# Anyframe Flex Query Plugin

**Version 1.1.0**

저작권 © **2007-2011** 삼성**SDS**

본 문서의 저작권은 삼성**SDS**에 있으며 **Anyframe** 오픈소스 커뮤니티 활동의 목적하에서 자유로운 이용이 가능합니다. 본 문서를 복제, 배포할 경우에는 저작권자를 명시하여 주시기 바라며 본 문서를 변경하실 경우에는 원문과 변경된 내용을 표시하여 주시기 바랍니다. 원문과 변경된 문서에 대한 상업적 용도의 활용은 허용되지 않습니다. 본 문서에 오류가 있다고 판단될 경우 이슈로 등록해 주시면 적절한 조치를 취하도록 하겠습니다.

I. Introduction ..... 1
II. Flex Query ..... 2
1. Introduction ..... 3
1.1. Flex ..... 3
1.1.1. Flex의 동작 원리 ..... 3
1.1.2. MXML의 컴파일과 Flex Builder ..... 3
1.2. BlazeDS ..... 3
2. Spring BlazeDS Integration 설치 ..... 5
2.1. BlazeDS의 설치 ..... 5
2.1.1. BlazeDS 설정 파일 ..... 5
2.1.2. BlazeDS 라이브러리 ..... 6
2.2. Spring BlazeDS Integration 설치 ..... 6
2.2.1. Spring BlazeDS Integration 라이브러리 ..... 6
2.2.2. Servlet 설정 ..... 7
2.2.3. Spring WebApplicationContext 설정 ..... 8
2.3. Spring Bean Exporting ..... 9
2.4. Flex Project 생성 ..... 9
2.5. Remoting Service Call ..... 14
3. Spring BlazeDS Integration 환경 설정 ..... 16
3.1. Spring BlazeDS MessageBroker 환경 설정 ..... 16
3.1.1. Spring DispatcherServlet ..... 16
3.1.2. MessageBroker ..... 16
3.2. Exporting Spring Beans ..... 18
3.2.1. RemotingService 환경 설정 ..... 18
3.2.2. remoting-destination 태그 ..... 18
3.2.3. @RemotingDestination ..... 18
4. Flex의 Data 연동 ..... 20
4.1. Flex 기본 Data 연동 ..... 20
4.1.1. HTTPService ..... 20
4.1.2. WebService ..... 20
4.2. BlazeDS를 이용한 Data 연동 ..... 21
4.2.1. HTTPProxyService ..... 21
4.2.2. RemotingService ..... 22
4.3. Domain객체와 ASObject의 Mapping ..... 22
5. FlexService ..... 24
5.1. DataSet ..... 24
5.1.1. DataSet ..... 24
5.2. DataService ..... 24
5.3. Service 클래스 생성 ..... 25
5.3.1. FlexService ..... 25
5.3.2. FlexServiceImpl ..... 25
5.4. RemotingService ..... 26
5.4.1. Page 조회 ..... 26
5.4.2. saveAll메소드 ..... 28

# I.Introduction

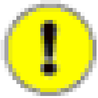
BlazeDS는 remoting, messaging 등, Flex 클라이언트와 Java 서비스간의 다양한 연결 채널을 제공하는 Adobe의 오픈 소스 프로젝트이다. Spring은 BlazeDS와의 손쉬운 연계 방안을 제시하여 Flex를 사용한 Spring 기반의 Rich Internet Application 개발을 가능하게 한다. Anyframe의 flex-query plugin은 이 BlazeDS의 설치는 물론, 실제 개발 시에 참조할 수 있는 다양한 Flex UI Sample, 그리고 Spring과 BlazeDS의 연계를 위한 라이브러리들과 설정파일들을 제공하고 있다.

**Installation**

Command 창에서 다음과 같이 명령어를 입력하여 flex query plugin을 설치한다.

```
mvn anyframe:install -Dname=flex-query
```

installed(mvn anyframe:installed) 혹은 jetty:run(mvn clean jetty:run) command를 이용하여 설치 결과를 확인해볼 수 있다.

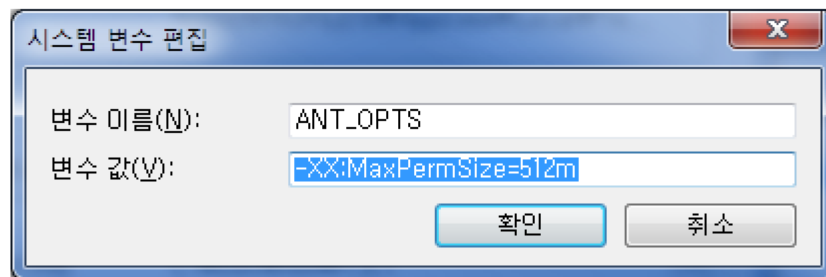


## out of memory exception : Java Heap Space 해결 방법

Flex Query Plugin은 디자인을 위한 폰트, 이미지, CSS등이 포함 되어 있으므로 mxml 컴파일 시 512m 이상의 permGen 메모리가 필요하다. Plugin 설치 전 반드시 512m 이상의 permGen메모리를 아래와 같이 설정한다.

MAVEN을 이용한 설치 시

ANT을 이용한 설치 시




**Dependent Plugins**

| Plugin Name | Version Range |
| --- | --- |
| Query [http://dev.anyframejava.org/docs/anyframe/plugin/optional/query/1.1.1/reference/htmlsingle/query.html] | 2.0.0 > * |
| Fileupload [http://dev.anyframejava.org/docs/anyframe/plugin/optional/query/1.0.1/reference/htmlsingle/fileupload.html] | 2.0.0 > * |

# II.Flex Query

Spring, BlazeDS, Flex를 연계하기위한 방법에 대해서 상세히 살펴 보도록 하자.

# 1.Introduction

Abobe의 Flex는 MXML를 작성해 플래시로 된 화면을 만드는 RIA개발 솔루션이다. 컴파일된 결과가 플래시 파일로 만들어지기 때문에 플래시플레이어만 설치되어 있다면 어떠한 브라우저에서 실행 가능하고 플래시 화면처럼 화려하고 다양한 화면을 만들 수 있다.

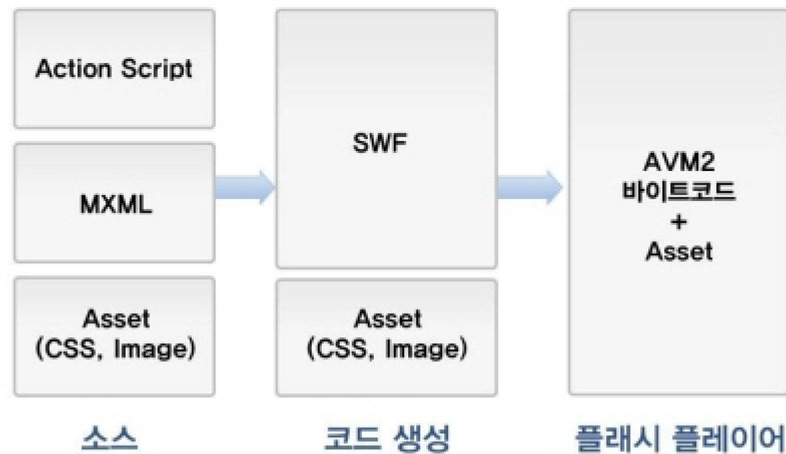
Flex에서 기본으로 제공하는 HTTPService만으로도 Web Application개발이 가능하기는 하지만 데이터 전송 방식이 XML이라는데 많은 제약이 있다. BlazeDS를 사용하게 되면 HTTPService뿐만 아니라 RemotingService, MessageService등을 사용해 Client와 Server간의 데이터를 주고 받을 수 있다.

Spring BlazeDS Integration은 BlazeDS와 Spring Framework를 이용해 Flex Web Application을 개발 할 때, 복잡하고 어려운 설정들을 간소화 할 수 있는 기능들을 제공한다.

# 1.1.Flex

## 1.1.1.Flex의 동작 원리

Flex는 기본적으로는 Action Script, MXML, Asset(CSS, Image)로 구성되어 있고 컴파일 된 SWF파일이 플래시 플레이어에서 실행 된다.




## 1.1.2.MXML의 컴파일과 Flex Builder

MXML파일을 컴파일 하기 위해서는 아래와 같이 Flex SDK에서 제공하는 mxmlc.exe명령어를 이용하면 된다.

```
mxmlc -file -specs Main.mxml
```

Flex Builder 3는 Flex개발을 위한 IDE이며, mxmlc와 compc 컴파일러가 포함되어 있어 mxml을 자동으로 컴파일해준다. Flex Builder 3는 독립실행 버전과 Eclipse플러그인 버전이 있다. 독립실행 버전은 별도의 이클립스 설치가 필요 없지만, JDT(Java Development Tools)를 사용 할 수 없다.

