

Extensibility

In This Topic

- [Extension Consumers](#)
 - [Extension Providers](#)
 - [Publishing and Subscribing to Events](#)
-

XperienCentral extensibility provides a framework for the runtime extension of XperienCentral components like services, elements and panels. This can be done by defining extension points in plugins (called consumers) that are implemented and provided at runtime by other plugins (called providers). The extension points are realized by the delegation of services in the framework to other plugins. These services can be service components like a "find books" service; the delegation of MVC controller calls is also possible within this framework. This delegation of controllers makes it possible to, for example, replace the complete view of plugins in the framework with an alternate view from another plugin, to add a view (new text field) and/or services.

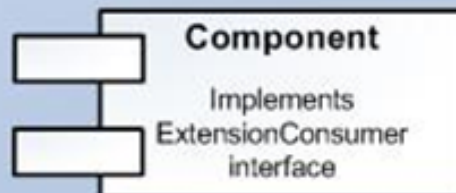
The Extensibility framework consists of a set of framework interfaces like the `Extensible`, `ExtensionPoint` and `ViewExtension` interfaces. By providing an implementation of these interfaces in your plugin and registering these interfaces and extension points in the service framework, plugins can consume and provide extension points.

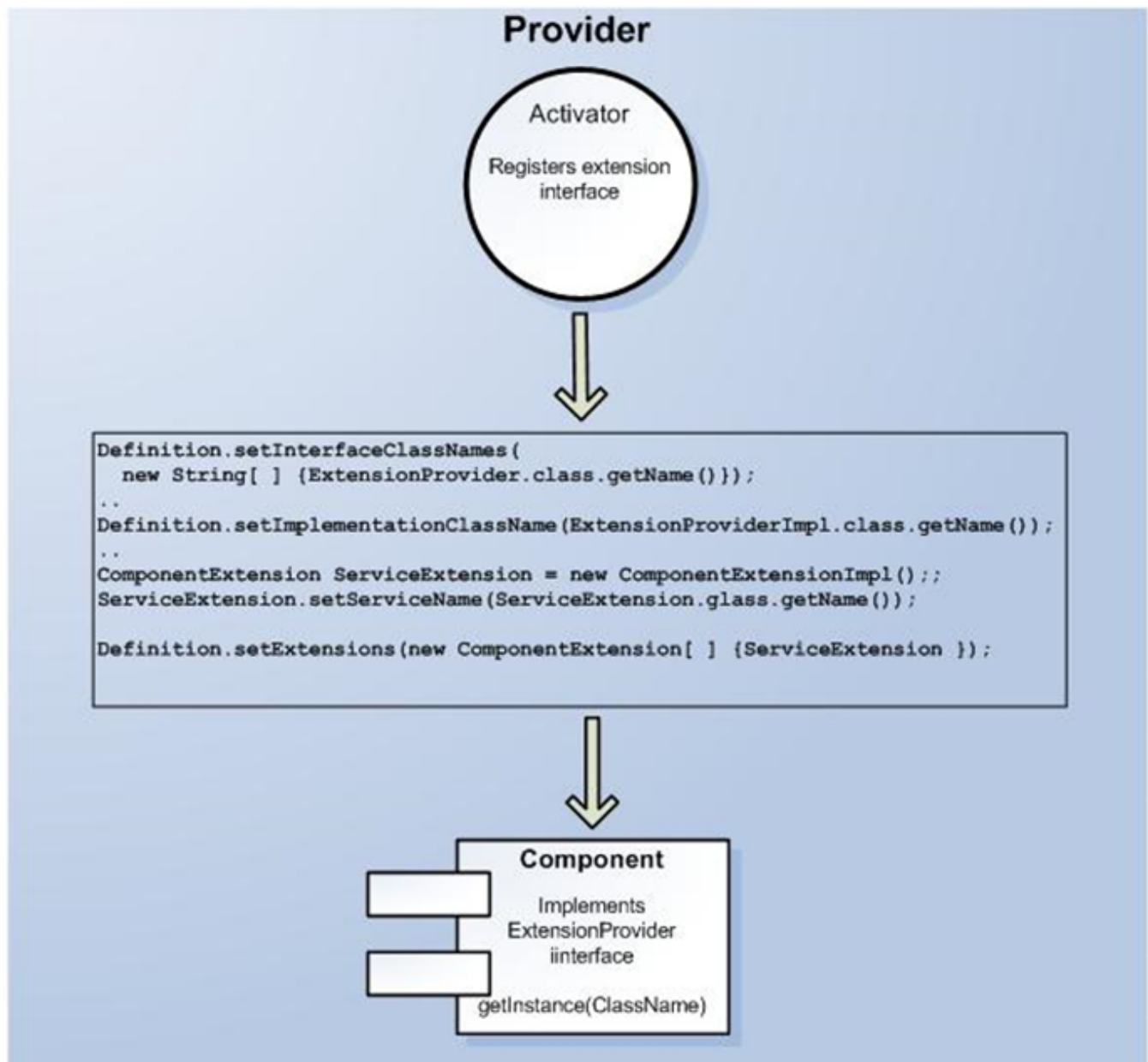
The following figure illustrates the basic steps that are required to make a plugin extendable (consumer) and to create an extension for it (provider):

