01-2017 | Frühling | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

Java ist die beste Wahl

Einfacher programmieren
Erste Schritte mit Kotlin

Security
Automatisierte Überprüfung von Sicherheitslücken

Leichter testen
Last- und Performance-Test verteilter Systeme

4 191978 304903 01

iJUG Verbund

Der Unterschied von Java EE zu anderen Enterprise Frameworks

Kotlin ist eine ausdrucksstarke Programmiersprache, um die Lesbarkeit in den Vordergrund zu stellen und möglichst wenig Tipparbeit erledigen zu müssen

Innerhalb von Enterprise-Anwendungen spielen Sicherheits-Aspekte eine wichtige Rolle

Automatisiertes Testen in Zeiten von Microservices

Christoph Deppisch und Tobias Schneck, ConSol Software GmbH

Die Software-Entwicklung ist im Wandel. Immer schneller, immer häufiger, immer einfacher müssen neue Features in Produktion gebracht werden. Große, schwergewichtige Alleskönner werden durch mehrere kleine, individuelle Services ersetzt. Jeder Microservice bildet einen Aspekt der gesamten Fachlichkeit ab und lässt sich deshalb unabhängig entwickeln und warten. Welche Auswirkungen hat diese veränderte Sicht einer Software-Architektur auf die Qualitätssicherung in der Entwicklung? Ist hier auch alles einfacher, schneller und besser? Die Antwort lautet: „Ja und nein". Während die dezentralisierten Services Vorteile für die Testbarkeit und mehr Flexibilität versprechen, ist die Integration der Services eine Herausforderung im automatisierten Test.

„Eine Software muss ausreichend getestet werden." Dieser Grundsatz ist so alt wie das erste „Hello World" – aber in Zeiten von Microservices und Continuous Delivery treffender denn je. Software muss immer schneller den Weg in den Markt finden. Änderungen, Bugfixes und Features sollen im kontinuierlichen Rhythmus in möglichst kurzen Abständen in Releases gepackt und dem Nutzer zur Verfügung gestellt werden. Dieses Vorgehen hat auch unweigerlich Auswirkungen auf die Qualitätssicherung in der Software-Entwicklung. Man kann es sich nicht mehr leisten, vor einem Release aufwändige Tests zu fahren, um dann kurz vor der Deadline diverse Bugs zurück an die Entwicklung zu melden, in der diese dann umständlich in weiteren Testzyklen verifiziert werden müssen.

Die Qualitätssicherung muss voll automatisiert und kontinuierlich schon während der Entwicklung stattfinden. „Continuous Integration" nennt sich dieser Ansatz, in dem alle Tests vollständig automatisiert ablaufen und fest in den Build-Lifecycle einer Anwendung integriert sind. Mit jeder Code-Änderung werden alle Tests ausgeführt und geben bei eventuell aufgetretenen Fehlern schnelles Feedback an die Entwickler.

Im Bereich der Unit-Tests ist diese Automatisierung bereits als Standard etabliert. Die Entwickler erstellen die Unit-Tests entwicklungsbegleitend und integrieren sie fest in den Build-Lifecycle mit Tools wie Maven *[1]*, Gradle *[2]* und Jenkins *[3]*.

Testbedarf auf mehreren Ebenen

Unit-Tests sind leider nicht in der Lage, alle Aspekte einer Software ausreichend unter Test zu stellen. Sie durchlaufen, wie der Name (Unit = Einheit) schon sagt, kleine Code-Abschnitte, also eine Klasse oder Methode. Dies findet in kompletter Isolation zu anderen Einheiten statt. Alle Abhängigkeiten zu Klassen, Komponenten oder Ressourcen werden im Test durch Mocks ersetzt. Diese Isolation hat viele Vorteile. Die Unit-Tests sind sehr schnell und ohne großen Aufwand wiederholt ausführbar. Eine direkte Folge dieser Isolation ist aber auch, dass wir die Interaktion und Integration mit anderen Komponenten nicht getestet haben. Wichtige Aspekte wie User Interfaces, Konfiguration, Dependency Injection, Ressourcen-Verwaltung und Schnittstellen nach außen bleiben damit ungetestet.

