Anyframe Async Support Plugin

Version 1.1.0

I. Introduction 1
1. Configuration for Asychronous Support 2
2. Asynchronous Support in Servlet 3 4
2.1. Asynchronous support Background concepts 4
2.2. Meeting Ajax challenges 4
2.3. Servlet 3 Asynchronous Support Features 6
2.4. Asynchronous Related API 8
2.5. Servlet 3 AsyncContext Example 10
2.6. Resources 12
3. Asynchronous Support in Spritng MVC 3.2 13
3.1. Callable 13
3.2. DeferredResult 14

I.Introduction

Async Support Plugin은 Servlet 3에 새롭게 추가된 비동기적인 처리의 특징과 사용법을 설명하고 그에 기반해 스프링 3.2.2.RELEASE - - Spring Framework 3.2.3.RELEASE API [http://static.springsource.org/spring/docs/3.2.x/javadoc-api/] 에서 추가된 비동기적인 처리에 대한 활용 방법을 가이드하기 위한 샘플 코드와 이 오픈 소스를 활용하는데 필요한 가이드라인으로 구성되어있다.

Installation

Command 창에서 다음과 같이 명령어를 입력하여 Async Support plugin을 설치한다.

```
mvn anyframe:install -Dname=async-support
```

본 플러그인은 JDK 6 이상, TOMCAT 7이상, SPRING 3.2.2RELEASE에서 동작한다. 아래의 설정을 통해서 어플리케이션을 실행하여 설치 확인을 하도록한다.

Dependent Plugins

Plugin Name	Version Range
Core [http://dev.anyframejava.org/docs/anyframe/plugin/core/1.6.0/reference/htmlsingle/core.html]	2.0.0 > * > 1.5.1

1.Configuration for Asychronous Support

- **Anyframe Plugin Config**

  servlet 2.5 기반을 둔 애니프레임 5.5.1 에서 배포되는 본 플러그인을 servlet 3 환경에서 사용하기 위해서 Servlet api가 업그레이드 되었다. 애니프레임 기반으로 프로젝트를 생성하고 async-support 플러그인이 설치된 것을 확인한 pom.xml의 servlet-api.jar 라이브러리의 버전을 확인하다.

  ```
  <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>3.0</version>
        <scope>provided</scope>
  </dependency>
  ```

  servlet 3를 지원하기 위하여 애니프레임 코어 1.5.1 버전은 Spring 3.2.2Release 버전에 기반을 두고 개발되었다. async-support 플러그인을 설치한 후 pom.xml 파일의 프로퍼티 항목을 체크하여 JDK는 1.6으로 수정하고 Spring버전이 3.2.2 이상인지 확인한다

  ```
  <properties>
          <inspection.dir>${user.home}/.anyframe/inspection</inspection.dir>
          <spring.version>3.2.2.RELEASE</spring.version>
          <targetJdk>1.6</targetJdk>
          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      </properties>
  ```

  Spring 버전을 확인하였다면 다음의 라이브러리가 pom.xml에 dependency로 설정되어 있다면 해당하는 의존성이 바로 제거되어 있는지 확인한다.

  ```
  <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-asm</artifactId>
        <version>${spring.version}</version>
  </dependency>
  ```

- **Servlet 3 Async Config**

  일반적인 Servlet 3 개발 환경 혹은 Servlet 3 표준에 기반한 스프링 3.2에서 비동기적인 처리를 사용하기 위해서는 web.xml 파일을 다음과 같이 수정해야한다.

  ```
  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/
  xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
  web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  ```

- **Spring MVC 3.2 Async Config**

  Spring MVC 3.2에서는 DispatcherServlet 이나 Filter 역시 <async-supported>true</async-supported> 속성을 정의해줘야한다. 주의해야할 점은 비동기적인 처리에 관여하는 필터는 반드시 비동기 Dispatcher 타입을 지원하도록 설정이 되어야한다는 것이다. 스프링 프레임워크와 함께 제공되는 모든 필터를 위한 비동기 dispatcher type을 사용하게 하는 것은 안전하지 않다. 왜냐하면 필요가 없다면 모든 필터가 비동기적인 처리에 사용되지 않을 것이기 때문이다.

