

# BlazeDS: Spring trifft Flex

(Link zum Artikel: <http://www.it-republik.de/jaxenter/artikel/2329>)

## Flex-Clients an Spring-Backends anbinden

*Text: Dirk Eismann*

Mit Flex 3 stellt Adobe ein leistungsfähiges Open-Source-Framework zur Erstellung von Rich-Internet-Anwendungen (RIAs) zur Verfügung. Mit dem neuen Spring-BlazeDS-Integration-Projekt ist die Anbindung von Flex-Clients an Spring Backends schnell und einfach realisiert.

Wer sich ernsthaft mit RIAs beschäftigt, kommt derzeit um Flex nicht herum: Das Flex 3 SDK von Adobe ist nach wie vor der Platzhirsch und ist auch den Konkurrenten Silverlight aus dem Hause Microsoft und JavaFX von Sun um einiges voraus. Insbesondere das Programmiermodell von Flex ist für Java-Entwickler attraktiv und mit BlazeDS stellt Adobe eine Open-Source-Java-EE-Messaging-Lösung bereit, um Flex-Clients an Java-EE-Backends anzubinden. Das fehlende Glied war bisher eine "runde" Anbindung an Spring. Mit dem neu erschienenen Spring-BlazeDS-Integration-Projekt wurde diese Lücke nun geschlossen.

Seit der Veröffentlichung des [Flex 3 SDKs](#) als quelloffene Software im Februar 2008 erfreut sich das Flex-Framework zunehmender Beliebtheit, wenn es um die Realisierung von Rich-Internet-Anwendungen geht. Wohlwollend hat die anfangs skeptische Community zur Kenntnis genommen, dass Adobe es mit seiner Open-Source-Strategie offensichtlich ernst meint. Mittlerweile ist die Entwicklergemeinde in den einschlägigen Open Source Communities wie Sourceforge und Google Code fleißig damit beschäftigt, das SDK zu erweitern oder neue Integrationsmöglichkeiten mit anderen Technologien zu entwickeln. Eine der interessantesten Neuigkeiten in dieser Hinsicht war die im Dezember 2008 veröffentlichte Ankündigung von Springsource über den ersten Meilenstein des [Spring-BlazeDS-Integration-Projekts](#), das eine einfache Anbindung von Flex-Clients an Spring verspricht. Zunächst aber ein kurzer Blick auf Flex und BlazeDS.

## Flex

Flex (oder genau genommen das Flex SDK) stellt ein Programmiermodell und verschiedene Tools bereit, um RIAs zu erstellen und zu kompilieren. Das Modell besteht dabei aus einem deklarativen XML-Namespace (MXML) und der objektorientierten, strikt typisierten Programmiersprache ActionScript 3, die auf dem ECMAScript-262-Edition-4-Entwurf basiert. Während mit MXML das eher statische Layout einer Flex-Anwendung beschrieben wird, codiert man clientseitige Logik wie Validierung oder komplexe Datenfilterung in ActionScript 3. Neben den typischen UI-Komponenten wie Button-, TextInput-, oder List-Controls stellt das Flex-Framework auch eine Vielzahl an nicht visuellen Klassen wie clientseitigen Proxies für den Aufruf von entfernten Web Services oder XML-RPC-Endpunkten zur Verfügung.

Mit dem Kommandozeilen-Compiler *mxmhc* schließlich wird der Quellcode in eine SWF-Datei kompiliert und kann dann auf einem Webserver deployt werden. Wer übrigens vor der Kommandozeile zurückschreckt und die Annehmlichkeiten einer IDE bei der Entwicklung nicht missen möchte, kann auf den Eclipse-basierten Flex Builder 3 zurückgreifen, der eine nahtlose Integration mit dem SDK und dem Framework bietet.