Flex Builder 3는 MXML의 자동 컴파일 외에 디버깅, 자동완성, 모니터링 등의 개발 시 필요한 기능들과 컴포넌트 디자인을 쉽게 할 수있도록 디자인 뷰를 제공한다.

# 1.2.BlazeDS

Flex에서 기본으로 제공하는 Data통신 방식인 HTTPService는 XML형태로 데이터를 주고 받기 때문에 대용량 데이터 전송에 적합하지 않고 MXML의 소스코드내에 url정보가 입력되기 때문에 보안상 문제

가 있다. BlazeDS는 Client와 Server사이에 데이터 전송을 위해 Adobe에서 제공하는 Opensource S/W 이다. BlazeDS는 크게 다음과 같이 구성되어 있다.

- **Client** : RemoteObject, HTTPService, WebService, Producer, Consumer
- **Server** : BlazeDS 라이브러리, BlazeDS 설정 파일, MessageBrokerServlet

BlazeDS는 Client와 Server 사이의 데이터 전송 방식으로 RPC(HTTPService, WebService, RemoteObject) Service, Messaging Service를 제공하는데 각 Service에 대한 상세 내용은 BlazeDS를 이용한 Data 연동을 참고한다.

# 2.Spring BlazeDS Integration 설치

본 문서에서는 Flex Plugin을 사용하지 않고 별도로 Flex개발 환경을 구성하는 방법에 대해서 다루도록 한다. Spring BlazeDS Integration을 설치 하기 위해서는 BlazeDS가 먼저 설치 되어야 한다. Web Application 프로젝트에 BalzeDS를 설치 한 후, Spring BlazeDS Integration을 추가로 설치 하는 방법에 대해 살펴 보도록한다.

## 2.1.BlazeDS의 설치

Adobe Open Source [http://opensource.adobe.com/wiki/display/blazeds/Downloads] Site에서 최신 버전의 BlazeDS을 다운 받을 수 있다. Turnkey버전과 Binary Distribution을 다운 받을 수 있다. Turnkey의 경우 BlazeDS설치 파일 이외에 Reference 문서, Sample Application, BlazeDS Console Application 등이 포함되어 있다.

BlazeDS를 다운 받아 압축을 풀면 blazeds.war파일이 포함되어 있다. blazeds.war파일은 아래와 같은 폴더들로 구성된다.

다음은 각 폴더와 파일들에 대한 간략한 설명이다.

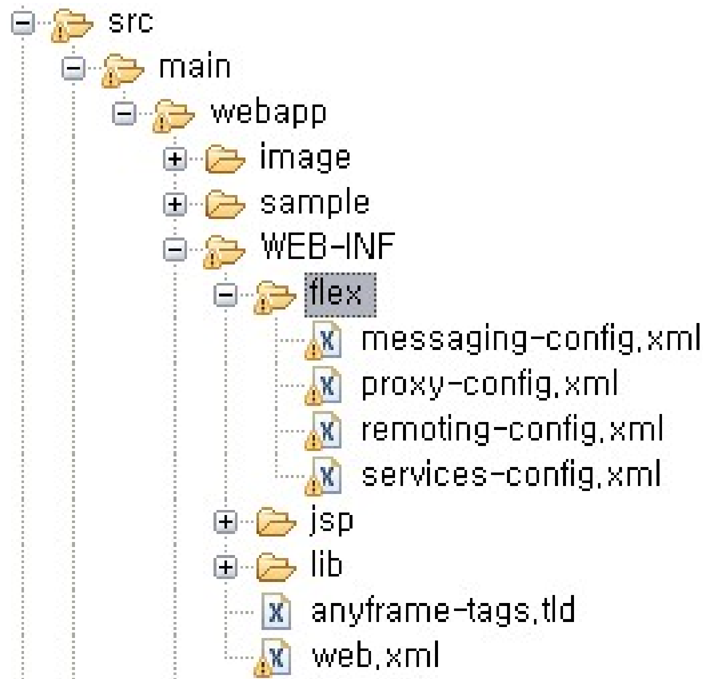
- **WEB-INF/web.xml** : 어플리케이션의 배포지시자로 MessageBrokerServlet에 대한 Servlet설정과 HttpFlexSession 등에 대해 설정되어 있다. SpringBlazeDS Integration을 사용할 경우에는 MessageBrokerServlet이 아닌 SpringMVC의 DispatcherServlet이 Client요청을 처리 할 수 있도록 Servlet설정을 변경해야 한다.

- **WEB-INF/flex** : BlazeDS의 환경 설정 파일들이 있다. 서버측과 통신하기 위한 채널, 아답터, 로깅등의 정보가 세팅되어 있는 파일들이 위치한다.

- **WEB-INF/lib** : BlazeDS 서버 측 라이브러리들이 위치해 있다.

### 2.1.1.BlazeDS 설정 파일

BlazeDS의 WEB-INF/flex폴더의 4개의 xml파일(services-config.xml, remoting-config.xml, proxy-config.xml, messaging-config.xml)을 Web Application Project의 {Web Root folder}/WEB-INF/flex란 이름의 폴더를 생성한 후 아래 그림과 같이 복사한다.




service-config.xml 파일을 열어 remoting-config.xml, messaging-config.xml파일의 include 부분을 삭제 또는 주석처리 한다. 두 파일을 include에서 제외하는 이유는 Spring BlazeDS Integration의 설정파일에서 Remoting, Messaging 통신에 대한 환경설정을 할 수 있기 때문이다. 그리고 Web Application 레벨의 default-channels을 설정한다. 아래는 수정된 service-config.xml파일의 일부이다.

```
<services-config>
    <services>
        <!-- service-include file-path="remoting-config.xml" / -->
        <service-include file-path="proxy-config.xml" />
        <!-- service-include file-path="messaging-config.xml" / -->
        <default-channels>
         <channel ref="my-amf"/>
     </default-channels>
    </services>

    <security>
    <!-- 중략 -->
```

## 2.1.2.BlazeDS 라이브러리

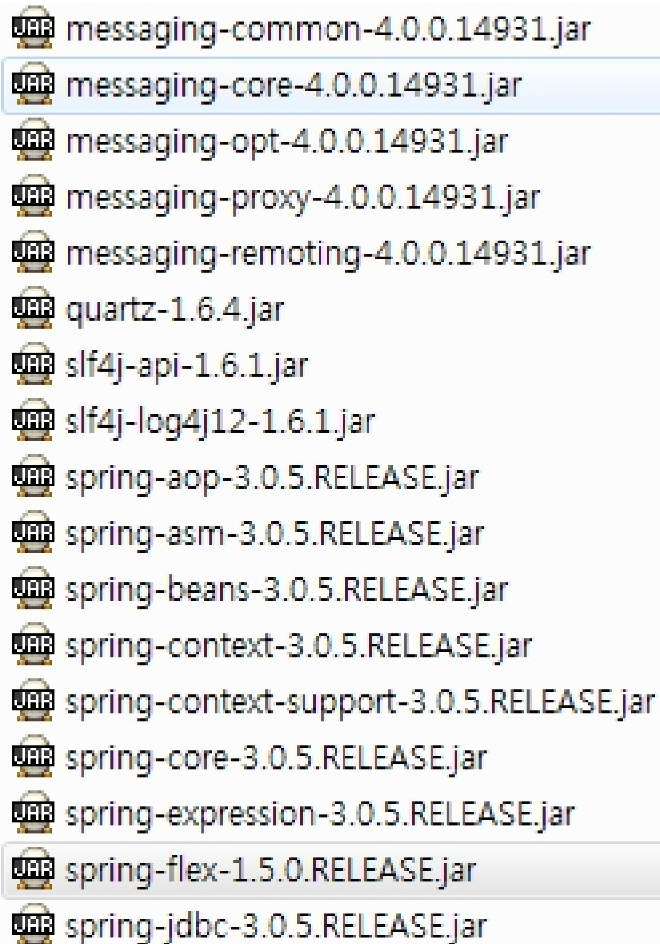
BlazeDS의 WEB-INF/lib폴더의 BlazeDS 서버 측 라이브러리를 Web Application Project의 {Web Root folder}/WEB-INF/lib폴더에 복사한다. 이 때 Web Application에 이미 등록되어 있는 라이브러리와 충돌나지 않도록 확인해서 복사한다. 예를들어 Anyframe의 Core Plugin으로 설치된 Web Application에는 common-logging-1.1.1.jar을 포함하고 있으므로 common-logging.jar은 제외한 후 복사한다.