Consumer



```
String[] componentInterfaces = [Component.class.getName()  
ElementComponent.class.getName(), Extensible.class.getName() ];  
elementDefinition.setInterfaceClassNames (componentInterfaces);
```





[Back to Top](#)

Extension Consumers

Extension consumers are plugin components that consume the extension services of plugins that provide these services. A plugin component becomes a consumer by adding the interface classname `Extensible` to its interface classnames. The framework recognizes this interface and will register the component in the framework as a plugin component extension consumer.

A consumer can consume the extension services of more than one component and the developer of the plugin component that consumes the service decides how and in what order these services are consumed. The developer therefore has access to the set of extensions that are registered at the consumer component by the `getExtensions()` method.

Registering a Plugin Consumer Component

By registering the `Extensible` interface, the component becomes a plugin component consumer and will be ready for consuming extension services in the service framework.

```
String[] componentInterfaces = { Component.class.getName(),
    ElementComponent.class.getName(), Extensible.class.getName() };
elementDefinition.setInterfaceClassNames(componentInterfaces);
```

Registering Consumer Extensions

To define which services a plugin component will consume, you must register the extension services. This can be done by registering the exact interfaces the component will consume:

```
// Extensions
ComponentExtensionImpl viewExtensionPoints = new ComponentExtensionImpl();
viewExtension.setServiceName(ViewExtension.class.getName());
elementDefinition.setExtensions(new ComponentExtensionImpl[] { viewExtension } );
```

[Back to Top](#)

Extension Providers

Extension providers are plugin components that provide extension services for other plugins. A plugin component becomes an extension provider by adding the interface classname `ExtensionProvider` to its interface classnames. The framework recognizes this interface and will register the component in the framework as a plugin component extension provider.

Register a Plugin Provider Component

By registering the `ExtensionProvider` interface, the component becomes a plugin component provider and will be ready to provide extension services to the service framework:

```
definition.setInterfaceClassNames(new String[]{ExtensionProvider.class.getName()});
```

The plugin component also has to register a valid implementation of the `ExtensionProvider` interface that can act as a factory for the service extensions:

```
definition.setImplementationClassName(ExtensionProviderImpl.class.getName());
```

The provider implementation has to implement the `ExtensionProvider` interface and therefore has to implement the following methods :

- `public String getTargetComponentId()` - This method must return the ID of the component that is allowed to consume the extension services provided by the component.
- `public Object getServiceInstance(Class interfaceClass)` - This method must return an instance of the implementation class of the requested interface.

