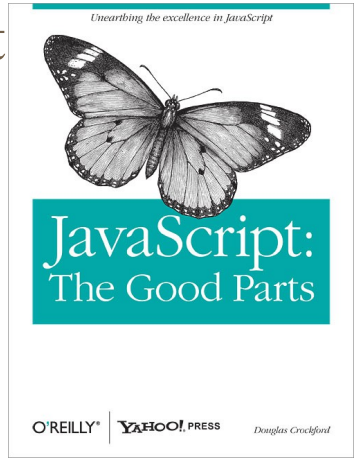
  ' \t\r\n' == 0 //true

  → Nur „===“ und „!==“ verwenden



Schwächen der Sprache (Beispiele)

- UTF Unterstützung auf 16bit beschränkt
- NaN
  - typeof NaN === 'number'
  - NaN === NaN // → false
  - NaN !== NaN // → true
  - isNaN (NaN) // → true



JSLint aus Maven

```
<encoding>utf-8</encoding>
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jslint-maven-plugin</artifactId>
  <version>1.0.1</version>
  <executions>
    <execution>
      <goals>
        <goal>jslint</goal>
        <goal>test-jslint</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Beispiel JSLint

- Code Beispiel

Beispiel JSLint

- Deklaration von Globals notwendig, damit diese keine Fehler erzeugen

```
/*global QUnit, checkHorizontalChecksum */
```

Dependency Management: AMDJS

- AMDJS
  - Identifiziert Bibliotheken an Globalen Variablen
  - Verwendet die gleiche Signatur wie JLLint
  - Dadurch besteht Konflicktpotential

- Beispiel

```
/*global define, window */
```

Dependency Management - Maven

- Depenency-Management für JS-Artifakte möglich
- js-import-maven-plugin
- Bindet alle js-Depenencies und lokale Dateien in AMDJS ein
- Bietet zusätzlich einen Mechanismus um Dateien ohne globale Objekte anzuziehen

```
/**
 * @import com.jqueryui:jquery-ui
 */
```

Dependency Management - Maven

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>js-import-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>import-js</goal>
        <goal>test-import-js</goal>
        <goal>generate-html</goal>
        <goal>test-generate-html</goal>
      </goals>
    </execution>
  </executions>
```

Dependency Management - Maven

```
  <dependencies>
    <dependency>
      <groupId>com.github.jrburke</groupId>
      <artifactId>almond</artifactId>
      <version>0.1.3</version>
      <type>js</type>
    </dependency>
  </dependencies>
</plugin>
```

Dependency Management - Maven

- Code Beispiel

Testausführung

- JsTestDriver
  - Setzt auf installierte Browser wie Firefox, IE und Chrome
  - Es werden alle Bibliotheken geladen, die für den Test definiert wurden
    - Doppelte Konfiguration
  - jstd-maven-plugin

Testausführung

- JsTestrunner
  - Setzt auf PhantomJS
    - Headless-Browser auf Basis von WebKit
    - JS-Api zum Steuern
  - Wird mittels surefire-plugin und einer Wrapper-Klasse in maven eingebunden
  - Lässt sich auch aus der IDE starten
    - In Kombination mit anderen Schritten aufwendig zu konfigurieren

Testausführung

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <systemPropertyVariables>
      <org.codehaus.jstestrunner.commandPattern>$
        {org.codehaus.jstestrunner.commandPattern}
      </org.codehaus.jstestrunner.commandPattern>
    </systemPropertyVariables>
    <includes>
      <include>**/JsTestRunnerAdapter.class</include>
    </includes>
  </configuration>
</plugin>
```

Testausführung

```
package tk.hildebrandt.consulting.hc2012;


import org.codehaus.jstestrunner.junit.JSTestSuiteRunner;
import org.junit.runner.RunWith;


@RunWith(JSTestSuiteRunner.class)
public class JsTestRunnerAdapter {
}
```

Testausführung

- Beispiel JsTestrunner

Kompression und Zusammenfassung

- Motivation
  - Deutliche Verbesserung der Ladezeiten
  - Bessere Ausführungszeiten, da wendiger Code interpretiert werden muss

Kompression und Zusammenfassung

- Motivation
  - Deutliche Verbesserung der Ladezeiten
  - Bessere Ausführungszeiten, da wendiger Code interpretiert werden muss
- `webminifier-maven-plugin`
  - Bearbeitet alle Bibliotheken, die in produktiven HTML-Seiten eingebunden sind
  - Modifiziert alle HTML-Seiten

Kompression und Zusammenfassung

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>webminifier-maven-plugin</artifactId>
    <version>1.0.1</version>
    <executions>
        <execution>
            <goals>
                <goal>minify-js</goal>
            </goals>
        </execution>
    </executions>
```

Kompression und Zusammenfassung

```
            <configuration>
                <jsSplitPoints>
                    <property>
                        <name>js/my-js-lib1.js</name>
                        <value>my-js-lib1</value>
                    </property>
                </jsSplitPoints>
            </configuration>
        </plugin>
```

Kompression und Zusammenfassung

```
            <configuration>
                <jsSplitPoints>
                    <property>
                        <name>js/my-js-lib1.js</name>
                        <value>my-js-lib1</value>
                    </property>
                </jsSplitPoints>
            </configuration>
        </plugin>
```

Entwicklungsvorgehen

- Einfaches Neuladen der html-Dateien ist nicht mehr möglich
- Auslieferung der modifizierten Dateien mittels Webserver notwendig
- Umkonfiguration des jetty-Plugins

Entwicklungsvorgehen - jetty-plugin

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <configuration>
   <webAppConfig>
     <contextPath>/</contextPath>
     <resourceBases>
       <resourceBase>${project.build.outputDirectory}</resourceBase>
       <resourceBase>${project.build.testOutputDirectory}</resourceBase>
     </resourceBases>
    </webAppConfig>
   </configuration>
</plugin>
```

Und bei Gradle?

- Lint, Minify und Jetty Plugins sind vorhanden
- Testausführung und Depenency-Management warten auf bessere JS-Unterstützung in Gradle

Fazit

- Mit dem Maven-Vorgehen wird die Entwicklung schwergewichtiger
- Bei Gradle ist die JS-Entwicklung am Anfang
- Alternative Ansätze auf JS-Basis sollten ggf. als Übergangstechnik genutzt werden
  - z.B. stealJS aus dem javascriptmvc-Umfeld

Noch Fragen?

Vielen Dank!

Stefan Hildebrandt

consulting.hildebrandt.tk

@hildebrandttk

consulting.hildebrandt.tk

- Beratung, Coaching und Projektunterstützung
  - Java EE
    - CDI, EJB3.1, JSF2, JPA
    - Spring, Hibernate
    - Tomcat, JBoss, TomEE
    - Maven, Gradle
    - Troubleshooting (Performance, Speicher)
  - Singepage Applications mit REST und JavaScript
  - Testautomatisierung
  - Testen in agilen Projekten

consulting.hildebrandt.tk
@hildebrandttk