Vor allem die zuletzt genannten Schnittstellen bekommen im Sinne von Microservices eine immer größere Bedeutung. Die individuellen Services tauschen untereinander Daten über Schnittstellen wie HTTP REST oder JMS aus. Die Schnittstellen folgen zwar gewissen Regeln, befinden sich aber auch stetig im fachlichen Umbau. Die Zusammenarbeit mehrerer Services und die reibungslose Integration der Services untereinander sind daher enorm wichtige Aspekte für die Qualitätssicherung.

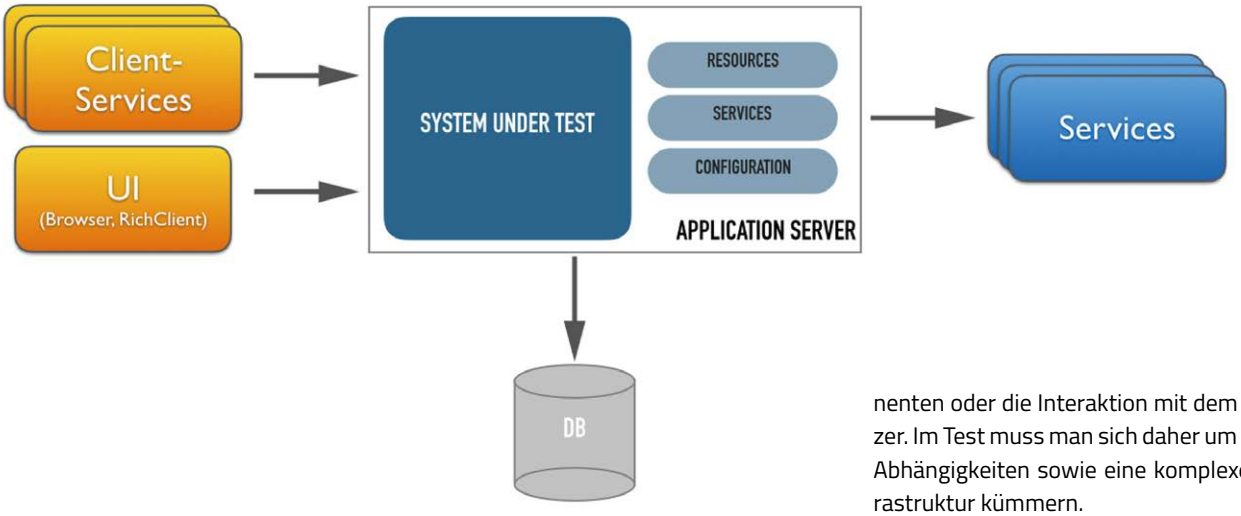

Abbildung 1: Beispiel-Infrastruktur im Integrations- beziehungsweise UI-Test

Wenn alle Units für sich allein erfolgreich getestet wurden, heißt das noch lange nicht, dass auch die Zusammenarbeit aller Units reibungslos funktioniert. Hier helfen nur die Integrations- beziehungsweise Acceptance-Tests, die genau das Zusammenspiel mehrerer Komponenten als primäres Testziel verfolgen. Hinzu kommen die UI-Tests, die sich mit der Benutzeroberfläche einer Software beschäftigen und die richtige Anzeige der Daten testen. Auch hier ist ein funktionsfähiges Backend mit entsprechenden Daten in der Datenhaltung Voraussetzung für den sinnvollen Test. Natürlich müssen diese Tests im Sinne des Continuous Delivery auch vollautomatisiert werden; dies bringt einige Herausforderungen mit sich.

Herausforderungen bei der Automatisierung

Integrations- beziehungsweise UI-Tests haben einen entscheidenden Nachteil. Die in den Unit-Tests praktizierte Isolation ist hier leider nicht mehr gegeben. Ziel der Tests ist ja genau die Integration mehrerer Komponenten oder die Interaktion mit dem Benutzer. Im Test muss man sich daher um diverse Abhängigkeiten sowie eine komplexere Infrastruktur kümmern.

Abbildung 1 zeigt eine solche Infrastruktur als Beispiel. In der Regel wird eine zu testende Software-Komponente (System-Under-Test, kurz „SUT“) in einem Application Server oder einer ähnlichen Laufzeitumgebung eingerichtet und konfiguriert. Hinzu kommen diverse Abhängigkeiten zu anderen Services und der Datenhaltung (zum Beispiel zur Datenbank). Mehrere Clients konsumieren im Test nun die vom SUT angebotenen Services. Das SUT greift wiederum auf die Persistenz in der Datenbank oder auf weitere Services zu.