  ```
  <filter>
  ```

```
        <filter-name>encodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-
class>
        <async-supported>true</async-supported>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>utf-8</param-value>
        </init-param>
</filter>
...
<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:/spring/*-servlet.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
</servlet>
```

위와 같이 web.xml에 정의를 할 수 있고 WebMvcConfigurer 클래스에서 제공하는 configureAsyncSupport 메소드를 통해서 비동기적인 처리를 지원할 수도 있다. 두 옵션 다 비동기적인 처리를 위한 타임아웃을 설정할 수 있는데 임의로 설정을 하지 않는 다면 기본적으로 Servlet Container의 세팅을 따른다(e.g Tomcat에서는 10초)

- Servlet 3.0에 기반한 비동기적인 처리를 지원하는 WAS 혹은 서블릿 컨테이너

Server	Server version	
Apache Tomcat	7.0.x 이상	
Jetty	8.x 이상	
GlassFish	3.0.x 이상	
Weblogic	12.1.x 이상	
Websphere	8.0.x 이상	
Jeus	7.0.x이상	

2.Asynchronous Support in Servlet 3

본 챕터에서는 서블릿 3.0 에 새롭게 추가된 비동기적인 처리 기법들과 그를 이해하기 위하 관련 비동기처리 기술에 대해서 알아보도록 한다.

2.1.Asynchronous support Background concepts

Web 2.0 기술은 브라우저와 같은 웹 브라우저와 웹 서버간의 트래픽을 획기적으로 줄였다. Servlet 3.0 에서는 이러한 새로운 변화에 대응하기 위하여 비 동기적인 처리를 할 수 있는 API들이 추가되었다. Serlvet 3.0에서 다루고 있는 비동기적인 처리방식에 대한 이해를 돕기 위하여 HTTP 커뮤니케이션에 대하여 좀 더 알아보도록 하자.

- **HTTP 1.0 to HTTP 1.1**

  HTTP 1.1 표준의 가장 주요한 개선사항은 persistent connections 이라고 할 수 있다. HTTP 1.0에서는 웹 클라이언트와 서버간의 연결이 단일 요청/응답 사이클 후에 바로 종료되는 반면 HTTP 1.1에서는 이런 단일 연결이 계속 유지가 되고 복수의 요청을 처리하기 위하여 다시 사용된다. 클라이언트는 Persistent connections를 통해 각각의 요청 후에 다시 TCP 연결을 하지 않아도 되기 때문에 커뮤니케이션 지연을 상당히 줄일 수 있게 되었다.

- **Thread per connection**

  HTTP 1.1의 persistent connections 에 기반을 둔 Thread per HTTP connection모델은 벤더들이 채택한 가장 일반적인 해결책 중 하나이다. 이 방법을 사용하면 클라이언트와 서버간의 각각의 HTTP 커넥션은 서버 측의 하나의 스레드와 연결된다. 스레드는 서버에서 관리되는 스레드풀에 할당된다. 일단 연결이 종료가 되면 해당 스레드는 스레드풀로 반환되어 재사용되며 다른 일들을 처리하도록 대기한다. 하드웨어 설정에 따라서 이런 방식의 접근은 대량의 서버-클라이언트 연결을 동시에 처리할 수 있다. 주요 웹 서버들을 통한 테스트에서 얻어낸 수치적인 결과는 HTTP connection의 수와 거의 비슷한 비율로 메모리 소비가 증가한다는 것을 보여준다 그 이유는 스레드는 메모리 사용 관점에서 상대적으로 비싸기 때문이다. 일단 스레드풀의 모든 스레드가 요청에 점유가 된다면 새로운 클라이언트들로부터의 요청은 서버로부터 처리되지 않을 수 있기 때문에 고정된 수의 스레드만 사용할 수 있게 설정된 서버들은 쉽게 스레드 부족 문제를 일으킬 수 있다. 그에 비해 많은 웹사이트들은 유저들로부터의 페이지 요청이 산발적으로 이루어진다.(page by page 모델). 연결 스레드들은 거의 모든 시간을 대기를 하는데 소요하는데 이것은 크나큰 자원의 낭비이다.

- **Thread per Request**

  자바 플랫폼(NIO) 패키지를 위한 JAVA 4의 새로운 I/O API에 추가된 non-blocking I/O 덕분에 (레퍼런스를 제공) 단일 persistent HTTP connection은 더 이상한 하나의 스레드를 지속적으로 점유를 할 필요가 없어졌다. 스레드는 오직 요청이 처리될 때만 할당되며 해당 커넥션은 분리된 스레드를 소비하지 않고 새로운요청을 처리하기 위하여 집중화된 NIO 선택 집합 안에 위치한다. 이것을 Thread per request 모델이라 부르는데 이것은 웹 서버로 하여금 더 많은 사용자 요청을 한정된 숫자의 스레드로 처리할 수 있도록 한다. 동일한 하드웨어 설정에서는 thread per connection 모드에 비하여 월등하게 많은 처리를 할 수 있다. 요즘 인기 있는 Tomcat, Jetty, GlassFish (Grizzly), WebLogic 혹은 WebSphere 등을 포함한 모든 서버들이 JAVA NIO 를 통한 thread per request 모델을 사용한다. 웹 서버가 non blocking I/O를 서블릿 API를 통하여 어플리케이션에 노출이 없이 숨겨진 방법으로 구현함으로써 어플리케이션 개발자는 좀 더 쉽게 개발을 할 수 있게 되었다.

2.2.Meeting Ajax challenges

좀 더 즉각적으로 반응하는 인터페이스와 풍부한 사용자 경험을 제공하기 위하여 많은 웹 어플리케이션들이 Ajax를 사용한다. Ajax 어플리케이션을 사용하는 사용자들은 클라이언트를 통하여page-by-page

모델보다 훨씬 더 빈번하게 웹 서버와 데이터를 주고 받는다 보통의 사용자 요청과는 다르게 한 클라이언트로부터 복수의 Ajax요청이 서버로 요청될 수 있다. 또한 클라이언트에서 수행되는 클라이언트와 스크립트들은 업데이트를 위하여 정규적으로 서버에 데이터를 요청할 수 도 있다. 동시다발적인 요청이 대량의 스레드를 소비하도록 유발하는데 이와 유사한 대부분의 경우에 thread-per-request 모델의 장점을 쉽게 상쇄시킨다.

- Slow running, limited resources