# 2.2.Spring BlazeDS Integration 설치

BlazeDS설치가 끝났으면 Spring BlazeDS Integration을 Web Application에 설치 하도록 한다.

## 2.2.1.Spring BlazeDS Integration 라이브러리

Spring Source Community [http://www.springsource.org/spring-flex]에서 Spring BlazeDS Integration 1.5.0을 다운 받아 org.springframework.flex-1.5.0.RELEASE.jar파일을 Web Application Project의 {Web Root folder}/WEB-INF/lib에 복사한다.




## 2.2.2.Servlet 설정

Spring BlazeDS Integration 라이브러리 복사가 끝났으면, Flex UI에서 RPC요청이 왔을 때 MessageBrokerServlet이 아닌 Spring MVC의 DispatcherServlet이 요청을 처리 할 수 있도록 web.xml파일에 Servlet설정을 아래와 같이 추가 한다. 이 때 Spring MVC에 대한 Servlet설정이 이미 정의되어 있으면 새로운 Servlet을 추가 정의한다.

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:/springmvc/common-servlet.xml,
                    classpath:/springmvc/generation-servlet.xml,
        </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet>
    <servlet-name>SpringBlazeDS</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
```

```
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:/springmvc/flex-servlet.xml
        </param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
```

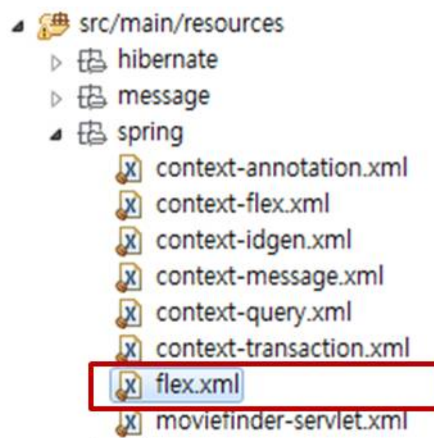
위와 같이 SpringBlazeDS Servlet설정을 완료 한 후 아래와 같이 **/messagebroker/***에 대한 요청을 SpringBlazeDS Servlet이 처리 할 수 있도록 servlet-mapping을 정의한다.

```
<servlet-mapping>
    <servlet-name>SpringBlazeDS</servlet-name>
    <url-pattern>/messagebroker/*</url-pattern>
</servlet-mapping>
```

Flex Client에서 RemotingService, HTTPProxyService, MessageService의 Request url 패턴은 /messagebroker/*형태이다.

# 2.2.3.Spring WebApplicationContext 설정

다음 그림과 같이 Flex관련 Spring 설정 파일(flex.xml)을 생성하고, Web Appliaction의 classpath:/spring 하위에 위치시키도록 한다.




flex-servlet.xml 파일을 열어 flex namespace를 사용하기 위해 다음과 같이 xsd를 정의한다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:flex="http://www.springframework.org/schema/flex"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
 http://www.springframework.org/schema/flex
 http://www.springframework.org/schema/flex/spring-flex-1.5.xsd">

</beans>
```

BlazeDS 설정 파일인 services-config.xml파일을 읽어 MessageBrokerFactoryBean에 등록할 수 있도록 아래와 같이 flex namespace를 사용해 정의한다.

```
<flex:message-broker/>
```

위는 Default설정으로 services-config.xml파일이 classpath 상위에 있을 경우에 사용할 수 있다. 추가적인 설정에 대해서는 Spring BlazeDS Integration Configuration을 참고한다.

# 2.3.Spring Bean Exporting

BlazeDS, Spring BlazeDS Integration의 설치가 끝났으면 Spring Bean을 Flex의 RemotingService가 접근할 수 있도록 RemotingDestinationExporter에 등록한다. Spring Bean을 Exporting하기 위해서는 여러 방법이 있는데 여기에서는 @RemotingDestination annotation을 사용하도록 한다.

```
@Service("genreService")
@RemotingDestination
public class GenreServiceImpl implements GenreService {

 @Inject
 @Named("genreDao")
 private GenreDao genreDao;

 public List<Genre> getList() throws Exception {
  return genreDao.getList();
 }

}
```

@RemotingDestination annotation에 아무런 속성을 부여하지 않으면 Bean Id가 Destination Id가 된다. 위의 소스코드에서는 'genreService'이 Destination Id가 되고 Flex Client에서는 genreService란 이름으로 해당 Service에 접근한다.

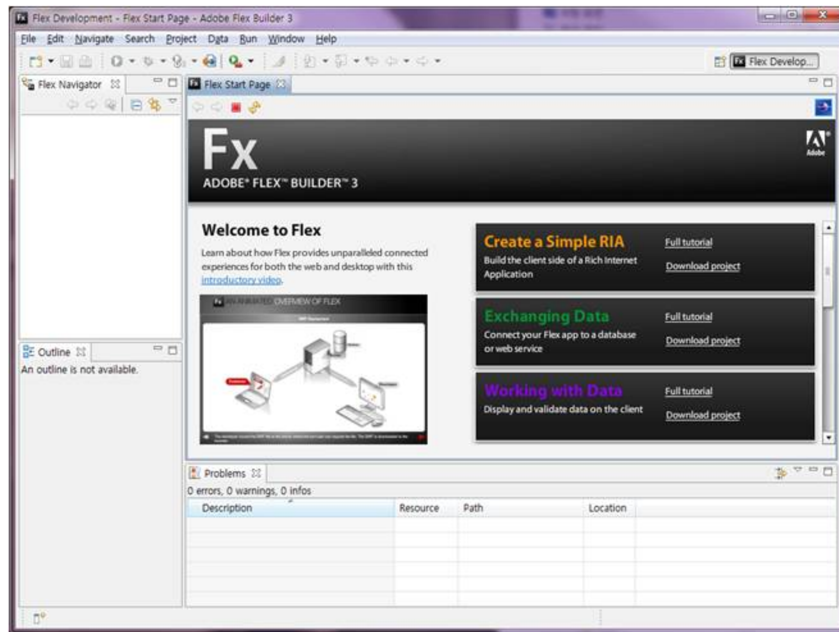
RemotingService가 접근 할 수 있도록 Spring Bean을 Exporting하는 자세한 방법은 Spring Blazeds Integration Configuration을 참고한다.

# 2.4.Flex Project 생성

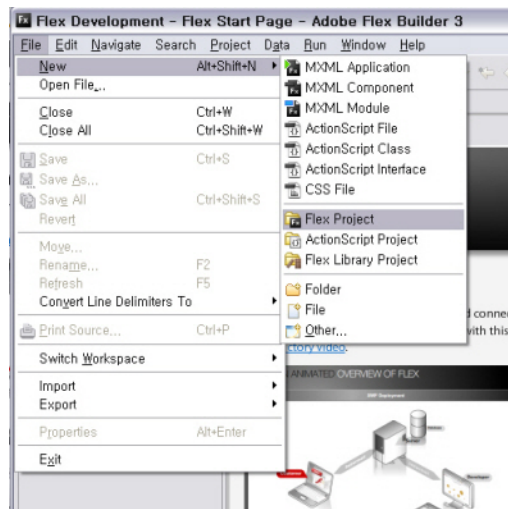
Flex Web Application개발을 하기 위한 서버 측 설정이 완료 됐으면 Flex프로젝트를 생성한다. Flex UI를 개발하기 위해서는 편집기를 이용한 방법과 Flex Builder를 이용한 방법이 있다. Flex Builder는 Flex UI를 개발하기 위한 이클립스 기반의 상용 S/W 로 WYSIWYG Editor, mxml의 자동컴파일, 디버깅등 의 다양한 기능을 제공한다. 본 문서에서는 Flex Builder를 이용해 개발하는 방법에 대해서만 설명한다.

Adobe Flex Builder 3 Download [http://www.adobe.com/cfusion/entitlement/index.cfm?e=flex3email]에서 60일동안 사용가능한 Trial버전을 다운 받을 수 있다.