Diese Infrastruktur ist in einem automatisierten Test vollständig aufzubauen. Clients und Backend-Services müssen im Test entsprechend simuliert werden. UI-Tests betätigen die Benutzeroberflächen und stellen sicher, dass die gelieferten Daten entsprechend angezeigt sind. Die Test-Durchführung ist hier weitaus komplexer als in den Unit-Tests. Trotzdem müssen alle Tests automatisiert und vor allem effektiv wiederholbar sein, um die geforderten Release-Zyklen im Continuous Delivery einhalten zu können.

```
public class Greeter {
  public void sayHello(PrintStream to, String user) {
     to.println(sayHello(user));
  }

  public String sayHello(String user) {
     return "Hello, " + user + "!";
  }
}

@RunWith(Arquillian.class)
public class SayHelloTest {

  @Deployment
  public static JavaArchive createDeployment() {
     return ShrinkWrap.create(JavaArchive.class)
        .addClass(Greeter.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  }

  @Inject
  Greeter greeter;

  @Test
  public void should_say_hello() {
     Assert.assertEquals("Hello, Yoda!",
        greeter.sayHello("Yoda"));
     greeter.sayHello(System.out, "Yoda");
  }
}
```

Listing 1

Tools für den Integrationstest

Zum Glück gibt es eine Reihe von Tools und Frameworks, die bei der Umsetzung dieser Aufgaben unterstützen. Arquillian *[4]* ist ein Open-Source-Tool, um eine Java-Anwendung in einem Container automatisiert zu testen. Das Framework bietet dafür ein hervorragendes Container-Lifecycle-Management sowohl für gängige Java Application Server (WildFly, GlassFish, Tomcat, JBoss AS etc.) als auch für Docker-Container-Umgebungen. Arquillian kümmert sich im Vorfeld eines Tests um den Start des Containers und das automatische Deployment der zu testenden Java-Archive (JAR, WAR oder EAR). Nach dem Deployment stehen der Java-Anwendung während des Tests diverse Container-Res-

sourcen wie JNDI, JMS, CDI Injection und EJB-Ressourcen zur Verfügung *(siehe Listing 1)*.

Ähnlich nützlich für den automatisierten Infrastruktur-Aufbau ist das Docker-Maven-Plug-in von fabric8 *[5]*. Damit lassen sich Docker-Images im Maven-Build-Lifecycle bauen und im Docker-Container starten. Somit können beispielsweise Datenbanken und Komponenten wie ein JMS Message Broker vollautomatisch vor dem Test gestartet werden.

Ebenfalls möglich ist das automatische Deployment der SUT-Komponente in einem Application Server. Damit wird die komplette Infrastruktur für den Test automatisch im Maven Build initialisiert. Nach dem Testdurchlauf können die Komponenten entsprechend automatisch wieder gestoppt werden, um eine saubere Test-Umgebung zu hinterlassen.

Das Docker „fabric8 maven"-Plug-in in *Listing 2* zeigt die Konfiguration für eine MySQL-Datenbank sowie für einen AqtiveMQ Message Broker, die jeweils in einem Docker-Container ausgeführt werden. Beide Container können über das Plug-in im Maven-Lifecycle automatisch gestartet und gestoppt werden. Dabei greift das Beispiel auf fertige Docker-Images der einzelnen Komponenten zurück. Damit lassen sich die für den Test benötigten Komponenten einfach und bequem starten. Der Maven-Build-Prozess wartet sogar, bis die Container erfolgreich hochgefahren wurden, bevor die Testfälle gestartet werden.