  Slow-running back-end 루틴들은 상황을 더 악화시킨다. 예를 들어 클라이언트가 요청을 했을 때 성능이 저하된 JDBC connection 풀 이나 처리량이 낮은 end- point 웹 서비스에 의하여 블락 될 수 있다. 자원이 사용가능해지기 전까지는 해당 스레드는 오랫동안 요청을 기다리며 정체될 수 있다. 이럴 경우에는 요청을 사용 가능한 리소스를 기다리도록 특정 큐에 담아두고 해당하는 스레드를 재활용하는 것이 유리하다. 이것은 Slow-running back-end 처리량에 맞춰서 효과적으로 요청 스레드의 수를 조절한다. 또한 요청을 처리하는 중에(해당 요청이 큐에 저장되어 있을 때) 어떤 스레드도 요청을 위해 소비되지 않는 어떤 시점을 제공한다. 서블릿 3.0의 비 동기적인 처리는 일반적이고 쉬운 접근을 통해서 이런 시나리오를 얻을 수 있도록 디자인 되었다.

- Server Push

  서블릿 3.0의 비 동기적인 처리 특징에 관한 좀 더 흥미롭고 필수적인 사용의 예는 Server push이다, Gmail 유저들이 온라인으로 채팅을 할 수 있는 Google Chat이 그것이다. Google Chat은 사용자에게 보여줄 새로운 메시지가 있다고 해도 그것을 확인하기 위하여 빈번히 서버를 체크하지 않는다. 대신에 서버가 새로운 메시지를 보내주기를 기다린다. 이런 방식의 접근은 2가지 큰 장점이 있다. 서버로 요청을 보내지 않음으로써 커뮤니케이션에 속도의 저하가 없으며 또한 네트워크 대역폭을 낭비하지 않음으로써 서버의 자원을 확보하는 것이다.

  Ajax는 한 사용자로부터의 동시에 다수의 요청을 받는다 하더라도 사용자로 하여금 웹 어플리케이션과 즉각적으로 상호작용할 수 있도록 한다. 대표적인 예가 브라우저가 별도로 유저를 방해하지 않고(입력을 받지 않고 혹은 인터럽트 하지 않고) 정기적으로 서버로부터 어떤 상태가 업데이트 됐는지를 확인하는 것이다. 하지만 이러한 루틴을 통해 높은 빈도로 서버에 요청을 보낸다면 서버의 자원과 네트워크 대역폭을 낭비하게 될 것이다. 반대로 서버가 능동적으로 데이터를 브라우저로 보낼 수 있다면, 다시 말해서, 어떤 이벤트가 발생했을 때 (어떤 리소스의 상태가 변했을 때) 클라이언트로 비 동기적인 메시지를 보낼 수 있다면 Ajax 어플리케이션은 서버와 네트워크 자원의 소모를 줄이고 좀 더 효율적으로 동작할 수 있을 것이다.

  HTTP 프로토콜은 요청/응답 프로토콜이다. 클라이언트는 서버로 요청 메시지를 보내고 서버는 응답 메시지로 응답을 한다.서버는 클라이언트와의 연결을 초기화하거나 클라이언트가 예상치 않은 메시지를 보낼 수가 없다. 이런 HTTP 프로토콜의 이런 측면은 외관상으로는 Server push를 구현하는 것이 불가능해 보일지도 모른다. 하지만 몇 가지 독창적인 기술들이 이런 제약사항들을 피하기 위하여 고안되었다

  - Service Streaming(Streaming)

    Service Streaming(streaming)을 통하여 서버는 어떤 이벤트가 발생했을 때 클라이언트로부터의 명시적인 요청이 없어도 클라이언트로 메시지를 보낼 수 있다. 실제 구현상에서는 클라이언트가 요청을 통해서 서버와의 연결을 초기화 하고 서버 측에서 이벤트가 발생할 때 마다 응답을 받는다. 이론적으로는 응답은 끊어지지 않고 계속 지속이 된다. 해당 응답은 클라이언트 쪽의 자바 스크립트에 의해서 해석이 되어지며 브라우저의 점진적인 렌더링 기능(HTML5등) 에 의하여 표시가 된다

  - Long Polling

    비 동기적인polling 이라고도 불린다, Long polling 은 Server push와 클라이언트 polling을 결합한 일종의 하이브리드 기법이라고 할 수 있다. Long polling은 topic-based publish-subscribe 정책에 사용하는 Bayeux 프로토콜에 기반을 둔다. 클라이언트는 서버에 있는 채널에 요청을 보냄으로써 서버와의 연결을 구독한다(subscribe). 서버는 요청을 잡고 있다가 이벤트가 발생할 때까지 기다린다. 이벤트가 발생하면(혹은 미리 정의해둔 타임아웃이 발생하면) 처리가 완료된 응답 메세지를 클라이언트로 보낸다. 응답을 받은 클라이언트는 즉시 새로운 요청을 보낸다. 그러면 서버는 거의 대

부분의 서버 측 이벤트에 대한 응답을 통해 데이터를 전달하는데 사용할 수 있는 효과적인 요청을 갖게 된다. Long polling 은 Streaming에 비해서 비교적으로 브라우저 상에서 구현이 쉽다

- Passive piggyback

서버가 클라이언트로 보내야 할 업데이트가 있을 때 브라우저로부터 새로운 요청이 올 때까지 기다렸다가 브라우저가 기대하는 응답과 합께 업데이트를 보내는 기법이다

Comet 혹은 Reverse Ajax라고도 알려져 있는 Streaming과Long polling은 Ajax로 구현되었다.(어떤 개발자들은 정기적 Regular polling, Comet 그리고 Piggyback을 포함한 상호작용형 기술들을 reverse Ajax라고 부른다.)

2.3.Servlet 3 Asynchronous Support Features

서블릿 비 동기적인 처리를 하기 위해 다음과 같은 3가지의 방법으로 서블릿이나 서블릿 필터를 정의할 수 있다

- **Annotations**

방법	예시
Annotation	@WebServlet(aSyncSupported=true)
Programming	servletRegistration.setAsyncSupported(true);
Web.xml	<async-supported>true</async-supported>

Annotation Attributes

서블릿 3.0 에서는 웹 어플리케이션 안에서 서블릿을 설정하기 위한 배포서술자의 대안으로 @WebServlet과 @WebFilter라는 두 가지의 새로운 어노테이션을 제공한다. 두 어노테이션 모두 ‘asyncSupported’라는 속성을 포함한다. asyncSupported’를 true로 설정하면 서블릿이나 서블릿 필터는 비 동기적인 처리를 지원하게 된다. asyncSupported 속성은 비 동기적인 컨텍스트에서 위해 만들어진 코드를 동기적인 처리를 하기 위하여 만들어진 코드로부터 구분을 하기 위하여 반드시 필요하다. 비 동기적인 처리를 사용하는 어플리케이션에서는 요청을 처리하는 일련의 과정에서 반드시 배포서술자 혹은 어노테이션을 통해서 이 속성을 정의해야 한다

아래의 코드는 어노테이션을 통해서 서블릿을 선언하고 해당 서블릿이 비 동기적으로 처리될 수 있도록 한 예제이다.