It is your choice to decide whether the `getServiceInstance(...)` method should:

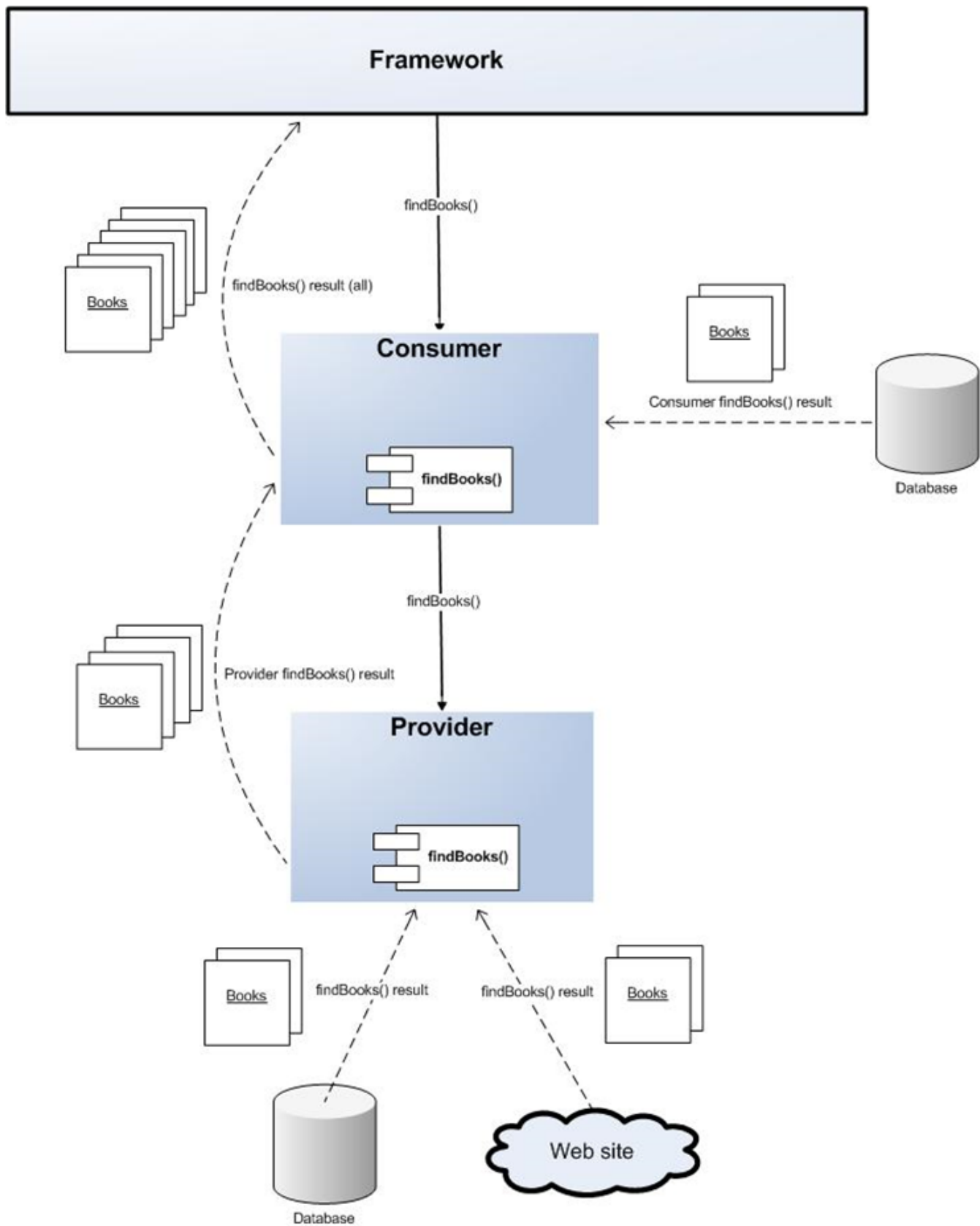
- Always returns the exact same object (singleton instance). This object is cached within the service component
- Create a new instance of the requested object for each request to the extension service.