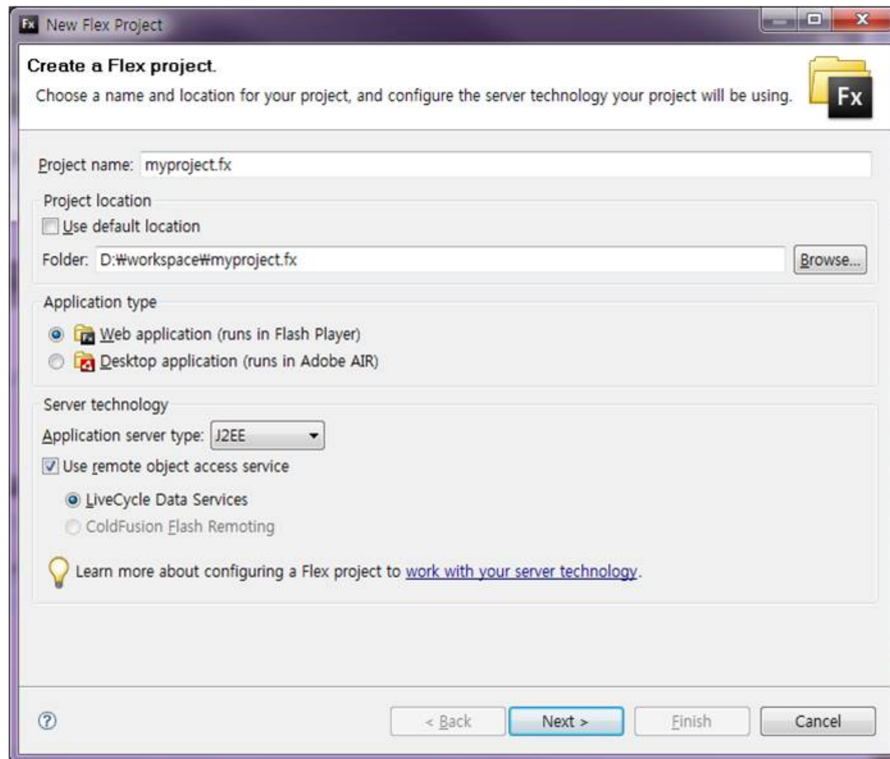
Flex Builder 3를 처음 실행 하게 되면 아래와 같은 화면을 볼 수 있다.




Flex Builder에서 File -> New -> Flex Project를 선택하면 새로운 Flex Project를 생성할 수 있다.

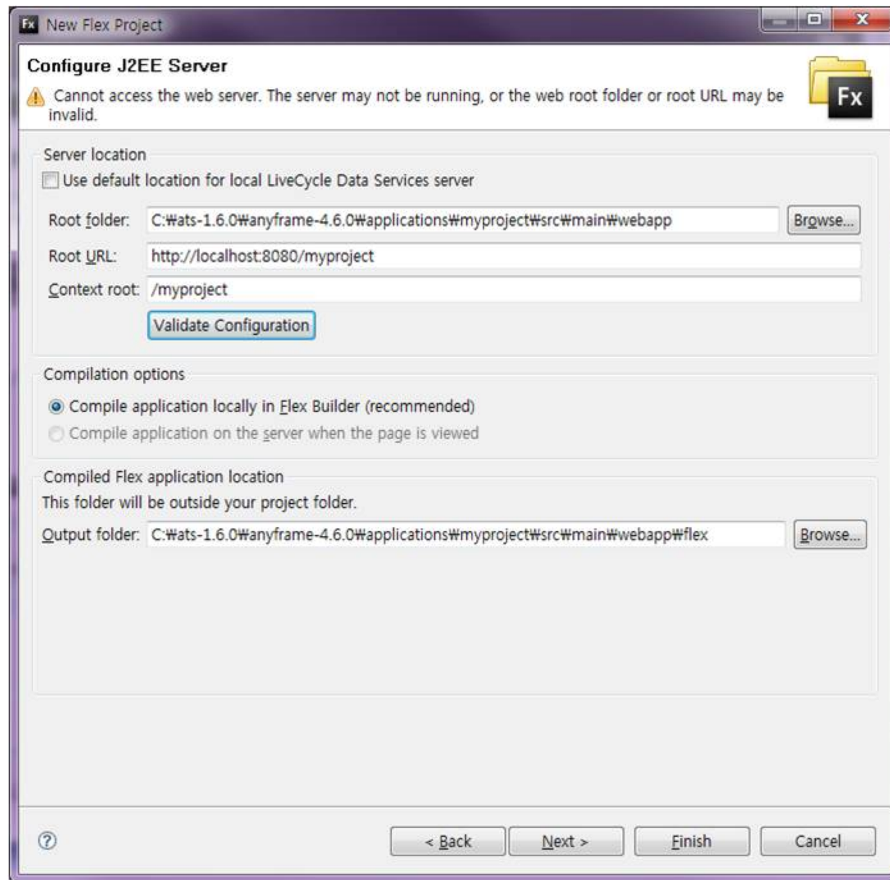



New Flex Project 창이 열리면 Project Name과 Flex Project의 Location등을 입력하고 Application type은 **Web application(runs in Flash Player)**을 선택, Server technology의 Application server type는 **J2EE**, **Use remote object access service**의 체크 박스를 선택, **LiveCycle Data Services**를 선택한 후 Next버튼을 클릭한다.

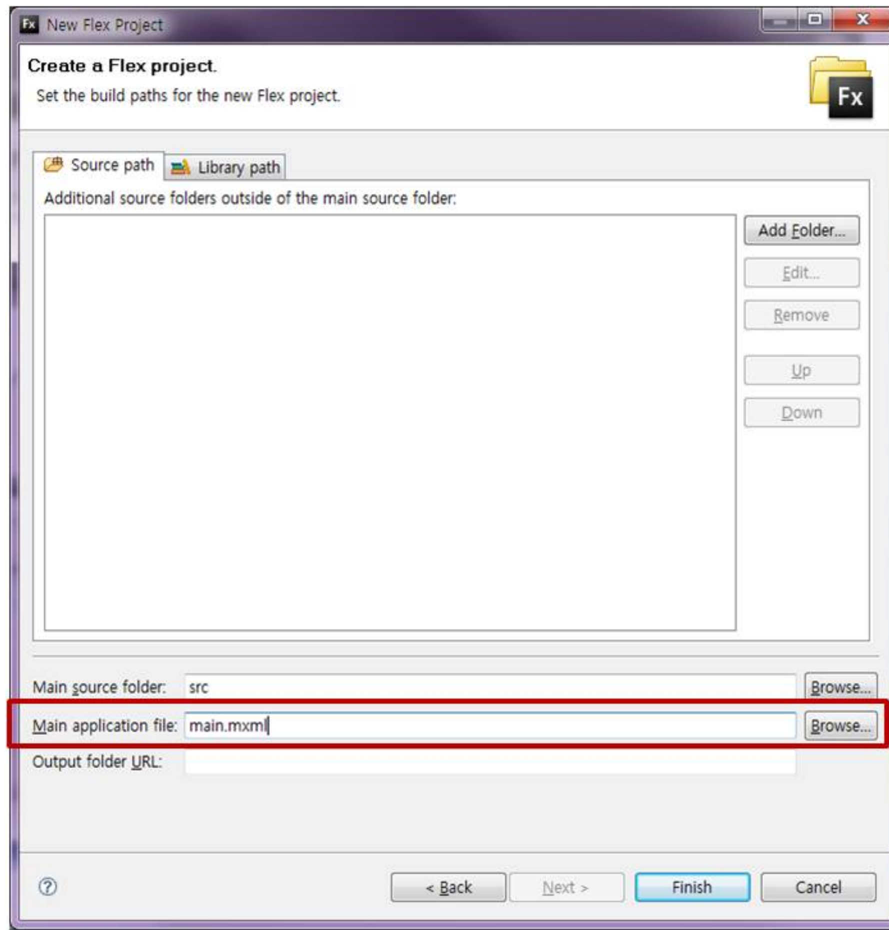



Configure J2EE Server창 Server location에서는 BlazeDS가 설치된 Web Application의 정보를 세팅한다. 서버 정보를 입력 한 후 Validate Configuration 버튼을 클릭해 BlazeDS가 설치 된 Web Application인지 확인한다.