Um die Flex-Anwendung ausführen zu können, wird als Laufzeitumgebung ein Flash Player benötigt, der mindestens in der Version 9 auf dem Clientrechner installiert sein muss. Die SWF-Datei wird dann in der Regel durch den Browser auf den Rechner heruntergeladen und über das Flash Player Plug-in (bzw. ActiveX Control im IE) ausgeführt. Ein JIT-Compiler sorgt für das Extra an Performance auf dem Zielsystem. Die aktuelle Version 10 des Flash Players ist für Windows, Mac OS X sowie Linux verfügbar. Die Solaris-Version befindet sich derzeit in einem Public-Beta-Stadium.

## Flex und SOA

Während das Flex-Framework über die clientseitigen Serviceklassen *mx.rpc.http.HTTPService* und *mx.rpc.soap.WebService* einen einfach zu realisierenden Zugriff auf entfernte HTTP-basierte Dienste wie XML-RPC-Endpunkte oder SOAP Web Service ermöglicht, so stellt sich in der Praxis schnell ein wesentliches Manko dieses Ansatzes heraus: Die vom Flex-Client zur Laufzeit erhaltenen Daten sind nur bedingt typisiert und müssen in der Regel manuell über ActionScript geparkt und in eine typisierte Form gebracht werden. Die aus der Java-Welt bekannte vielfältige Unterstützung durch entsprechende Tools oder Stub-Generatoren vermisst man in der aktuellen Version von Flex schmerzlich.

Zwar kann Flex Builder 3 mit einem auf Axis basierenden "WSDL Import Wizard" aufwarten, der anhand einer WSDL den ActionScript-Stub-Code generiert und so eine strikt typisierte Kommunikation ermöglicht, das Ergebnis erfordert in den meisten Fällen aber manuelle Überarbeitung bzw. ist oftmals schlichtweg nicht brauchbar. Somit stellt XML-RPC und SOAP zwar

eine einfache Möglichkeit dar, Flex an vorhandene Services anzubinden, in der Praxis ist man aber gut beraten, auf eine andere Form der Datenübertragung und -serialisierung auszuweichen.

(Link zum Artikel: <http://www.it-republik.de/jaxenter/artikel/2330>)

### "Pimp my RPC" mit AMF

Die mit Abstand effizienteste Möglichkeit, typisierte Daten zwischen einem Flex-Client und einer entfernten Serviceschnittstelle auszutauschen, ist die Verwendung des AMF-Protokolls. AMF steht für "Action Message Format" und ist seit dem Jahr 2001 als nativer Serialisierungs- und Deserialisierungsmechanismus im Flash Player implementiert. AMF arbeitet intern ähnlich wie SOAP und kann verschiedene Nachrichtentypen transportieren, komplexe Objektdaten werden dabei typischer im binären AMF-Stream abgelegt. Die aktuelle Version des AMF-Protokolls wird AMF3 genannt, unterstützt ActionScript 3 und wurde von Adobe erstmals im Flash Player 9 implementiert. Seit Dezember 2007 liegt eine offizielle Spezifikation des vormals proprietären AMF-Protokolls [seitens Adobe vor](#).

Da die Spezifikation keinen Transportmechanismus für das AMF-Protokoll vorgibt, können AMF-Daten auf vielfältige Weise zwischen Client und Server ausgetauscht werden. Typischerweise werden AMF-Daten aber von einem Flex-Client über einen HTTP Post Request mit dem Content-Type *application/x-amf* an einen entsprechenden Endpunkt verschickt und der Server antwortet entsprechend mit einer im AMF-Format encodierten Response. Im Vergleich zu XML-RPC oder SOAP ist der Protokoll-Overhead von AMF verschwindend gering, einen Leistungsvergleich der verschiedenen Transportmechanismen finden Sie [hier](#). Durch die Offenlegung des AMF-Protokolls existieren mittlerweile für nahezu jede gängige Programmiersprache entsprechende AMF-Endpunktimplementierungen, sodass Flex-Clients mit geringem Aufwand über AMF an Java, .NET, PHP oder Ruby on Rails angebunden werden können.

