

The image features a dark blue background with a low-poly, geometric pattern in various shades of blue. The Hybris Software logo is prominently displayed in the center-left. The logo consists of a stylized 'h' inside a circle, followed by the words 'hybris software' in a bold, white, sans-serif font. Below this, the text 'An SAP Company' is written in a smaller, white, sans-serif font.

(h) hybris software
An SAP Company

hybris Developer Training Part I - Core Platform

Event system



- The hybris Event System is based on the Spring event system
- One software component acts as a source and publishes an event that is received by registered listeners
- Event listeners are objects that are notified of events and perform business logic corresponding to the event that occurred
- Events can be published locally or across cluster nodes
- Events might be transaction aware

- An Event is an instance of a subclass of AbstractEvent and contains a source object:

```
public class AfterItemCreationEvent extends
AbstractPersistenceEvent
{
    private final String typeCode;

    public AfterItemCreationEvent(final String typeCode, final
        PK pkCreated)
    {
        super(pkCreated);
        this.typeCode = typeCode;
    }
}
```

- Event listeners allow you to react to an event
- To implement an event listener:
 - Extend the `AbstractEventListener` class
 - Override the `onEvent()` method

```
public class AfterInitializationEndEventListener extends
    AbstractEventListener<AfterInitializationEndEvent>
{
    ...
    @Override
    protected void onEvent(final AfterInitializationEndEvent event)
    {
        getValidationService().reloadValidationEngine();
        LOG.info("Reloaded validation framework.");
    }
}
```

→ Two options to register the event listener:

→ as a bean in the Spring application context:

```
<bean id="myEventListener"  
      class="my.package.MyEventListener"/>
```

→ by dynamically adding listeners at runtime using eventService:

```
@Resource  
EventService eventService;  
eventService.registerEventListener( new MyEventListener() );
```

→ The Service Layer's `EventService` allows you to:

→ Register event listeners with

```
eventService.registerEventListener( myEventListener )
```

→ Publish events using the method

```
eventService.publishEvent( myEvent )
```

→ To access this service, add a Spring resource to your class:

```
@Resource;  
private EventService eventService;
```

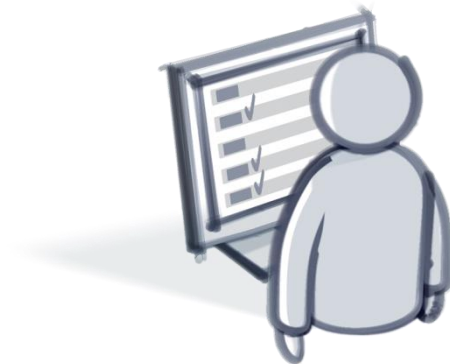
- Events are processed synchronously by default
- This is often undesirable as the main thread waits until events are processed, and this may impede performance
- There are two possible ways to force events to be processed asynchronously:
 - by configuring `PlatformClusterEventSender`
 - by using `ClusterAwareEvents`
- For each asynchronous event, network traffic occurs.

- ➔ To implement a `ClusterAwareEvent`, adding the following method to your event class causes events to be published only to the source node of the event:

```
@Override
public boolean publish(final int sourceNodeId,
                      final int targetNodeId)
{
    return (sourceNodeId == targetNodeId);
}
```

- Events that are published only at the end of a transaction
- Implement the `TransactionAwareEvent` interface
 - `publishOnCommitOnly`: Event will be published depending on the success of the transaction
 - `getId`: two events with the same id will be published only once

- There are predefined **ClusterAware** and **TransactionAware** events that are processed asynchronously:
 - **AfterItemCreationEvent** – Triggered after an item is created
 - **AfterItemRemovalEvent** – Triggered after an item is removed



➔ Non-ClusterAware Events are only published synchronously

AfterInitializationEndEvent

Triggered after initialization has ended

AfterInitializationStartEvent

Triggered after the initialization has started

AfterSessionCreationEvent

Triggered after the session was created

AfterSessionUserChangeEvent

Triggered after a new user is assigned to the session

BeforeSessionCloseEvent

Triggered before a session is closed

Dynamic scripting allows user to create listeners and make them available at run time and without rebuilding the system.

```
class MyScriptingEventListener extends AbstractEventListener<AbstractEvent>
{
    @Override
    void onEvent(AbstractEvent event)
    {
        if(event instanceof TestScriptingEvent){
            println 'hello groovy! ' + new Date();
        }
        else {
            println 'another event published '
            println event
        }
    }
}
new MyScriptingEventListener();
```

- **ScriptingEventService** is a scripting dedicated event service that allows registering and unregistering the dynamic event listeners by scriptURI at runtime.

```
scriptingEventService.registerScriptingEventListener(  
    'model://myEventListenerScript')
```

- The **ScriptListenerWrapper** wraps the dynamic listener, which is necessary to always get the same listener instance for a given ScriptURI

- Publishing a custom event using a script

```
event = new TestScriptingEvent('myEvent')  
eventService.publishEvent(event);
```

1. How do you publish an event?
2. Explain how you can force events to be published asynchronously.

(x)