Nun fehlen noch diverse Clients und Backend-Services, die während des Tests Daten abfragen beziehungsweise Daten liefern. Dafür bietet sich ein weiteres Java-Open-Source-Framework mit dem Namen „Citrus" *[6]* an. Damit ist man in der Lage, nachrichtenbasierte Schnittstellen client- und serverseitig im Test zu simulieren. Citrus stellt dafür als Framework fertige Komponenten für das Senden und Empfangen von Nachrichten zur Verfügung. Diese sogenannten „Endpoints" gibt es für HTTP REST, JMS, SOAP, FTP, RMI, XML, JSON und viele weitere Transportwege und Datenformate. Citrus bietet eine Java-DSL, um die Testlogik für den Austausch von Nachrichten über diverse Schnittstellen zu definieren. Die Anweisungen sind einfach in einen Arquillian- oder JUnit-Test integrierbar *(siehe Listing 3)*.

Das Citrus-Beispiel zeigt zunächst die Konfiguration der Citrus-Endpoints für den Test. Im Test werden diese entsprechend referenziert. Der Test umfasst eine einfache HTTP-REST-Request-Response-Kommunikation, wobei Citrus als Client das SUT über die Schnittstelle aufruft und eine entsprechende HTTP-200-Ok-Response empfangen möchte. Als zweiten Schritt erwartet Citrus eine eingehende Mail-Nachricht, die auf dem SUT ausgelöst wurde. Im Test werden die realen Schnittstellen des SUT bedient und somit auf die korrekte Funktionsweise überprüft. Citrus überprüft alle eingehenden Nachrichten gegen ein erwartetes Template, das sowohl Body- als auch Header-Informationen beinhalten kann.

Tools für den UI-Test

Im Bereich der UI-Tests ist das wohl bekannteste Web-Testing-Framework Selenium *[7]*. Es ist in der Lage, Interaktionen eines Benutzers im Browser zu simulieren. Dabei werden Document-Object-Model-Elemente (DOM) wie Buttons, Textfelder oder Links anhand von Namen, IDs oder CSS-Informationen identifiziert und angesteuert. Selenium-Tests lassen sich dabei über JUnit sehr gut in den etablierten Build-Lifecycle integrieren und automatisch ausführen.

Sollen neben rein Web-basierten Inhalten auch Rich-Clients wie PDF Reader oder native Anwendungen wie ein Mail-Client Gegenstand eines automatisierten Oberflächentests sein, bietet sich der Einsatz eines Frameworks wie Sakuli *[8]* an. Es kann neben den DOM-basierten Elementen auch Inhalte außerhalb

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <configuration>
     <logDate>default</logDate>
     <verbose>true</verbose>
     <images>
        <image>
           <alias>mysqldb</alias>
           <name>mysql:5.5</name>
           <run>
              <namingStrategy>alias</namingStrategy>
              <ports>
                 <port>3306:3306</port>
              </ports>
              <env>
                 <MYSQL_ROOT_PASSWORD>admin</MYSQL_ROOT_PASSWORD>
                 <MYSQL_DATABASE>todolist</MYSQL_DATABASE>
                 <MYSQL_USER>todo</MYSQL_USER>
                 <MYSQL_PASSWORD>secret</MYSQL_PASSWORD>
              </env>
              <wait>
                 <log>MySQL init process done</log>
                 <time>60000</time>
              </wait>
              <log>
                 <enabled>true</enabled>
                 <color>green</color>
              </log>
           </run>
        </image>
        <image>
           <alias>activemq-broker</alias>
           <name>consol/activemq-5.12:latest</name>
           <run>
              <namingStrategy>alias</namingStrategy>
              <ports>
                 <port>61616:61616</port>
                 <port>8161:8161</port>
              </ports>
              <wait>
                 <http>
                    <url>http://localhost:8161</url>
                    <method>GET</method>
                    <status>200</status>
                 </http>
                 <time>60000</time>
              </wait>
              <log>
                 <enabled>true</enabled>
                 <color>green</color>
              </log>
           </run>
        </image>
     </images>
  </configuration>
</plugin>
```

Listing 2

```
<citrus-http:client id="reportingClient"
  request-method="GET"
  request-url="http://localhost:8080/report/services"/>

<citrus-mail:server id="mailServer"
  port="${mail.server.port}"
  auto-accept="true"
  auto-start="true"/>

@Test
public class ReportingIT extends TestNGCitrusTestDesigner {

  @Autowired
  private HttpClient reportingClient;

  @Autowired
  private MailServer mailServer;