Registering Provider Extensions

To define exactly which services a plugin component will provide, the registration of these extensions services is needed. This can be done by registering the exact interfaces the component will provide in the plugin Activator:

```
// Extensions
ComponentExtensionImpl bookServiceExtension = new ComponentExtensionImpl();
bookServiceExtension.setServiceName(FindBookService.class.getName());
definition.setExtensions(new ComponentExtension[] { bookServiceExtension });
```

The extension service interfaces must be registered at the consumer and provider component. The following figure shows the workflow of an example of an extension for the GX Books Example plugin. The extension (provider) extends the `findBooks()` method of the consumer further by searching web sites and other databases for books when a user executes a search:



Alternate View Extension Example

To extend a plugin with an alternate view, the framework interface `ViewExtension` is provided by the framework which can be registered by a plugin extension consumer and provider and can be implemented by the provider extension. The alternate view extension is a subclass of the view extension. Registration of the `ViewExtension` interface in the `Activator` looks like:

```
// Extensions
ComponentExtensionImpl alternateViewExtension = new ComponentExtensionImpl();
alternateViewExtension.setServiceName(ViewExtension.class.getName());
```

The `ViewExtension` interface must be registered at the extension provider and consumer component. The consumer component will automatically delegate the MVC calls to the controllers provided by the view extension points.

The extension provider must provide an implementation of the `ViewExtension` interface and return an instance of the implementation through the `getServiceInstance(...)` method of the `ExtensionProvider` interface:

```
public class ExtensionProviderFactory extends SimpleServiceComponent implements ExtensionProvider {

    public Object getServiceInstance(Class interfaceClass) {
        return new AlternateViewProvider(this);
    }
    public String getTargetComponentId() {
        return "com.libris4you.books.bookreviewselement";
    }
}
```

AlternateViewProvider

The extension should return an implementation of the `ViewExtension` interface as a result of the call to the `getServiceInstance()` method. The component that is extended and consumes the view extension will delegate the MVC calls to the controller that is returned by the `ViewExtension`. `getDelegatedController()` method:

```
public class AlternateViewProvider implements ViewExtension {
    private Component component;

    AlternateViewProvider(Component component) {
        this.component = component;
    }

    public DelegatedController getDelegatedController() {
        ComponentController controller = new AlternateViewController();
        controller.setComponent(component);
        return controller;
    }
}
```

AlternateViewController

The controller for the alternate view has access to the parent controllers through `getParent` and can be used to manipulate the behavior of this parent controller. To override the complete default view of an extension consumer the `createEditViews()` method should be implemented and its implementation should add a default edit view to the parent controller in order to override the default view of this controller and its component:

```

public class AlternateViewController extends ElementComponentController {
    @Override
    public Object formBackingObject(HttpServletRequest request)
        throws ServletException {
        super.formBackingObject(request);
        Object fbo = getParentController().formBackingObject(request);
        setDelegatedControllers(fbo, request);
        return fbo;
    }

    @Override
    protected void createEditViews() {
        super.createEditViews();
        ComponentControllercontroller = (ComponentController) getParentController();
        controller.addEditView("editBookReviewsElement.jspf", "", this.getComponent());
    }
}

```

Implement Alternate View (JSP)

To provide an alternate view for a consumer plugin component a JSP file must be provided in the `src/main/resources/editpresentation` directory of the plugin. This JSP view must be added as a view in the `createEditViews()` method of the controller as can be seen in the above example.

To add the view to the controller the method `addEditView(String jspLocation, String viewname, Component component)` must be used. The third component argument needs to tell the parent controller where to find the exact location of the JSP which is determined by ID of the component bundle that provides the JSP.

[Back to Top](#)

Publishing and Subscribing to Events

The publish and subscribe pattern is a pattern in which publishers broadcast an event which is received by the subscribers. A subscriber subscribes itself to particular types of events such that it only receives events in which it has interest.

XperienCentral supports the publish and subscribe pattern by the implementation of an Event Manager service. All event management-related interfaces are contained by the package `nl.gx.webmanager.services.event`. The most important interfaces of this package are:

- `EventManagerService`. The event manager service (the publisher)
- `EventHandler`. The event handler (the subscriber)
- `Event`. The event broadcasted by the Event Manager service and received by the event handler.

There is only one publisher in XperienCentral, which is the only implementation of the Event Manager service. By default, this service only broadcasts events on basic operations. The following table shows the events that are published by this Event Manager for each action type on each object type. For example, when creating a new page, a POST event is published, when deleting a page section, a PRE event is published, the page section is deleted, and then a POST event is published, and so forth.