### BlazeDS

Adobes eigene AMF-Implementierung für Java findet sich in dem Open-Source-Produkt [BlazeDS](#), einer leichtgewichtigen Java-EE-Webanwendung, die als Messaging und Remoting Gateway für Flex-Anwendungen fungiert und eine einfache Anbindung von Flex-Clients an Java Backends ermöglicht. Über XML-Konfigurationsdateien können in BlazeDS verschiedene Services und deren Endpunkte definiert werden. Ein solcher Service kann z.B. ein *MessagingService* sein, mit dem eine Anbindung von Flex-Clients an JMS möglich wird, oder ein *RemotingService*, der Flex-Anwendungen RPC-Aufrufe auf Java-Klassen ermöglicht.

Um eine Java-Klasse als RPC-Service für Flex-Clients bereitzustellen, muss die Klasse in BlazeDS als so genannte *Destination* exportiert werden. Eine Destination verfügt neben einem symbolischen Namen (z.B. *productService*) über den voll qualifizierten Klassennamen der Java-Klasse, die exportiert werden soll. Auf diese Weise lassen sich einfache POJO-Klassen als Servicefacade in BlazeDS bereitstellen. Eine so in BlazeDS definierte Klasse kann dann transparent von Flex-Clients aufgerufen werden. Jede als *public* deklarierte Methode der Serviceklasse ist dann für den Remote-Aufruf verfügbar. So eine Servicefassade kann dann z.B. auf EJBs oder anderweitig vorhandene Businesslogik zugreifen (Listing 1).

1. <!-- Beispiel für eine Destination in BlazeDS -->
2. <destination id="productService">
3. <properties>
4. <source>de.richinternet.service.ProductService</source>
5. <scope>application</scope>
6. </properties>
7. </destination>

Listing 1

Neben diesen Destination-Definitionen müssen in BlazeDS außerdem *Channels* konfiguriert werden. Ein Channel definiert ein Kommunikationsprotokoll (z.B. AMF), den Transportmechanismus (z.B. HTTPS) sowie den URL des eigentlichen Endpunkts in BlazeDS. Channels können in BlazeDS mehrfach wiederverwendet werden und müssen letztlich einer Destination zugeordnet werden. Erst durch diese Zuordnung kann ein exportierter Service überhaupt aufgerufen werden.

Den Kern von BlazeDS stellt schließlich der *MessageBroker* dar, der über das *MessageBrokerServlet* mit AMF/HTTP-Requests bedient wird, die von Flex-Clients stammen. Der MessageBroker leitet die AMF-Nachrichten dann entsprechend an die jeweils zuständige Destination weiter und ruft auf der entsprechenden Serviceklasse die vom Flex-Client gewünschte Methode auf. Eventuelle Rückgabewerte der Methode werden entsprechend als AMF-Paket mit dem HTTP-Response zurück an die Flex-Anwendung übermittelt.

Um die Serialisierung der Daten zwischen Flex und Java muss man sich übrigens nicht mehr kümmern, das übernimmt BlazeDS unter Verwendung des AMF-Protokolls selbst: Zwischen den primitiven Datentypen in ActionScript 3 und Java findet ein Standard-Mapping statt. Komplexe Typen, die z.B. als DTOs in Java implementiert sind, werden typischer in AMF serialisiert, der voll qualifizierte Klassenname wird ebenfalls mit in AMF verpackt. Wird ein solches Paket auf Flex-Seite empfangen, so versucht der Flash Player eine entsprechend vom Entwickler gemappte *ActionScript*-Klasse zu instanzieren und befüllt die Instanz der Klasse mit den entsprechenden Daten. Gibt es keine mappende Klasse, so wird ein anonymes Objekt erstellt. Über so genannte *BeanProxies* kann dieses Standardverhalten in BlazeDS an eigene Bedürfnisse angepasst werden.