```
@WebServlet( urlPatterns = "*.do", asyncSupported = true)
public class AnyframeServlet extends HttpServlet {
 private static final long serialVersionUID = 1L;
 public AnyframeServlet() {
  super();
  System.out.println("Servlet has been constructed"); }
 protected void doGet(HttpServletRequest request, HttpServletResponse  response) throws
 ServletException, IOException {
  System.out.println("Servlet started");
  final AsyncContext aCtx = request.startAsync(request, response);
  System.out.println("Asyncronous support has been started");
  aCtx.setTimeout(5000l);
  aCtx.addListener(new AnyframeAsyncListener());
  aCtx.start(new Runnable() {
   @Override
   public void run() {
    System.out.println("Another Thread starts...");
    try {
```

```
      Thread.sleep(5000l);

    } catch (InterruptedException e) {
     e.printStackTrace();
    }
    aCtx.dispatch("/HelloWorld/AsyncHelloWorld.jsp");
    System.out.println("Another thread is done.");
   }
  });
  System.out.println("Servlet closed");
 }
 protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
 ServletException, IOException {
 }
}
```

AsyncSupported 속성이true로 설정이 되어있을 때는 doGet메소드가 종료가 되더라도 서블릿이 응답을 보내지 않는다. startAsync() 메소드를 호출하면 응답/요청 객체를 가지고 있는 AsyncContext 객체를 반환하는데 해당 AsyncContext 객체는 application-scoped 큐에 저장이 된다. 요청이 오는 즉시 doGet 메소드는 결과값을 반환하고 원래의 요청 스레드는 반환된다. 다시 말하자면 서블릿은 또 사용자로부터 또 다른 요청을 받을 준비가 되었다는 것이다. 어플리케이션 런치가 큐를 모니터링하고 요청 처리를 재개하는 동안에 분리된 ServletContextListener 객체 안에서 분리된 스레드가 초기화가 된다. 요청이 일단 처리되고 나면 사용자는 호출된 서블릿이 response객체의 getWriter() 메소드로 결과값을 출력하거나 complete() 메소드로 응답을 커밋할 수 있다.위의 예제 코드에서는 분리된 스레드가 5초동안 대기하다가 결과 페이지로 포워딩을 해준다. 기본적으로 JSP 페이지는 asyncSupported 속성이 false가 디폴트로 설정된 서블릿이라는 것을 알아두자.

다음의 예제 코드는 위에서 제공한 서블릿을 테스트하기 위한ServletContainerInitializer 예제이다

```
public class ServletContainerInitializerImpl implements
  ServletContainerInitializer {

 @Override
 public void onStartup(Set<Class<?>> setOfClassesinterestedIn,
   ServletContext context) throws ServletException {
  // going to add a context attribute to show the set of classes that were
  // passed in
  if (setOfClassesinterestedIn != null) {
   context.setAttribute("SET_OF_SERVLETS_IN_APP",
     setOfClassesinterestedIn);
  } else {
   context.setAttribute("SET_OF_SERVLETS_IN_APP", null);
  }
  // Add a ServletContextListener programmatically
  context.addListener(anyframe.core.listener.AnyframeListener.class);
  // Add a Filter programmatically
  // if this jar is used as a shared library, then this filter will be
  // applied to all requests
  FilterRegistration.Dynamic dynamic = context.addFilter("AnyframeServletExample",
(anyframe.core.filter.AnyframeFilterExample.class));
  dynamic.addMappingForUrlPatterns(EnumSet.allOf(DispatcherType.class),true, "/*");

 }
}
```

소스에서 볼 수 있듯이 서블릿 3.0은 프로그래밍적인 방법으로 listener 혹은 filter를 등록할 수 있는 메소드를 제공한다. addListener() 메소드와 addFIlter() 메소드가 그것이다. startSync() 메소드가 요청에 의해 호출된 후에는(비동기 모드로 들어간 다음에는) 비 동기적인 처리가 끝나거나 타임아웃 되자마자 ServletContextEvent 가 등록된 리스너로 보내어진다. 해당 컨텍스트 이벤트에는 AsyncContext 객체에 있던 것과 같은 요청/응답 객체가 들어있다

AsyncListener Interface

AsyncListener 인터페이스는 비 동기 이벤트가 발생했을 때 호출되는 다음과 같은 메소드을 정의한다.

- **Annotations**

메소드명	설명
public void OnStartAsync(AsyncEvent event)	비 동기적인 처리가 시작될 때 호출된다
public void OnComplete(AsyncEvent event)	비 동기적인 처리가 종료될 때 호출된다
public void OnError(AsyncEvent event)	비 동기적인 처리가 실패했을 때 호출된다
public void OnTimeout(AsyncEvent evet)	비 동기적인 처리 중에 타임아웃이 발생했을 때 호출된다

아래의 예제코드는 서블릿 3.0에 추가된 AsyncListener 인터페이스를 구현한 예제코드이다. AsyncListener 클래스에는 @WebListener 어노테이션을 붙이지 않기 때문에 이벤트를 가져오기 위해서는 직접 AsyncContext에 AsyncListener를 등록해야 한다. 예제로 제공된 AnyframeServlet.java 에서 사용한 것처럼AsyncContext 클래스에서 제공하는 addLIstener () 메소드를 통해서 추가하면 된다.