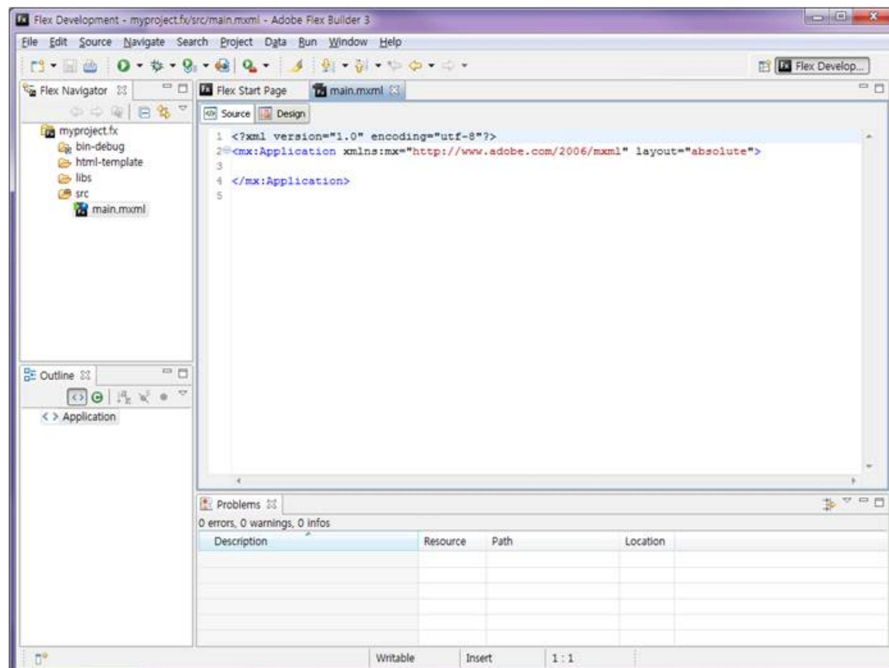
Compiled Flex application location의 Output folder는 Flex Project가 컴파일 되서 위치할 경로를 입력한다. 일반적으로는 Web Application Project의 Web Root folder 하위의 특정 폴더를 지정한다. 여기에서는 {Web Root folder}/flex를 Output folder로 설정했다. 입력이 끝났으면 Next버튼을 클릭한다.



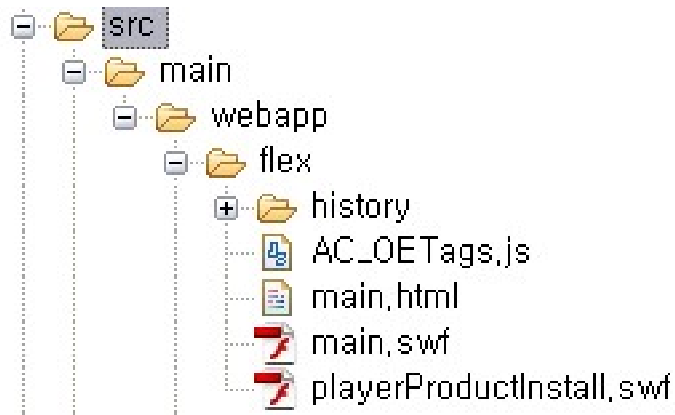

다음으로는 Flex Project의 build path를 설정하는 창이 열리는데 Main application file에 Main Application mxml파일의 파일 이름을 입력한 후 Finish버튼을 클릭한다.



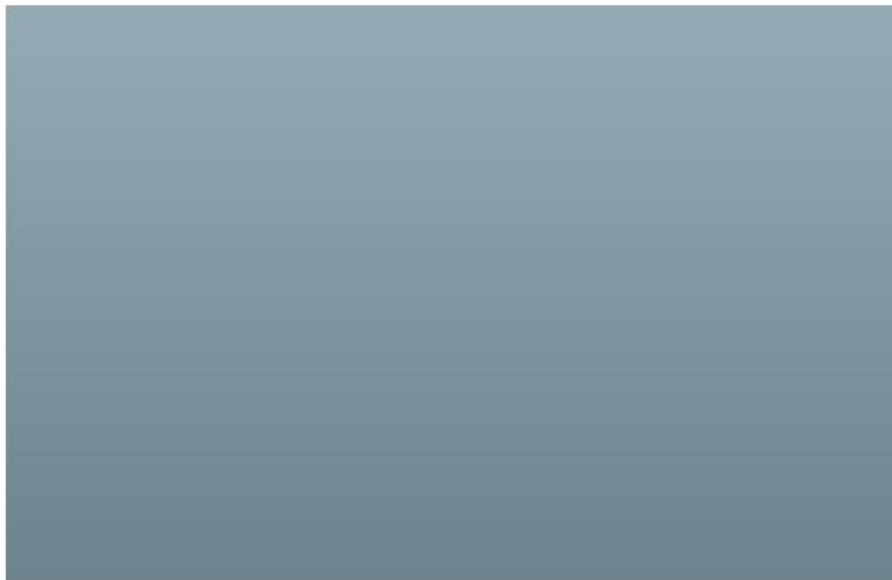

아래 그림과 같이 신규 프로젝트가 생성되고 Main Application mxml파일이 열리면 성공적으로 Flex Project가 생성된 것이다.




Web Application Project의 {Web Root folder}/flex폴더에 Flex Project가 컴파일 되어 아래와 같은 파일들이 생성된 것을 확인 할 수 있다. Flex Application의 Main Application mxml(main.mxml) 파일명(main)이 Flex Application 실행 html 파일명(main.html)이 된다.




WAS를 시작해 Flex Application실행 html을 브라우저에서 호출한다.(http://localhost:8080/myproject/flex/main.html) 아래와 같이 화면이 출력 됐다면 Flex Project가 정상적으로 설치 된 것이다.



# 2.5.Remoting Service Call

Flex Applicatin Main mxml에서 Spring Bean(genreService)의 getList메소드를 호출한다. 아래와 같이 RemotingObject태그를 사용해 destination(Spring Bean)과 호출할 method를 정의한다.

```
<mx:RemoteObject id="remotingService" destination="genreService">
    <mx:method name="getList" result="resultHandler(event)"/>
</mx:RemoteObject>
```

genreService getList의 호출 결과는 resultHandler메소드에서 처리한다. 아래는 genreService getList메소드를 호출한 결과를 DataGrid에 바인딩하는 예이다.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
 initialize="getList()">
    <mx:Script>
            import mx.rpc.events.ResultEvent;
            import mx.collections.ArrayCollection;

            [Bindable]
            private var genreList:ArrayCollection;
```

```
            private function getList():void{
                remotingService.getList();
            }

            private function resultHandler(event:ResultEvent):void{
                genreList = event.result as ArrayCollection;
            }
    </mx:Script>

    <mx:RemoteObject id="remotingService" destination="genreService">
        <mx:method name="getList" result="resultHandler(event)"/>
    </mx:RemoteObject>
   <mx:Array id="columnsInfo">
  <mx:DataGridColumn dataField="genreId" headerText="GENRE ID" editable="true"/>
  <mx:DataGridColumn dataField="name" headerText="NAME" editable="true"/>
 </mx:Array>

 <mx:DataGrid id="grdGenre" dataProvider="{genreList}" columns="{columnsInfo}"/>
</mx:Application>
```

위와 같이 작성 한 후 main.html을 호출 하면 다음과 같은 화면을 볼 수 있다.

# 3.Spring BlazeDS Integration 환경 설정

Spring BlazeDS Integration은 Flex를 이용해 Web Application을 개발 할 경우, 서버측과 RPC방식으로 통신할 때 필요한 BlazeDS를 Spring Framework와 연계하여 편리하게 사용할 수 있도록 한다. 복잡한 환경 설정이 flex namespace를 통해 간편해 졌고 Spring Bean을 Exporting하기 위한 annotation이 제공된다.

Spring BlazeDS Integration을 사용하기 위한 환경은 다음과 같다.

- Java 5 이상
- Spring 2.5.6 이상
- Adobe BlazeDS 3.2 이상

본 메뉴얼을 Spring 3.0.5, Adobe BlazeDS 4, Spring Flex Integration 1.5.0 기준으로 작성되었다.

## 3.1.Spring BlazeDS MessageBroker 환경 설정

### 3.1.1.Spring DispatcherServlet

BlazeDS 에서는 MessageBrokerServlet 을 front Controller 로 사용하여 Flex client 의 요청을 처리했다. Spring BlazeDS Integration 을 사용하면 Spring MVC 의 DispatcherServlet 이 Flex client 의 요청을 처리한다. 따라서 web.xml 에 아래와 같은 설정을 추가한다.

```
<servlet>
    <servlet-name>SpringBlazeDS</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:/springmvc/flex-servlet.xml</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>SpringBlazeDS</servlet-name>
    <url-pattern>/messagebroker/*</url-pattern>
</servlet-mapping>
```

### 3.1.2.MessageBroker

Spring BlazeDS Integration에서는 BlazeDS와 Spring과의 연계를 위한 복잡한 환경 설정을 간단히 하기 위해 flex namespace를 제공한다. flex namespace를 사용하기 위해서는 Spring 설정 파일에 다음과 같이 xsd를 정의를 추가한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:flex="http://www.springframework.org/schema/flex"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
```

```
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/flex
        http://www.springframework.org/schema/flex/spring-flex-1.5.xsd">

        ...
</bean>
```

web.xml 파일의 설정에 의해 flex client 의 요청을 DispatcherServlet 이 처리하므로 그 요청을 MessageBroker로 위임해야 한다.