Object Type	Page	Page Version	MediaItem	MediaItemVersion	Element	Website
RETRIEVE					X ¹	
COPY	X	X	X	X	X	
DELETE	X	X	X	X	X	X
MOVE	X	X	X	X	X	
CREATE	X ¹	X ¹	X ¹	X ¹	X	X
UPDATE	X ²	X ²	X ²	X ²	X	
CHANGED	X ¹	X ¹	X ¹	X ¹	X ¹	
PublicationStatus		X ¹		X ¹		

¹ Only a POST event is published.

² The Content API does not publish this event if you change a content item; this is the responsibility of the updater. In XperienCentral, this event is published by the Spring MVC and the REST API.

Each of the above object types implements an event interface: `PageEvent`, `PageModelEvent`, `PageVersionEvent`, `MediaItemEvent`, `MediaItemVersionEvent`, `ElementEvent`, and `WebsiteEvent`. These implementations are used to publish RETRIEVE, COPY, DELETE, MOVE, CREATE, and UPDATE events. The scope of these events is the content type's class, for example `Page.class`. In addition, `HistoryEvent` is used to publish the CHANGED event, and `PublicationStatusEvent` is published to indicate that the publication status of a content item has changed. The scope of these events depends on the content type, and that the latter does not have an event action.

XperienCentral `nl.gx.webmanager.services.event` Package

The `nl.gx.webmanager.services.event` package contains the following interfaces:

Interface	Description
<code>EntityEvent</code>	Contains the events specific to CRUD operations. This class is implemented by the <code>ElementEvent</code> , <code>MediaItemEvent</code> , <code>MediaItemVersionEvent</code> , <code>PageEvent</code> , <code>PageVersionEvent</code> , and <code>WebsiteEvent</code> interfaces.
<code>Event</code>	Contains methods for retrieving information about the event that occurred.
<code>EventHandler</code>	Contains a method for reacting to event notifications.
<code>EventManagerService</code>	Contains methods for publishing, subscribing, and unsubscribing to events.

In addition to the standard actions (create, copy, update, move, retrieve, and delete), you can also create custom actions that perform other functions. These custom actions can also be published and subscribed to. If you want to publish custom events, you will need to implement a custom event and publish this event using the Event Manager service.

Writing an Event Handler

A custom event handler should implement the `nl.gx.webmanager.services.event.EventHandler` interface. This interface contains only one method, which is `onEvent`. The `onEvent` method takes the event that triggered the handler as input argument. The event contains information about the action that triggered the event, the component that did throw the event and the event type (PRE or POST). What to do with the event is up to the implementation of the event handler.

The following code snippet provides an example implementation of a custom event handler.

```

public class CustomMediaItemEventHandler implements EventHandler {
    public CustomMediaItemEventHandler() {
    }

    public void onEvent(Event event) {
        if (event instanceof MediaItemEvent) {
            MediaItemEvent mediaItemEvent = (MediaItemEvent) event;

            // Get the media item version
            MediaItemVersion mediaItemVersion = null;
            if (mediaItemEvent.getMediaItem().getCurrent() != null) {
                mediaItemVersion = mediaItemEvent.getMediaItem().getCurrent();
            } else {
                mediaItemVersion = mediaItemEvent.getMediaItem().getPlanned();
            }

            // Check if this really is our custom article
            // media item version
            if (!CustomArticleMediaItemVersion.class.isAssignableFrom(mediaItemVersion.getClass()))
        {
            return;
        }

        // Handle the event
        if (event.getEventAction().equals(EntityEvent.CREATE)) {
            handleCreateEvent(mediaItemEvent, mediaItemVersion);
        }
        if (event.getEventAction().equals(EntityEvent.DELETE)){
            handleDeleteEvent(mediaItemEvent, mediaItemVersion);
        }
    }

    /**
     * Does something when an article of our custom type is created.
     */
    private void handleCreateEvent(MediaItemEvent mediaItemEvent, MediaItemVersion mediaItemVersion) {
        ...
    }

    /**
     * Does something when an article of our custom type is deleted.
     */
    private void handleDeleteEvent(MediaItemEvent mediaItemEvent, MediaItemVersion mediaItemVersion) {
        ...
    }
}