```
@WebListener
public class AnyframeAsyncListener implements AsyncListener {
 @Override
 public void onComplete(AsyncEvent arg0) throws IOException {
  System.out.println("onComplete");
 }
 @Override
 public void onError(AsyncEvent arg0) throws IOException {
  System.out.println("onError");
 }
 @Override
 public void onStartAsync(AsyncEvent arg0) throws IOException {
  System.out.println("onStartSync");
 }
 @Override
 public void onTimeout(AsyncEvent arg0) throws IOException {
  System.out.println("onTimeOut");
 }
}
```

참고

어떤 어플리케이션이 비 동기적인 처리를 시작하려고 하는데 비 동기적인 처리를 지원하지 않는 서블릿이나 서블릿 필터가 요청 처리 과정에 포함이 되어 있다면 'IllegalStateException'이 발생할 것이다. 다시 말하자면 서블릿이 비 동기방식으로 선언되어있고 서블릿에 의해서 비 동기적인 처리를 하려고 하면 서블릿 필터도 역시 비 동기방식으로 선언이 해야 한다는 것이다

2.4.Asynchronous Related API

본 챕터에서는 서블릿 3.0 에 새롭게 추가된 비동기처리에 관련된 API에 대한 설명을 제공한다.

- **Servlet Request Methods**

  서블릿 3.0에서는 비 동기적인 처리를 위하여 다음과 같은 서블릿 요청 메소드들을 제공한다.

클래스명	메소드명	설명
javax.servlet.ServletRequest		

클래스명	메소드명	설명
	startAsync(servletRequest, servletResponse)	명시적으로 요청과 응답 객체를 처리한다
javax.servlet.ServletRequest	startAsync()	묵시적으로 요청과 응답 객체를 처리한다-> 필터로 랩핑시 주의가 필요
javax.servlet.ServletRequest	isAsyncSupported()	비 동기적인 처리를 지원하는 요청인지 확인한다
javax.servlet.ServletRequest	isAsyncStarted()	비 동기적인 처리가 시작됐는지 확인한다.
javax.servlet.ServletRequest	getDispatcherType()	해당 요청의 dispatcher 타입을 반환한다

비 동기적인 처리를 위하여 요청 처리 체인에서 사용자가 asyncSupported 속성을 설정하고 나면 사용자는 “startAsync(servletRequest, servletResponse) 혹은 startAsync() 를 호출하여 비 동기적인 요청을 생성할 수 있다. 이 두 메소드들의 차이점은 startASync가 묵시적으로 원래의 요청과 응답을 사용하는 반면에 startAsync(servletRequest, servletResponse) 메소드는 호출을 할 때 전달된 응답과 요청 객체를 사용한다는 점이다.(위의 36.3에서 제공하고 있는 서블릿 예제에서는 명시적으로 응답과 요청객체를 전달해서 사용하고 있다.) 그래서 해당 요청 처리 과정에서 해당하는 요청과 응답을 다른 서블릿이나 필터로 감쌀 수(Wrapping) 있다. startAsync 메소드는 AsyncContext 객체를 반환한다는 점에 주목하자. 해당 AsyncContext 객체는 사용되는 메소드에 따른 요청과 응답과 함께 적절히 초기화가 된다. 사용자는 전달인자가 없는 startAsync 메소드를 호출하고 결과를 필터로 랩핑할 때 주의를 기울여야 한다. 어떤 데이터가 어떤 필터에 의해 감싸진 응답에 들어있고 그것이 숨겨진 응답의 흐름에 들어가지 않는다면 사용자는 해당 데이터를 잃을 수 있다. 또한 ServletRequest 클래스에는 요청이 비동기인지 확인할 수 있는 메소드를 제공함으로써 사용자의 편의를 돕는다.

- **AsyncContext API**

  서블릿 3.0에서는 다음과 같이 조금씩 다르게 작동하는 몇가지의 dispatch 메소드와 다른 API를 제공한다

클래스명	메소드명	설명
javax.servlet.AsyncContext	dispatch()	원래 요청을 받았던 URL 요청을 되돌려 보낸다.(원래의 서블릿 실행)
javax.servlet.AsyncContext	dispatch(String path)	입력 받은 경로로 요청을 보낸다.(다른 서블릿 실행)
javax.servlet.AsyncContext	dispatch(ServletContext context, String path)	입력 받은 컨텍스트와 연관된 다른 경로로 요청을 보낸다(서블릿을 실행)
javax.servlet.AsyncContext	complete()	해당 AsyncContext object를 초기화 한 응답을 종료시킨다
javax.servlet.AsyncContext	setTimeOut(long timeout)	비동기 처리를 위한 타임아웃 설정 메소드
javax.servlet.AsyncContext	addListener()	타임아웃 혹은 에러 종료를 알리기 위한 리스너를 등록할 수 있다.

dispatch 매소드는 요청 처리를 계속하기 위하여 호출될 수 있으며 응답을 처리한다. 사용자는 비 동기적인 runnable 스레드들 중 하나로부터 disptach를 호출하거나 또 사전에 startAsync메소드가 호출된 경우에도(startAsync메소드를 통하여 비 동기처리 모드로 들어왔을 때) dispatch를 호출할 수 있다. 만약 dispatch가 그 이전의 서블릿에서 호출이 되었다면 해당 dispatch는 이전의 서블릿이 해당 메소

드를 종료하기 전까지는 유효하지 않다. 보통 응답은 서블릿이나 필터를 통해서 생성이 된다. 비 동기적인 처리를 사용하는 장점은 계속적으로 처리를 하기 위하여 컨테이너로 스레드를 반환하는 것이다. 기본적으로 비 동기적인 처리는 WorkManager 스레드 풀에서 이루어진다. AsyncContext.complete() 메소드는 해당 AsyncContext object를 초기화 하기 위하여 사용된 요청으로부터 시작된 비 동기적인 처리를 종료하는데 사용된다. 이 메소드는 해당 AsyncContext object를 초기화 한 응답을 종료시킨다. 예를 들면 사용자는 비동기적인 처리에 의해 생성된 응답이 끝내기 위하여 complete 메소드르 호출 할 수 있다. 어플리케이션이 차후에 요청에 대한 startAsync 호출이 없는 forward 메소드를 사용하여 요청을 처리하여 컨테이너로 보낸다면 컨테이너는 암묵적으로 이 메소드를 호출한다.

2.5.Servlet 3 AsyncContext Example

아래의 코드들은 AsyncContext를 이용해 웹페이지가 추가적인 Request 없이 계속해서 서버에서 데이터를 받을 수 있도록 구현된 채팅 예제이다. 이 예제의 기본적인 처리방식은 생성된 AsyncContext를 complete()로 처리하지 않고 계속 관리하면서 서버에서 데이터를 전달하는 것이다.