MessageBroker 설정은 flex namespace 추가로 아래와 같이 간단한 설정만으로 가능하다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:flex="http://www.springframework.org/schema/flex"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/flex
        http://www.springframework.org/schema/flex/spring-flex-1.5.xsd">

    <flex:message-broker />
</beans>
```

위의 설정의 경우 기본적으로 /WEB-INF/flex/services-config.xml 파일을 참조하게 된다. 설정 파일의 위치가 다를 경우 services-config-path attribute 설정을 통해 변경이 가능하다.

```

<flex:message-broker services-config-path="classpath*:services-config.xml" />
```

flex namespace를 사용하지 않을 때에는 아래와 같이 설정한다.

```
<bean id="messageBroker"
          class="org.springframework.flex.core.MessageBrokerFactoryBean">
    <property name=""serviceConfigPath" value="classpath*:services-config.xml"/>
</bean>
```

위 Configuring the Spring DispatcherServlet에서 처럼 /messagebroker/*요청에 대한 servlet-mapping을 별도로 설정하지 않고 모든 요청에 대해서 DispatcherServlet이 처리 할 경우 /messagebroker/*에 대한 요청을 MessageBroker가 처리 하기 위해서는 아래와 같이 SimpleUrlHandlerMapping을 설정한다.

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <value>
        /messagebroker/*=_messageBroker
        </value>
    </property>
</bean>

<bean class="org.springframework.flex.servlet.MessageBrokerHandlerAdapter"/>
```

위의 SimpleUrlHandlerMapping설정은 flex namespace를 사용할 경우는 아래와 같다.

```
<flex:message-broker>
    <flex:mapping pattern="/messagebroker/*" />
</flex:message-broker>
```

# 3.2.Exporting Spring Beans

## 3.2.1.RemotingService 환경 설정

Application 레벨의 채널 세팅은 BlazeDS설정 파일인 services-config.xml파일에 services태그 안에 아래와 같이 정의 한다. Flex Client에서 RemotingService에 대해 별도의 채널 설정이 필요없을 경우 my-amf 채널을 이용한다.

```
<services>
    ..중략
    <default-channels>
        <channel ref="my-amf"/>
    </default-channels>
</services>
```

RemotingService에 대해 특정 채널을 이용하고자 할 경우에는 아래와 같이 remoting-service의 default-channels속성에 채널 이름을 등록한다.

```
<flex:message-broker>
    <flex:message-service
        default-channels="my-amf,my-streaming-amf,my-longpolling-amf,my-polling-amf"/>
</flex:message-broker>
```

## 3.2.2.remoting-destination 태그

remoting-destination 태그는 Spring Bean 을 Remote 객체로 노출 시킨다.

```
<bean id="productService" class="flex.samples.product.ProductServiceImpl" />
<flex:remoting-destination ref="productService" />
```

다음과 같은 설정도 가능 하다.

```
<bean id="productService" class="flex.samples.product.ProductServiceImpl">
    <flex:remoting-destination />
</bean>
```

include-methods / exclude-methods 속성을 이용해 method 단위의 제어도 가능하다. include-method / exclude-method를 정의하지 않을 경우 default는 include-method이다.

```
<flex:remoting-destination ref="productService"
    include-methods="read, update"
    exclude-methods="create, delete"
    channels="my-amf, my-secure-amf" />
```

## 3.2.3.@RemotingDestination

@RemotingDestination 을 사용해서도 Spring Bean 을 Remote 객체로 노출이 가능하다.

```
@Service("productService")
@RemotingDestination
```

```
public class ProductServiceImpl implements ProductService {
..중략
```

@RemotingInclude, @RemotingExclude annotation을 이용해 method별 노출 여부를 설정할 수 있다.

```
@RemotingInclude
public Category getCategoryList(SearchVO searchVO) throws Exception{
..
}

@RemotingExclude
public void removeCategory(Category category) throws Exception{
...
}
```

@RemotingDestination annotaion을 이용해 Spring Bean을 노출 할 때 Destination 채널 설정은 아래와 같이 한다.

```
@Service("categoryService")
@RemotingDestination(channels={"my-amf","my-secure-amf"})
public class CategoryServiceImpl implements CategoryService {
```

# 4.Flex의 Data 연동

## 4.1.Flex 기본 Data 연동

Flex Client와 Server측과의 데이터 통신 방신에는 기본적으로 두 가지 방식이 있다. HTTPService와 WebService이다. 이 두가지 통신 방식은 LCDS나 BlazeDS없이도 실행 가능하다.

- **HTTPService** : HTTPService 방식은 Flex Client에서 Get, Post방식으로 서버측에 데이터를 전송하고 결과 값을 XML로 받는다. 기존의 Web Application을 수정하여 XML로 데이터를 리턴하기만 하면 Flex UI기반으로 쉽게 개발 할 수 있다. 대신 XML의 경우 데이터량이 증가할 경우 데이터 전송 용량이 커지게 되므로 성능에는 문제가 있다.
- **WebService** : WebService 방식은 WebService로 노출되어 있는 URL을 호출하고 결과값을 SOAP형태로 리턴받는다. HTTPService방식과 마찬가지로 데이터량이 증가할 경우 성능에 문제가 있다.

### 4.1.1.HTTPService

Flex MXML파일에서 HTTPService를 사용하기 위해서는 <mx:HTTPService>를 이용하면 된다. 다음은 HTTPService를 이용해 사용자 인증을 하는 샘플 소스이다.

```
<mx:HTTPService id="categoryService" method="post"
    url="../categoryList.do" result="getCtegoryListResult(event)" useProxy="false">
    <mx:request>
        <search_condition>{searchCondition.text}</search_condition>
        <search_keyword>{searchKeyword.text}</search_keyword>
    </mx:request>
</mx:HTTPService>
...중략

private function getCategoryList():void
{
    ...중략
    categoryService.send();
}
```

위의 소스의 경우 getCategoryList메소드가 호출되면 HTTPService에 설정된 url이 호출되고 request태그 안의 값들을 post방식으로 전달한다. 결과 값은 XML형태로 리턴받고 getCategoryListResult메소드에서 처리한다. HTTPService를 BlazeDS없이 사용할 경우에는 useProxy값을 false로 세팅해야 한다.

### 4.1.2.WebService

WebService를 호출 하기 위한 예는 다음과 같다.

```
<mx:WebService id="categoryService"
    wsdl="http://localhost:8080/anyframe.samples.ria.flex/getCategoryList.cfc?wsdl"
    result="getCategoryListResult(event)" useProxy="false">
    <mx:operation name="getCategoryList">
        <mx:request>
            <search_condition>{searchCondition.text}</search_condition>
          <search_keyword>{searchKeyword.text}</search_keyword>
        </mx:request>
    </mx:operation>
</mx:WebService>
```

categoryService.getCategoryList.send()를 실행하게 되면 지정된 wsdl 값을 이용하여 WebService를 호출하게 되고 결과는 getCategoryListResult함수에서 처리한다.

```
private function getCategoryListResult(event:ResultEvent):void{
    grdCategory.dataProvider = event.result;
}
...중략
<mx:DataGrid id="grdCategory.dataProvider" fontSize="14" width="800"/>
```

# 4.2.BlazeDS를 이용한 Data 연동

BlazeDS를 사용할 때에는 HTTPService, WebService 외에 Proxy 방식의 HTTPService, WebService와 RemotingService, MessageService를 사용할 수 있다.

- **HTTPProxyService** : Flex에서 기본으로 제공하는 HTTPService와 비슷하지만 MXML에 직접 url을 입력하지 않고 BlazeDS 설정 파일에 등록된 Destination을 참고로 url을 호출한다. 호출해야 할 url이 변경되어도 MXML파일을 수정하지 않아도 되고 MXML에 url이 노출되지 않으므로 보안에도 이점이 있다.
- **WebService(Proxy)** : HTTPProxy서비스와 마찬가지로 wsdl의 위치를 BlazeDS 설정 파일에 Destination으로 등록한다.
- **RemotingService** : RemotingService는 WAS에 등록된 자바 객체의 메소드를 호출하는 방법으로 AMF 채널을 이용해 결과값을 직렬화된 객체로 리턴받는 방식이다. SpringBlazeDS Integration을 이용할 경우 Flex Client에서 Spring Bean의 메소드를 호출 할 수 있다.
- **MessageService** : Producer에서 데이터를 제공하면 BlazeDS Application MessageService 큐에 쌓이게 되고 쌓여진 메세지는 Consumer에게 전달된다.