[zurück](#)



[nach oben](#)



[drucken](#)



[kommentieren](#)

---

Anzeige



**Jetzt als IT-Solution-Designer  
(m/w) durchstarten: beim**

**“Take-off 2010”**

Bewerben Sie sich für das Accenture Recruiting-Event “Take-off 2010” am Frankfurter Flughafen. Lösen Sie die IT-Probleme von Fluglinien und testen Sie Ihr Geschick in einem echten Flugsimulator. Bewerben Sie sich jetzt online, um direkt in Ihre berufliche Zukunft durchzustarten:  
[entdecke-accenture.com/take-off](http://entdecke-accenture.com/take-off)

(Link zum Artikel: <http://www.it-republik.de/jaxenter/artikel/2331>)

### BlazeDS und Spring integriert

Ein wesentliches Feature von BlazeDS ist, dass die über die angesprochenen Destinations bereitgestellten Java-Klassen keine Abhängigkeiten zu BlazeDS besitzen und ähnlich wie bei Spring als einfache POJOs implementiert werden können. Ansonsten ist BlazeDS relativ genügsam in Bezug auf den Container, in dem es deployt wird. Das Einzige, was BlazeDS bisher fehlte, war eine enge Integrationsmöglichkeit mit dem Spring Framework. Das haben sich wohl auch Adobe und SpringSource gedacht, als sie gemeinsam das Spring-BlazeDS-Integration-Projekt [auf den Weg gebracht haben](#). Neben dem Ziel, neue Entwickler für die Flex-Technologie zu begeistern, soll es natürlich auch bereits überzeugten Entwicklern erleichtert werden, Flex an Spring anzubinden.

Die zum Redaktionsschluss vorliegende Version 1.0.0.M2 von Spring BlazeDS Integration kommt in der kompilierten Version als knapp 50 KB große JAR-Datei daher. Die Mindestvoraussetzungen für den Einsatz sind Java 5, Spring Framework 2.5.0 sowie BlazeDS 3.2. Die Beispiele für diesen Artikel wurden mit Eclipse 3.4 in einem Dynamic Web Project erstellt.

Ein erstes Test-Setup ist verhältnismäßig schnell erstellt: Nach dem Download der erforderlichen Dateien wird ein neues Dynamic Web Project erstellt. Die BlazeDS WAR-Datei wird manuell extrahiert und der enthaltene *WEB-INF*-Ordner in das neue Projekt kopiert. Danach fügt man *spring.jar*, *spring-webmvc.jar* sowie *org.springframework.flex-1.0.0.M2.jar* zum Projekt hinzu. Als Nächstes folgt die Anpassung der *web.xml*-Datei: Hier wird das von der BlazeDS-Webanwendung definierte *MessageBrokerServlet* nebst zugehörigem Servlet-Mapping entfernt, stattdessen konfiguriert man Springs *DispatcherServlet* (Listing 2).

```
1. <!-- Ausschnitt aus der web.xml -->
2. <servlet>
3. <servlet-name>DispatcherServlet</servlet-name>
4. <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
5. <init-param>
6. <param-name>contextConfigLocation</param-name>
7. <param-value>/WEB-INF/config/web-application-config.xml</param-value>
```

```
8. </init-param>
9. <load-on-startup>1</load-on-startup>
10. </servlet>
11. <servlet-mapping>
12. <servlet-name>DispatcherServlet</servlet-name>
13. <url-pattern>/dispatcher/*</url-pattern>
14. </servlet-mapping>
```

#### Listing 2

Somit hat das *DispatcherServlet* nun die Entscheidungshoheit über ankommende HTTP-Requests. Der für das Verteilen der AMF-Nachrichten zuständige *MessageBroker* von BlazeDS wird nun von Spring verwaltet und über eine *MessageBrokerFactoryBean* in Springs Web ApplicationContext als Bean definiert (in diesem Beispiel in der *web-application-config.xml*).

In der aktuellen M2-Version müssen die BlazeDS-eigenen XML-Konfigurationsdateien hinsichtlich der Channels bedauerlicherweise noch manuell gepflegt werden. In der Roadmap ist aber bereits vorgesehen, dass diese Konfiguration in Zukunft ebenfalls im ApplicationContext stattfinden wird. Standardmäßig verwendet die *MessageBrokerFactoryBean* den Pfad */WEB-INF/flex/services-config.xml*, um die Channel-Konfiguration auszulesen, diese Eigenschaft kann aber entsprechend angepasst werden (Listing 3).

```
1. <!-- Spring-managed MessageBroker -->
2. <bean id="springManagedMessageBroker"
3. class="org.springframework.flex.messaging.MessageBrokerFactoryBean">
4. <property name="servicesConfigPath" value="classpath*:my-service-config.xml"/>
5. </bean>
```