```
@WebServlet(urlPatterns = "/enterServlet", asyncSupported = true)
public class EnterServlet extends HttpServlet {

 @Override
 protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
 ServletException, IOException {
  res.setContentType("text/html; charset=UTF-8");
  res.setHeader("Cache-Control", "private");
  res.setHeader("Pragma", "no-cache");
  res.setCharacterEncoding("UTF-8");
  PrintWriter writer = res.getWriter();
  // for IE
  writer.println("<!-- Comet is a programming technique that enables web servers to send
 data to the client without having any need for the client to request it. -->\n");
  writer.flush();
  AsyncContext asyncCtx = req.startAsync();
  addToChatRoom(asyncCtx);
  return;
 }

 private void addToChatRoom(AsyncContext asyncCtx) {
  asyncCtx.setTimeout(0);
  ChatRoom.getInstance().enter(asyncCtx);
 }

}
```

처음 채팅 화면으로 들어오면 해당 페이지는 EnterServlet을 요청하고, 이 서블릿은 AsyncContext를 생성 후 ChatRoom에 저장한다. 그리고 doGet()에 대한 return을 수행하므로 기본적인 Request에 대한 Response 처리는 끝난 것과 같다. 그러나, 사실 AsyncContext를 통해 채널이 계속 열려 있으므로 서버는 HTTP Streaming 방식으로 클라이언트인 웹브라우저에 계속 데이터를 전달할 수 있다. 아래 예제 코드에서, 하나의 클라이언트가 메시지를 보낼 때마다 ChatRoom에 있는 클라이언트들에게 메시지를 전달하는 방식을 확인할 수 있다.

```
public class ChatRoom {

 private static ChatRoom INSTANCE = new ChatRoom();

 public static ChatRoom getInstance() {
  return INSTANCE;
 }
```

```
private List<AsyncContext> clients = new LinkedList<AsyncContext>();
private BlockingQueue<String> messageQueue = new LinkedBlockingQueue<String>();
private Thread messageHandlerThread;
private boolean running;

private ChatRoom() {
}

public void init() {
 running = true;
 Runnable handler = new Runnable() {
  @Override
  public void run() {
   while (running) {
    try {
     String message = messageQueue.take();
     sendMessageToAllInternal(message);
    } catch (InterruptedException ex) {
     break;
    }
   }
  }
 };
 messageHandlerThread = new Thread(handler);
 messageHandlerThread.start();
}

public void enter(final AsyncContext asyncCtx) {
 clients.add(asyncCtx);
}

public void sendMessageToAll(String message) {
 try {
  messageQueue.put(message);
 } catch (InterruptedException e) {
  e.printStackTrace();
 }
}

private void sendMessageToAllInternal(String message) {
 for (AsyncContext ac : clients) {
  try {
   sendMessageTo(ac, message);
  } catch (IOException e) {
   clients.remove(ac);
  }
 }
}

private void sendMessageTo(AsyncContext ac, String message) throws IOException {
 PrintWriter acWriter = ac.getResponse().getWriter();
 acWriter.println(toJSAppendCoomand(message));
 acWriter.flush();
}

private String toJSAppendCoomand(String message) {
 return "<script type='text/javascript'>\n" + "window.parent.chatapp.append({ message: \""
+ message + "\" });\n" + "</script>\n";
}

public void close() {
 running = false;
```

```
  messageHandlerThread.interrupt();
  for (AsyncContext ac : clients) {
   ac.complete();
  }
 }