## 4.2.1.HTTPProxyService

Flex에서 기본으로 제공하는 HTTPService와 MXML의 작성방법은 유사하다. 단 useProxy의 속성 값을 true로 해야하고 url대신 destination 속성 값을 정의한다.

```
<mx:HTTPService id="categoryService"
    destination="categoryService" result="getCtegoryListResult(event)" useProxy="true">
    <mx:request>
        <search_condition>{searchCondition.text}</search_condition>
        <search_keyword>{searchKeyword.text}</search_keyword>
 </mx:request>
</mx:HTTPService>
...중략

private function getCategoryList():void
{
    ...중략
    categoryService.send();
}
```

위와 같이 MXML을 작성 했다면 BlazeDS 설정 파일 중 HTTPProxyService에 대한 설정 파일인 proxy-config.xml파일에 다음과 같이 categoryService란 이름의 destination을 등록한다.

```
<destinaion id="categoryService">
    <properties>
        <url>anyframe.samples.ria.flex/getCategoryList.do</url>
    </properties>
</destination>
```

## 4.2.2.RemotingService

RemotingService를 사용하기 위해서는 BlazeDS 설정 파일(remoting-config.xml)에 아래와 같이 destination을 추가한다.

```
<service id="remoting-service"
    class="flex.messaging.services.RemotingService">

    <adapters>
        <adapter-definition id="java-object"
 class="flex.messaging.services.remoting.adapters.JavaAdapter"
         default="true"/>
    </adapters>

    <default-channels>
        <channel ref="my-amf"/>
    </default-channels>

    <destination id="categoryService">
        <properties>
            <source>anyframe.sample.flex.category.CategoryService</source>
        </properties>
    </destination>

</service>
```

RemotingService에 등록된 destination을 MXML에서는 RemoteObject를 이용해 호출한다.

```
<mx:Script>
    import mx.rpc.events.ResultEvent;

    private function getCategoryList():void{
        var searchVO:FlexSearchVO = new FlexSearchVO();
        categoryService.getCategoryList(searchVO);
    }
    private function getCategoryListResultHandler(event:ResultEvent):void{
        grdCategory.dataProvider = event.result;
    }
</mx:Script>
<mx:RemoteObject id="categoryService" destination="categoryService">
    <mx:method name="getCategoryList" result="getCategoryListResultHandler(event)"/>
</mx:RemoteObject>
<mx:DataGrid id="grdCategory"/>
<mx:Button label="getCategoryList" click="getCategoryList()"/>
```

위의 소스코드는 RemoteObject를 이용해 remoting-config.xml에 등록된 destination의 getCategoryList 메소드를 호출하는 예이다.

Spring BlazeDS Integration을 이용하게 될 경우에는 remoting-config.xml파일에 destination을 별도로 등록하지 않고 @RemoteDestination과 같은 annotaion을 사용해 해당 Bean을 직접 destination으로 등록할 수 있다. Spring Bean의 Export방법에 대해서는 Exporting Spring Beans for Flex Remoting을 참고한다.

# 4.3.Domain객체와 ASObject의 Mapping

ActionScript에서 서버측에서 사용되는 Domain객체를 동일하게 사용할 경우에는 [RemoteClass]메타 데이터를 이용해 서버측 Domain객체와 ASObject간의 매핑정보를 설정해야한다.

```
Category.java

package anyframe.samples.flex.domain;
...중략
public class Category {
    private String categoryId;
    private String categoryName;
    private String categoryDesc;
    private String regDate;

    ..중략
    public String getCategoryId() {
        return categoryId;
    }
    public void setCategoryId(String categoryId) {
        this.categoryId = categoryId;
    }
    public String getCategoryName() {
        return categoryName;
    }
    public void setCategoryName(String categoryName) {
    this.categoryName = categoryName;
    }
    public String getCategoryDesc() {
        return categoryDesc;
..중략
```

```
Category.as

package samples.domain
{
 [Bindable]
 [RemoteClass(alias="anyframe.samples.flex.domain.Category")]
 public class Category
 {
  public var categoryId:String;
     public var categoryName:String;
     public var categoryDesc:String;
     public var regDate:String;
 }
}
```

[RemoteClass]메타 데이터를 이용해 Domain객체와 AS객체를 Mapping했을 경우 Client쪽에서 Category.as를 이용해 전달한 값은 Server측에서 Category.java에 해당하는 객체로 받는다. 역으로 Server측에서 Category.java에 해당하는 객체로 값을 리턴할 경우 Client에서는 Category.as에 해당하는 객체로 전달받는다.

# 5.FlexService

FlexService는 Flex를 사용하여 Web Application을 개발 할 때 DataSet 기반의 Service 인터페이스/구현클래스, Dao 인터페이스/구현클래스 등 기본 CRUD 메소드 기능이 모두 구현된 클래스를 직접 이용하거나 상속받아서 사용할 수 있는 기능을 제공하고 있다.

FlexService를 이용하여 개발 시 다음과 같은 특징을 갖는다.

- DataSet는 여러개의 DataRow로 구성되어 있고 DataRow는 ObjectProxy객체를 상속 받아 구현 되어있으므로 Flex UI에서 데이터 값이 변경었을 때 별도의 이벤트 처리를 해주지 않아도 된다.
- 개발자가 Business Layer, Data Access Layer 코드를 작성하지 않고 FlexService에서 제공하는 Service 클래스와 Dao 클래스들을 그대로 재사용하여 기본 CRUD 기능을 구현할 수 있다.
- 기본 CRUD 외의 부가 기능이 필요한 경우 FlexService에서 제공하는 클래스를 상속받아서 부가 기능에 대해서만 추가 기능을 구현할 수 있다.

다음은 FlexService 사용 방법이다.

- DataSet 설정
- DataService 설정
- Service 클래스 생성
- Dao 클래스 생성

# 5.1.DataSet

FlexService는 DataSet을 이용한다.

## 5.1.1.DataSet

DataSet은 여러개의 DataRow로 구성되어 있고 고유의 id와 DataSet name, 그리고 Query실행에 필요한 id를 갖고 있다. DataRow는 ObjectProxy객체를 상속 받는다. 각 DataRow는 ROWTYPE를 갖고 있어 데이터가 변경 또는 삭제 되었을 경우 이벤트가 발생해 ROWTYPE값이 변경된다.

```
<data:DataSet id="dsMain" dataSetName="mainDataSet"
selectQueryId="findMovieList"
insertQueryId="insertMovie"
updateQueryId="updateMovie"
deleteQueryId="deleteMovie"
useChangeInfo="true"/>
```

# 5.2.DataService

DataService는 RemoteObject를 상속받아 구현 되었다. default Destination은 flexService이고 사용자가 임의로 세팅 할 수 있다. 사용방법은 RemoteObject와 같지만 서버측에 전달하는 객체는 DataSet Array다.

DataService의 선언 방법은 다음과 같다.

```
<data:DataService id="dataService" fault="dataService_faultHandler(event)"/>
```

아래는 DataSet과 DataService를 이용해 조회하는 쿼리를 실행하는 예이다.

```
protected function search():void{
```

```
    var searchParam:Object = new Object;
    searchParam["firstName"] = searchStr.text;
    dataService.getList( ['dsContact'], searchParam );
}
```

# 5.3.Service 클래스 생성

Service 클래스의 경우 FlexService에서 제공하는 기본 CRUD 메소드 이외의 기능을 제공하는 경우나 기본 CRUD 메소드를 확장하여 사용해야 하는 경우 Service 구현 클래스를 생성하여 사용하도록 하고 기본 CRUD 메소드를 그대로 사용하는 경우 Service 구현 클래스를 생성하지 않는다.