  @CitrusTest
  public void shouldSendMailForLargeOrder() {
     echo("Add 1000+ order and receive mail");

     variable("orderType", "chocolate");
     variable("amount", "1001");

     http().client(reportingClient)
        .send()
        .put("/reporting")
        .queryParam("id", "citrus:randomNumber(10)")
        .queryParam("name", "${orderType}")
        .queryParam("amount", "${amount}");

     http().client(reportingClient)
        .receive()
        .response(HttpStatus.OK);

     echo("Receive report mail for 1000+ order");

     receive(mailServer)
        .payload(new ClassPathResource("templates/mail.xml"))
        .header(CitrusMailMessageHeaders.MAIL_SUBJECT, "Congratulations!");

     send(mailServer)
        .payload(new ClassPathResource("templates/mail_response.xml"));
  }
}
```

Listing 3

des Browser-Fensters ansteuern und im Test verifizieren. Je nach Bedarf macht sich Sakuli hierbei die Bildmuster-Erkennung zunutze und ist damit in der Lage, alle nativen Rich-Client-Anwendungen über User-Aktionen per Maus oder Tastatur auf der nativen UI des Betriebssystems zu bedienen *(siehe Listing 4)*.

Für die Ausführung der Tests ohne Anzeige („headless“) bietet Sakuli fertige Docker-Images mit vorinstalliertem Chrome und Firefox, die dann sogar die Ausführung der Tests in parallel laufenden Containern ermöglichen. So können die oftmals lang laufenden UI-Tests im Parallelbetrieb deutlich schneller ausgeführt werden. Ein komplettes Beispiel mit lauffähigem Continuous Integration Build ist auf GitHub *[9]* verfügbar.

Wartung und Pflege

Die Test-Automatisierung im Unternehmen nimmt ihren Lauf und immer mehr automatisierte Tests werden erstellt und kontinuierlich mit jeder Änderung ausgeführt. Daraus ergibt sich ein gewisser Pflegeaufwand für die Tests, der nicht zu unterschätzen ist.

Der entwickelte Code wird ständigen Änderungen und Anpassungen unterzogen. Features und fachliche Abläufe, die gestern noch aktuell waren, werden heute schon anders interpretiert und implementiert. Die Pflege der Tests muss daher entwicklungsbegleitend stattfinden. Wenn ein Entwickler eine Code-Änderung durchführt, muss auch die Testlogik bei Bedarf entsprechend nachgezogen werden. Im Idealfall kann der Entwickler alle Tests problemlos lokal bei sich auf dem Rechner ausführen, um eventuelle Testfehler zu analysieren und gegebenenfalls zu beheben, bevor der Check-in Fehler in der Continuous-Integration-Build-Pipeline auslöst.

Es ist jedoch kaum zu vermeiden, dass Fehler im Continuous Build auftreten – ausgelöst von fehlgeschlagenen Tests. In diesem Fall empfiehlt es sich, zeitnah Feedback an die Entwickler zu geben, damit diese die Test-Ergebnisse leichter den letzten Code-Änderungen zuordnen können. Dadurch lassen sich die Fehler viel besser analysieren und nachvollziehen.

Ein roter Continuous Build, der über Tage hinweg keine Beachtung findet, führt zu einer generellen Müdigkeit und Gleichgültigkeit gegenüber dem Build-Status. Wenn sich verschiedene Probleme und Fehler im Build dann auch noch überlagern, ist eine Fehler-Analyse und -Behebung deutlich erschwert. Bei hoher Test-Automatisierung ist eine kontinuierliche Anpassung und Fehlerbehebung als zwingend notwendig anzusehen. Dieser Grundsatz muss im Team etabliert werden und in den täglichen Arbeitsrhythmus fest integriert sein.

Fazit

Die vollständige Automatisierung aller Tests und die kontinuierliche Ausführung im Continuous Integration Build sind essenzielle Voraussetzungen für schnelle und häufige Lieferungen einer Software. Nur so kann die Entwicklung jederzeit die Release-Fähigkeit einer Anwendung beurteilen. Mit den vorgestellten Tools ist eine Automatisierung auch in

```
//open print preview
env.type("p", Key.CTRL);
screen.find("save_button").highlight().click().type(Key.ENTER);