}
```

위의 예제에서는 BlockingQueue를 이용하여 클라이언트에서 오는 메시지를 받은 후, 참여한 모든 클라이언트의 AsyncContext에서 Response에 있는 Writer를 이용하여 메시지를 전달하고 있다. 이처럼 AsyncContext는 Long Polling 방식의 처리 뿐만 아니라 HTTP Streaming 방식의 처리를 위해서도 사용될 수 있다. AsyncContext를 직접 다루는 코드는 Spring MVC 3.2에서 제공하는 Callable, WebAsyncTask, DeferredResult에서는 접근할 수 없으므로 상황에 맞추어 선택적으로 사용하여야 한다.

2.6.Resources

- 다운로드

  다음에서 sample 코드를 포함하고 있는 Eclipse 프로젝트 파일을 다운받은 후, 압축을 해제한다. 그리고 hsqldb 폴더 내의 start.cmd (or start.sh) 파일을 실행시켜 테스트 DB를 시작시켜 놓는다

  - Eclipse 기반 실행

    Eclipse에서 압축 해제 프로젝트를 import한 후, 해당 프로젝트에 대해 마우스 오른쪽 버튼을 클릭하고 컨텍스트 메뉴에서 Maven > Enable Dependency Management를 선택하여 컴파일 에러를 해결한다. 그리고 해당 프로젝트에 대해 마우스 오른쪽 버튼을 클릭한 후, 컨텍스트 메뉴에서 Run As > Run on Server (Tomcat 기반)를 클릭한다. Tomcat Server가 정상적으로 시작되었으면 브라우저를 열고 주소창에 http://localhost:8080/anyframe-sample-servlet-asynccontext/각 페이지 주소를 입력하여 실행 결과를 확인한다.

Name	Download
anyframe-sample-servlet-asynccontext.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/optional/async-support/1.1.0/reference/sample/anyframe-sample-servlet-asynccontext.zip]

3.Asynchronous Support in Spritng MVC 3.2

Spring MVC 3.2는 Servlet 3 기반의 비동기적인 요청 처리를 지원한다. 본 챕터에서는 Spring에 기반을 둔 비동기 요청 처리기법에 대한 몇가지 방법을 알아보고록 한다. 본 플러그인은 Spring API에 기반하여 Asynchronous Support in Servlet 3 에서 소개한 Server push 기술 중 Long polling 기법을 이용하여 게시판에 새로운 글이 등록되었을 때 각 클라이언트에게 알림를 보내는 웹 어플리케이션을 구현한 것과 함께 새로운 요청이 들어왔을 때 요청을 처리하기 위한 스레드를 바로 반환하여 요청 스레드를 확보하는 예제를 제공한다.

3.1.Callable

Callable

Spring MVC 3.2 에서는 컨트롤러는 어떤 값을 리턴하는 대신에 java.util.concurrent.Callable 객체를 리턴할 수 있으며 요청을 처리하는 스레드가 아닌 새로운 스레드로 응답할 수 있다. 그 동안에 원래의 Servlet Container 쓰레드는 반환되어 다른 요청들을 처리할 수 있게 된다. Spring MVC는 TaskExecutor의 도움으로 분리된 쓰레드안에 있는 Callable 클래스를 호출하는데 Callable 객체는 리턴이 될 때 그것에 의해 처리된 리턴값과 함께 원래의 요청에 대한 응답을 Servlet container로 보낸다.

아래의 예제는 Callable 클래스를 이용해서 만든 컨트롤러 예제이다. 이 예제는 플러그인과 함께 배포되며 async-support 플러그인을 설치한 뒤에 어플리케이션을 실행하여 'Async-Support 1.0.0 - Callable' 링크를 클릭하면 확인할 수 있다. 웹 어플리케이션을 실행 서버의 로그를 확인하면 스레드가 반환되고 사용되는 것을 실시간으로 확인할 수 있다.

```
....

 @RequestMapping(params = "method=list")
 public Callable<String> list(
   @RequestParam(value = "pageIndex", defaultValue = "1") final int pageIndex,
   final Movie movie, BindingResult result, final Model model) throws Exception {

  //Original thread has been returned to a threadpool.;
  //We can do something does not take a long time here.;

  return new Callable<String>() {
   public String call() throws Exception {

    //We can do someting takes a long time in call method;

    Page resultPage = movieFinder.getPagingList(movie, pageIndex);
    model.addAttribute("movie", movie);
    model.addAttribute("movies", resultPage.getList());
    model.addAttribute("resultPage ", resultPage);
    return "core/moviefinder/movie/list";
   }
  };
 }

....
```

위의 예제의 코드를 보면 좀 더 쉽게 이해가 가능하다. 클라이언트로부터 list 메소드가 호출이 되면 요청을 처리한 스레드는 다시 반환이 되고 새로운 스레드의 call 메소드안에 들어있는 로직들이 새로운 스레드에 의해서 처리가 되어서 Dispatcher servlet으로 응답을 보낸다. 다시 말하면 비 동기처리의 목

적이 요청을 처리시간이 오래 걸리는 작업이나 Remote Service를 호출하는 작업을 Call() 메소드 안에 작성하여 다른 스레드로 하여금 처리하도록 하고 간단한 작업은 요청을 받은 스레드로 처리하고 그것을 반환하여 요청 스레드 풀을 확보하는데 있다고 할 수 있다.

3.2.DeferredResult

비동기적인 요청 처리를 위한 두번째 옵션은 컨트롤러를 통해서 DefferdResult 의 인스턴스를 반환하는 것이다. 이 경우에도 역시 분리된 스레드에 의해서 리턴값이 처리가 된다. 하지만 이 스레드는 Spring MVC에서 관리되는 것이 아니다. 예를 들면 JMS 메세지나 스케쥴링된 작업 혹은 기타등등 같은 외부적인 이벤트에 대한 응답을 처리할때 결과가 만들어질 수 있다

아래의 예제는 DeferredResult 클래스를 이용해서 만든 컨트롤러 예제이다. 이 예제는 플러그인과 함께 배포된다.

```
.....
 @RequestMapping(params = "method=list")
 public DeferredResult<ModelAndView> list(
   @RequestParam(value = "pageIndex", defaultValue = "1") final int pageIndex,
   final Movie movie, BindingResult result, final Model model) throws Exception {

  //Original thread has been returned to a threadpool.;
  //We can do something does not take a long time here.;

  final DeferredResult<ModelAndView> deferredResult = new DeferredResult<ModelAndView>();

  new Thread(new Runnable() {
   @Override
   public void run() {
    //In another thread
    //We can do someting takes a long time in call method;
    Page resultPage = null;
    try {
     resultPage = movieFinder.getPagingList(movie, pageIndex);
    } catch (Exception e) {
     e.printStackTrace();
    }
    ModelAndView modelAndView = new ModelAndView("core/moviefinder/movie/list");
    modelAndView.addObject("movie", movie);
    modelAndView.addObject("movies", resultPage.getList());
    modelAndView.addObject("resultPage", resultPage);
    deferredResult.setResult(modelAndView);
   }
  }).start();