```

Subscribing to Events

To subscribe the event handler to particular events, the handler must be subscribed by the publisher (the Event Manager service). The `EventManagerService` interface contains the following methods:

Method	Description
publish	<p>Publishes an event to the framework.</p> <pre>void publish(Event event)</pre> <p>where <code>event</code> is the event to be published.</p>

subscribe	<p>Subscribes to a particular event handler.</p> <pre>void subscribe(EventHandler handler, Event.Type eventType, java.lang.Class<?> desiredScope)</pre> <p>where <code>handler</code> specifies the name of the handler to subscribe to, <code>eventType</code> specifies the type of event you are subscribing to, and <code>desiredScope</code> specifies the scope class of the event.</p>
unsubscribe	<p>Unsubscribes from a particular event handler.</p> <pre>void unsubscribe(EventHandler handler, Event.Type eventType, java.lang.Class<?> desiredScope)</pre> <p>where <code>handler</code> specifies the name of the handler to unsubscribe from, <code>eventType</code> specifies the type of event you are unsubscribing from, and <code>desiredScope</code> specifies the scope class of the event you are unsubscribing from.</p>

To subscribe the event handler, the `subscribe` method on the Event Manager service must be invoked. The `subscribe` method takes, in addition to the event handler, an event type and a desired scope as input arguments. The event type may be PRE or POST. The scope indicates the object type on which the event applies. For the basic events the appropriate scopes are mentioned in the event overview table. It is also possible to provide a superclass or interface as scope. For example, to receive all `PageVersion` events, you can subscribe to events with either scope `PageEvent.class`, or any class it extends or interface it implements.

The `unsubscribe` method can be used to unsubscribe the event handler from events. Usually an event handler is subscribed in the `onStart()` method of the component and unsubscribed in the `onStop()` method.



It is very important to unsubscribe. If you don't, the event handler will keep receiving and handling events even when the plugin is stopped. When a plugin is updated, the event will be received by an old as well as a new instance of the event handler and thus be handled twice.

The code snippet below shows an example of subscribing to an event handler of an event.

```
public class CustomMediaItemComponent extends SimpleMediaItemComponent {
    private CustomEventHandler myEventHandler = null;

    public void onStart() {
        // Initialize new event handler
        myEventHandler = new CustomEventHandler();

        // Subscribe to media item post event
        myEventManagerService.subscribe(myEventHandler, Event.Type.POST, MediaItem.class);
    }
    public void onStop() {
        // Unsubscribe to media item post event
        myEventHandler.unsubscribe(myEventHandler, Event.Type.POST, MediaItem.class);
    }
}
```

Publishing Events

To publish specific events, use the Event Manager service. This service contains a method `publish(Event)` which should be used to publish your event in XperienCentral.

There are several guidelines that are important to follow when publishing events:

- A plugin should never publish a CHANGED event. This is the history service's responsibility and it will publish it if a content item has changed. Even if you are certain the content item has changed, an UPDATE event should be published.
- An UPDATE event should be published only once. When multiple properties have changed, publish an event for all the properties at once and not individually.
- When publishing an event, both the PRE and POST event should be copied unless only a POST event is published (see the event overview table). The PRE event should be published before applying any change, and the POST event should be published thereafter.
- If available, the most specific implementation of an event should be used. For example, if a page has been updated, publish a `PageEvent` instead of an `EntityEvent` with scope `Page.class`.

The code snippet below shows an example of publishing a CREATE and UPDATE event. `MyItemEvent` is an implementation of `Event`.

```
public class CustomMediaItemComponent extends SimpleMediaItemComponent {

    public MyItem createMyItem() {
        MyItem newItem = new MyItem();

        myEventManagerService.publish(new MyItemEvent(Event.Type.POST, EntityEvent.CREATE, newItem);
        return newItem;
    }

    public void updateMyItem(MyItem item) {
        myEventManagerService.publish(new MyItemEvent(Event.Type.PRE, EntityEvent.UPDATE, item);

        // Update item property 1
        // Update item property 2
        // Update item property 3

        myEventManagerService.publish(new MyItemEvent(Event.Type.POST, EntityEvent.UPDATE, item);
    }
}
```