## 5.3.1.FlexService

서비스 인터페이스는 FlexService를 사용한다. FlexService에는 신규 생성, 수정, 목록 조회, 삭제에 관한 메소드가 선언되어 있고 수정,삭제,신규생성을 한번에 처리하는 메소드도 선언되어 있다.

```
public interface FlexService {

    public List findList(List dataSetList, Map param) throws Exception ;

    public Map saveAll(List dataSetList, Map param) throws Exception ;

    public Map find(String queryId, DataRow dataRow, Map param )throws Exception;

    public Map create(String queryId, DataRow dataRow, Map param )throws Exception;

    public Map update(String queryId, DataRow dataRow, Map param )throws Exception;

    public Map remove(String queryId, DataRow dataRow, Map param )throws Exception;
}
```

## 5.3.2.FlexServiceImpl

서비스 구현 클래스는 FlexServiceImpl를 사용하며 다음과 같은 구조로 되어 있다.

```
public class FlexServiceImpl implements FlexService{

    private FlexDao flexDao;

    public List findList(List dataSetList, Map param) throws Exception{
        for(int i=0; i < dataSetList.size() ; i++){
            DataSet ds = (DataSet)dataSetList.get(i);

            String queryId = ds.selectQueryId;

            ds.addAll(flexDao.getList(queryId, param));
        }
        return dataSetList;
    }

    public Map saveAll(List dataSetList, Map param) throws Exception {

        Map result = new LinkedHashMap();

        for(int i=0; i<dataSetList.size(); i++) {

            DataSet ds = (DataSet)dataSetList.get(i);
```

```
            for(int cnt=0; cnt<ds.size(); cnt++) {
                DataRow dr = (DataRow)ds.get(cnt);

                if(dr.ROWTYPE.equals("D")) {
                    String queryId = ds.deleteQueryId;
                    int deleteCnt = flexDao.delete(queryId, dr, param);
                    if(result.containsKey(queryId)) {
                        deleteCnt += ((Integer)result.get(queryId)).intValue();
                    }
                    result.put(queryId, deleteCnt);
                }
            }

            for(int cnt=0; cnt<ds.size(); cnt++) {
                DataRow dr = (DataRow)ds.get(cnt);
                if(dr.ROWTYPE.equals("I")) {
                    String queryId = ds.insertQueryId;
                    Object generatedKey = flexDao.create(queryId, dr, param);
                    int insertCnt = 1;
                    if(result.containsKey(queryId)) {
                     insertCnt += ((Integer)result.get(queryId)).intValue();
                    }
                    result.put(queryId, insertCnt);

                    if(generatedKey != null) {
                    if(result.containsKey("generatedKeys")) {
                        List generatedKeys = (List) result.get("generatedKeys");
                        generatedKeys.add(generatedKey);
                    } else {
                        List generatedKeys = new ArrayList();
                        generatedKeys.add(generatedKey);
                        result.put("generatedKeys", generatedKeys);
                    }
                }

                }else if(dr.ROWTYPE.equals("U")) {
                    String queryId = ds.updateQueryId;
                    int updateCnt = flexDao.update(queryId, dr, param);
                    if(result.containsKey(queryId)) {
                 updateCnt += ((Integer)result.get(queryId)).intValue();
                }
                result.put(queryId, updateCnt);
                }
            }
        }
        return result;
 }

..중략
}
```

# 5.4.RemotingService

## 5.4.1.Page 조회

다음은 BlazeDS의 RemotingService를 이용해 BoardService의 getList메소드를 호출하는 예이다.

```
<mx:Script>
```

```
<![CDATA[
...중략
private function getBoardList(currentPage:int, pageClick:Boolean = false):void {

 var searchVO:SearchVO = new SearchVO("Board");
 if ( pageClick ){
  searchVO.searchCondition = condition;
  searchVO.searchKeyword = keyword;
 }else{
  keyword = searchKeyword.text;
  condition = searchCondition.selectedItem.data;
  searchVO.searchKeyword = keyword;
  searchVO.searchCondition = condition;
 }
 searchVO.pageIndex = currentPage;
 boardService.getPagingList(searchVO);
}
private function getBoardListResultHandler(event:ResultEvent):void {

 page = event.result as Page;
 plb.totalCount = page.totalCount;
 plb.fetchSize = page.pagesize;
 plb.numPages = page.pageunit;
 plb.currentPage = page.currentPage;

 boardList = page.objects as ArrayCollection;

 setPageStatus(page);
}
...중략
]]>
</mx:Script>

<mx:RemoteObject id="boardService" destination="boardService" showBusyCursor="true">
    <mx:method name="getPagingList" result="getBoardListResultHandler(event)"
 fault="ResultHandler.faultMessage(event)"/>
    <mx:method name="saveAll" result="saveAllResultHandler(event)"
 fault="ResultHandler.faultMessage(event)"/>
</mx:RemoteObject>
```

boardService.getPagingList(searchVO)가 실행되면 destination으로 등록된 boardService getPagingList 메소드가 호출되고 id가 findBoardList인 query가 실행되어 결과 값은 Page객체로 리턴된다.

다음은 Anyframe의 Page클래스와 매핑되는 Page.as클래스이다.

```
package org.anyframe.pagination
{
    import mx.collections.ArrayCollection;

    [RemoteClass(alias="org.anyframe.pagination.Page")]
    public class Page
 {

    public var objects:ArrayCollection;

    public var currentPage:int;

    public var totalCount:int;

    public var pageunit:int;

    public var pagesize:int;
```

```
    public var maxPage:int;

    public var beginUnitPage:int;

    public var endUnitPage:int;
 }
}
```

# 5.4.2.saveAll메소드

BoardService saveAll메소드는 Flex의 DataGrid를 이용해 Data를 추가, 삭제, 수정 한 후 한번의 Service 호출로 화면에서 작업했던 Data를 DB에 반영할 때 사용한다.

DataGrid에서 수정, 삭제, 추가 이벤트가 발생할 경우 Board객체의 status값을 변경한다.

```
//DataGrid의 Row가 추가 됐을 때 실행되는 메소드
private function addBoard():void{
    var addBoard:Board = new Board();
 addBoard.status = 1;
 addBoard.regDate = Util.getToday();
 addBoard.regId = parentDocument.loginUser.userId;
 grdBoard.dataProvider.addItem(addBoard);
}
//DataGird의 Row가 수정 됐을 때 실행되는 메소드
private function updateBoard(event:DataGridEvent):void{
    if (event.reason == DataGridEventReason.CANCELLED){
        return;
    }

    var updateBoard:Board = grdBoard.itemEditorInstance.data as Board;

    var rowStatus:Number = updateBoard.status;

    if( rowStatus == 0 ){
     updateBoard.status = 2;
    }
}
//DataGrid의 Row가 삭제 되었을 때 실행되는 메소드
//삭제된 ASObject를 deleteBoardList에 임시로 저장한다.
private function deleteBoard():void{
    if(grdBoard.selectedIndex != -1){
        var deleteBoard:Board = grdBoard.selectedItem as Board;
        var rowStatus:int = deleteBoard.status;

        if (rowStatus != 1){
            deleteBoard.status = 3;
            deleteBoardList.addItem(deleteBoard)
        }
        grdBoard.dataProvider.removeItemAt(grdBoard.selectedIndex);
    }
}
```

다음은 boardService의 saveAll메소드를 호출한 후 결과를 처리하는 ActionScript이다.

```
//DataGrid에서 Binding Data중 status값이
// 0이 아닌 것과 deleteBoardList에 저장 된 DataGrid에서 삭제된 Data를
// saveBoardList에 추가해 BoardList의 saveAll메소드를 호출한다.
private function saveBoard():void{
```

```
    var saveBoardList:ArrayCollection = new ArrayCollection();
    for ( var j:int = 0 ; j < deleteBoardList.length ; j ++ ){
        saveBoardList.addItem(deleteBoardList.getItemAt(j));
    }
    for ( var i:int = 0 ; i < boardList.length ; i ++ ){
        var board:Board = boardList.getItemAt(i) as Board;
        if(board.status != 0){
            if(board.communityId == null){
                Alert.show( "글 " + board.title + "의 커뮤니티를 선택 하지 않았습니다.");
                return;
            }else if(board.title == null) {
            Alert.show( "Title을 입력하지 않은 글이 있습니다.");
            }else{
            saveBoardList.addItem(board);
            }
        }
    }
    if ( saveBoardList.length == 0 ){
        Alert.show( "변경된 데이터가 없습니다." );
    }else{
    boardService.saveAll(saveBoardList);
    }
}
// saveAll메소드의 호출 결과는 추가, 수정, 삭제된 Row수가 담긴 Map형태이다.
// saveAllResultHandler에서는 saveAll메소드의 실행결과를 Alert창을 통해 보여준다.
private function saveAllResultHandler(event:ResultEvent):void{
    var resultArray:Array = ["INSERT", "UPDATE", "DELETE"];

    var message:String = "";
    for ( var i:int = 0 ; i < 3 ; i ++ ){
        var count:int = event.result[resultArray[i]];
        message = message + count + " Row가 " + resultArray[i] + "\n";
    }
    message = message + "되었습니다.";

    mx.controls.Alert.show(message);
    deleteBoardList.removeAll();
    getBoardList(1);
}
```