  return deferredResult;
 }
}
```

이 코드 DeferredResult Class 예제는 위의 callable 클래스의 예제와 같은 동작을 하도록 작성하였다. 위의 코드를 잘 보면 요청을 받은 쓰레드가 아닌 또 다른 쓰레드 안에서 데이타베이스를 조회하는 작업 - 일반적으로 긴 시간을 요구하는 - 을 하고 있는데 Callable 클래스와는 다르게 DeferreResult 클래스의 인스턴스는 어떤 요청에 대한 응답을 열어둔채로 가지고 있다가 DeferredResult.setResult() 메소드가 호출되면 DispatcherSerlvet으로 응답을 보낸다. 다시 말하자면 DeferredResult 응답을 보내는 시점을 어플리케이션에서 선택할 수가 있다는 것이다. 이런 특성은 Asynchronous Support in Servlet 3 에서 소개한 Server Push 기술들을 쉽게 구현할 수 있는 바탕이 된다. (위의 예제코드는 일반적인 DeferredResult 클래스의 활용법이 아닌 Callable 클래스와 비교를 위한 코드임을 유의하자.)

아래의 예제는 본 플러그인에서 제공하는 DeferredResult 클래스를 이용해서 만든 Server push 어플리케이션의 일부이다. 본 플러그인의 어플리케이션을 tomcat에서 실행하고 브라우저를 2개 실행한다. 그

리고 2개의 브라우저에서 Callable Class Example을 클릭해서 영화 목록이 출력이 되는 화면으로 이동한다. 해당 화면에서 오른쪽 아래의 Add 버튼을 누르고 form의 형식에 맞게 입력을 하여 새로운 영화를 등록하면 2개의 브라우저 모두에게 새로운 영화가 등록되었다는 알림창이 표시가 된다. (테스트 브라우저는 크롬 버전 27.0.1453.116 m 이다.) 이 예제 역시 플러그인과 함께 배포된다.

```
@RequestMapping("/movieBroadCast.do")
public class MovieBroadCastController {

 Logger logger = LoggerFactory.getLogger(MovieBroadCastController.class);

 private MovieRepository movieRepository;

 @Autowired
 public MovieBroadCastController(MovieRepository movieRepository) {
  this.movieRepository = movieRepository;
 }

 private final Queue<DeferredResult<String>> responseBodyQueue = new
 ConcurrentLinkedQueue<DeferredResult<String>>();

 @RequestMapping(method = RequestMethod.POST)
 public @ResponseBody
 DeferredResult<String> newMoviceNotify() {

  final DeferredResult<String> result = new DeferredResult<String>(10000);

  this.responseBodyQueue.add(result);

  List<String> movies = this.movieRepository.getMovies();

  if (!movies.isEmpty()) {
   broadcastMovieUpdate();
  }

  result.onCompletion(new Runnable() {
   @Override
   public void run() {
    responseBodyQueue.remove(result);
   }
  });
  return result;
 }

 public void broadcastMovieUpdate() {
  logger.debug("broadcast");
  for (DeferredResult<String> result : this.responseBodyQueue) {
   result.setResult("New movie added");
   this.responseBodyQueue.remove(result);
  }
  this.movieRepository.removeMovie();
 }
 ......
}
```

브라우저로부터 /movieBroadCast.do에 대한 요청이 들어오면 새로운 DeferredResult 인스턴스가 생성되어 큐에 저장되며 그것은 비동기적인 요청이 끝날 때, 혹은 요청에 대하여 타임아웃이나 네트워크 에러가 날 때 호출이 되는 onCompletion()메소드에 의해서 큐에서 삭제가 될 것이다. 일단 요청이 들어오면 MovieRepository에 새로운 영화가 들어있는 지를 확인하는데 만약에 새로운 영화가 등록 되어 있다면 해당하는 같은 클래스내의 broadcastMovieUpdate()를 호출한다. broadcastMovieUpdate() 메소드가 호출이 되면 큐에 담겨져 있던 DeferredResult 인스턴스들을 모두 setResult () 메소드를 실행하면서 원

래의 요청들에게 응답을 보낸다. 그리고 나서 큐에 있던 DeferredResult 인스턴스를 삭제한다. 클라이언트(브라우저)상에서는 사용자에게 새로운 영화가 등록됐다는 알림 표시가 뜰 것이다.

```
@RequestMapping(params = "method=create")
 public String create(@Valid Movie movie, BindingResult results,
   SessionStatus status, HttpSession session) throws Exception {

  if (results.hasErrors())
   return "core/moviefinder/movie/form";

  movieService.create(movie);
  status.setComplete();
  this.movieRepository.addMovie("new movie added");
  return "redirect:/asyncSupportMovieFinder.do?method=list";
 }
```

위의 메소드는 사용자가 영화를 새롭게 등록하였을 때 호출되는 메소드로써 새로운 영화를 DB에 생성하고 나서 MovieRepository에 새로운 영화가 등록됐다는 String 인스턴스를 추가한다. 메모리에 새로운 영화가 등록됐을 저장하고 나면 원래의 영화 목록 페이지로 다시 화면이 전환되고 브라우저에서는 /movieBroadCast.do를 호출하게 된다.

참고

플러그인에서 제공하는 Server push 예제는 Long polling 기법의 특성상 브라우저에서 지속적으로 /movieBroadCast.do를 요청하도록 프로그래밍 되었다. JQuery 를 이용하여 사용자가 영화목록 화면으로 진입하면 10초에 한번씩 서버쪽으로 /movieBroadCast.do로 요청을 날리는데 해당하는 요청에 응답이 없으면 반복적으로 재귀적으로 요청을 다시하고 응답이 있으면 화면에 알림을 표시하도록 하였다. 알림을 표시하기 위하여 pnotify 라는 JavaScript notifications 라이브러리가 사용되었다. 상세한 세팅방법과 사용법이 알고 싶다면 Pines Notify [http://http://pinesframework.org/pnotify/] 를 참조하도록 한다.