//open pdf in new tab and validate
env.type("tl", Key.CTRL).paste(getPDFpath()).type(Key.ENTER);
screen.waitForImage("pdf_place_order.png", 5).highlight();
[
  "pdf_blueberry.png",
  "pdf_caramel.png",
  "pdf_chocolate.png"
].forEach(function (imgPattern) {
  screen.find(imgPattern).highlight();
});
```

Listing 4

den Bereichen der Integrations-, Acceptance- und UI-Tests möglich. Damit hat man nicht nur eine gute Absicherung gegen Bugs, sondern auch Sicherheit bei zukünftigen Anpassungen.

Mit zunehmender Automatisierung wächst jedoch auch die Verantwortung für alle Beteiligten im Hinblick auf die Pflege der Tests. Fehlgeschlagene Tests dürfen nicht zu lange aufgeschoben werden, damit ein roter Build nicht zur akzeptierten Gewohnheit wird.

Die Art und Weise, wie wir Software entwickeln, befindet sich im Wandel – stets mit dem Ziel, Flexibilität und Effizienz zu optimieren. Da ist es nur logisch, dass die Qualitätssicherung denselben Mechanismen unterzogen wird und die gleiche Beachtung im Unternehmen finden muss.

Weiterführende Weblinks

[1] *https://maven.apache.org*
[2] *https://gradle.org*
[3] *https://jenkins.io*
[4] *http://arquillian.org*
[5] *https://github.com/fabric8io/fabric8-maven-plugin*
[6] *http://www.citrusframework.org*
[7] *http://www.seleniumhq.org*
[8] *https://github.com/ConSol/sakuli*
[9] *https://github.com/toschneck/sakuli-example-bakery-testing*

Christoph Deppisch
christoph.deppisch@consol.de

Christoph Deppisch arbeitet als Consultant und Software-Architekt bei der ConSol Software GmbH und verfügt über mehr als zehn Jahre Erfahrung bei der Umsetzung großer Enterprise-Projekte vor allem im Middleware-Integration-Umfeld. Als aktiver Open-Source-Entwickler ist er für das Test-Framework „Citrus" verantwortlich. In letzter Zeit beschäftigt er sich vor allem damit, welche Einflüsse Microservices und Container-Technologien auf die Continuous-Delivery-Pipeline haben.

Tobias Schneck
tobias.schneck@consol.de

Tobias Schneck ist seit zehn Jahren in der IT-Branche tätig. Er sammelte weitreichende Erfahrungen in den Bereichen der IT-Administration bis hin zur Entwicklung von IT-Speziallösungen. In seiner derzeitigen Position als Software-Consultant bei der ConSol Software GmbH ist er Mitbegründer des Open-Source-Testing-Frameworks „Sakuli" und spezialisierte sich als Java-Entwickler auf den Bereich „Test-Automatisierung". Er ist Konferenz-Speaker sowie Organisator der MeetUp-Gruppe „Agile Testing @Munich" und interessiert sich für neue, innovative Technologien.

Continuous Delivery of Continuous Delivery

Gerd Aschemann, Freiberufler

Zahlreiche Organisationen nutzen für ihre Software-Entwicklung bereits Continuous Integration oder sogar Continuous Delivery zum Fertigen, Prüfen und Ausliefern ihrer Software-Artefakte. Dazu betreiben sie eine Plattform, die selbst aus Software-Komponenten und zahlreichen Anpassungen für den spezifischen Lieferprozess der Organisation besteht. Konsequentes Handeln bedeutet hier, auch die Plattform selbst kontinuierlich liefern zu können, also Continuous Delivery of Continuous Delivery zu betreiben. Der Artikel zeigt, wie dieser Prozess via „Platform as Code" von einer DevOps-Organisation umgesetzt werden kann.

Wer kennt das nicht? Bob konnte den Build-Prozess nicht beenden. Kurz vor Abschluss des Sprints ist plötzlich alles „rot", Builds sind nicht durchgelaufen, Tests gescheitert, Artefakte nicht im Repository gelandet, Instanzen wurden nicht neu gestartet. Im morgendlichen Daily schauen alle betreten zu Boden. Fingerpointing ist verpönt und erstmal hat auch keiner eine Idee, wen er überhaupt anklagen könnte.

Bob selbst sagt nichts. Bob ist das Maskottchen des Teams. Er wird von allen liebe-