#### Listing 3

Um den von Spring verwalteten *MessageBroker* mit ankommenden AMF-Anfragen zu bestücken, ist schließlich noch ein entsprechendes URL-Mapping einzurichten. Sollte die Spring-Anwendung ausschließlich von Flex-Clients aufgerufen werden, reicht hier ein einfaches *Wildcard*, um alle Requests an den *MessageBroker* durchzureichen. Soll neben Flex-Clients auch "klassischen" HTTP-Clients der Zugriff ermöglicht werden, so ist das Mapping entsprechend anzupassen. Die eigentliche Delegation des AMF-Requests an den *MessageBroker* übernimmt schließlich eine entsprechende *HandlerAdapter*-Implementierung (Listing 4).

```
1. <!-- URL Mapping auf den Spring-managed Messagebroker -->
2. <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
3. <property name="mappings">
4. <value>
5. /*=springManagedMessageBroker
```

```

6. </value>
7. </property>
8. </bean>
10. <!-- HandlerAdapter für den MessageBroker -->
11. <bean
12. class="org.springframework.flex.messaging.servlet.MessageBrokerHandlerAdapter" />

```

Listing 4

Nach dieser Vorarbeit können nun Spring Beans als Services für Flex exportiert werden. In unserem Beispiel gibt es ein *ProductService*-Interface, das von der *ProductServiceMock*-Klasse implementiert wird. Der Ansatz zum Export der Serviceklasse ähnelt hier dem Vorgehen, das auch bei Spring Remoting Anwendung findet: Im *ApplicationContext* definierte Beans werden über einen entsprechenden Exporter für den Remote-Aufruf verfügbar gemacht, in diesem Fall übernimmt das der *FlexRemotingServiceExporter*.

"Unter der Haube" bedeutet dies, dass zur Laufzeit der Anwendung eine neue Destination in BlazeDS generiert wird. Der symbolische Name der Destination ist die ID des Exporters, überschreiben lässt sich dies über die *serviceId*-Eigenschaft. Weiterhin lässt sich das Standardverhalten von BlazeDS, alle *public*-Methoden der jeweiligen Serviceklasse für Flex-Clients aufrufbar zu machen, über die Eigenschaften *includeMethods* und *excludeMethods* entsprechend einschränken (Listing 5).

```

1. <!-- ProductService Implementierung -->
2. <bean id="productServiceImpl" class="de.richinternet.service.ProductServiceMock" />
3. <!-- ProductService exportieren -->
5. <bean id="productServiceExporter"
6. class="org.springframework.flex.messaging.remoting.FlexRemotingServiceExporter" >
7. <property name="messageBroker" ref="springManagedMessageBroker" />
8. <property name="service" ref="productServiceImpl" />
9. <property name="serviceId" value="productService" />
10. <property name="excludeMethods" value="deleteProduct" />

```

Listing 5

Ein Flex-Client dürfte somit bis auf die *deleteProduct*-Methode alle als *public* markierten Methoden der Serviceklasse aufrufen. Versuche, die "verbotene Methode" trotzdem aufzurufen, würden mit einer entsprechenden Ausnahme quittiert werden, die an den Flex-Client weitergereicht wird und dort in Form eines *FaultEvents* signalisiert wird.



## Anzeige



**Jetzt als IT-Solution-Designer  
(m/w) durchstarten: beim**

### “Take-off 2010”

Bewerben Sie sich für das Accenture Recruiting-Event “Take-off 2010” am Frankfurter Flughafen. Lösen Sie die IT-Probleme von Fluglinien und testen Sie Ihr Geschick in einem echten Flugsimulator. Bewerben Sie sich jetzt online, um direkt in Ihre berufliche Zukunft durchzustarten:  
[entdecke-accenture.com/take-off](http://entdecke-accenture.com/take-off)

(Link zum Artikel: <http://www.it-republik.de/jaxenter/artikel/2332>)

Die Konfiguration der Serverseite ist damit abgeschlossen. Ein einfacher Flex-Client, der testweise den über Spring exportieren Service aufrufen kann, ist schnell entworfen: Eine *RemoteObject*-Instanz dient als clientseitiger dynamischer Proxy für die entfernte Serviceklasse. Zur Anzeige der Daten wird ein einfaches DataGrid verwendet. Bei Klick auf DATEN LADEN wird die *getAllProducts()*-Methode auf dem *RemoteObject* aufgerufen. Dieser Methodenaufwurf wird dann als AMF-Request über HTTP an das *DispatcherServlet* geschickt und dort, wie oben beschrieben, weiterverarbeitet. Die Antwort, die übrigens asynchron eintrifft, wird über ein Data Binding im DataGrid angezeigt (Listing 6).

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >
3. <mx:RemoteObject
4. id="service"
5. destination="productService"
6. endpoint="http://localhost:8080/SpringBackend/dispatcher/flex/amf" />
7. <mx>DataGrid id="grid"
8. dataProvider="{service.getAllProducts.lastResult}" />
9. <mx:Button label="Daten laden"
10. click="service.getAllProducts()" />
11. </mx:Application>
```

## Listing 6

Damit ist auch der grundlegende Funktionsumfang der M2-Version des Spring-BlazeDS-Integration-Projekts abgedeckt. Weitere nennenswerte Highlights sind die Integration mit Spring Security, um den Remote-Zugriff auf exportierte Klassen feingranuliert zu steuern, sowie ein eigener XML-Namespace, der die Bean-Konfiguration immens vereinfacht. Der komplette Quellcode für Spring und Flex findet sich [hier \(Rubrik Quellcode\)](#).

### Fazit

Der erste Wurf von Spring BlazeDS Integration hinterlässt einen guten Eindruck und macht Lust auf mehr. Für den nächsten Meilenstein ist geplant, BlazeDS komplett über Springs ApplicationContext zu konfigurieren. Außerdem soll die Möglichkeit bestehen, neben dem oben beschriebenen *RemotingService* auch den in BlazeDS verfügbaren *MessageService* über entsprechende Exporter zur Laufzeit bereitzustellen, mit dem z.B. Server-Push-Szenarien oder Anbindungen an JMS möglich sind. Die finale Version 1.0.0 ist für Ende März terminiert.

Mit Spring BlazeDS Integration bietet sich interessierten Entwicklern eine gute Möglichkeit, einen Blick auf Flex zu werfen: Die Anbindung von Flex an Spring war noch nie einfacher – probieren Sie es aus!

*Dirk Eismann ist Softwareentwickler und Consultant bei der Herrlich & Ramuschkat GmbH. Der Schwerpunkt seiner Arbeit ist die Entwicklung von Unternehmensanwendungen auf Basis von Flex, BlazeDS, LiveCycle und Java EE.*

### Links & Literatur

1. <http://opensource.adobe.com/wiki/display/flexsdk/>
2. <http://www.springsource.org/node/904>
3. <http://www.jamesward.com/census/>
4. <http://opensource.adobe.com/wiki/display/blazeds/>
5. <http://www.springsource.org/spring-flex>



[zurück](#)



[nach oben](#)



[drucken](#)



[kommentieren](#)

---

## Anzeige

  
Technology Solutions  
“Take-off 2010”

**Jetzt als IT-Solution-Designer**  
(m/w) **durchstarten: beim**

Bewerben Sie sich für das Accenture  
Recruiting-Event “Take-off 2010” am Frankfurter  
Flughafen. Lösen Sie die IT-Probleme von  
Fluglinien und testen Sie Ihr Geschick in einem  
echten Flugsimulator. Bewerben Sie sich jetzt  
online, um direkt in Ihre berufliche Zukunft  
durchzustrarten:  
[entdecke-accenture.com/take-off](http://entdecke-accenture.com/take-off)

© 1995–2010 Software & Support Verlag GmbH. Vervielfältigung nur mit Genehmigung des Verlags.