TIBCO PageBus™

Developer's Guide

Software Release 1.2
December 2007

TIBCO®
The Power of Now®

Important Information

Contents

Preface

This manual provides an introduction to TIBCO PageBus™, a publish/subscribe message delivery hub for mash-ups and rich internet applications that simplifies the development and extension of web applications composed of Ajax components, including rich portlets. This manual explains the PageBus™ concepts, programmatic interfaces, and best practices that help you successfully build composite rich internet applications (RIAs).

Topics

Related Documentation

This section lists documentation resources you may find useful.

TIBCO PageBus Documentation

The following document forms the TIBCO PageBus documentation set:

- *TIBCO PageBus Developer's Guide* Read this manual for instructions on using the product API.

Other TIBCO Product Documentation

You may find it useful to read the documentation for the following TIBCO products. However, this is not essential. You can use TIBCO PageBus without these other products.

- TIBCO General Interface™ Builder software: General Interface™ Builder is a development environment for building rich Internet applications. The object modeling features of General Interface Builder enable developers to quickly create reusable GUI components and assemble them into full applications or application modules. Applications can be accessed in a web browser from a URL, existing HTML page, or portal environment.
- TIBCO PortalBuilder® software: PortalBuilder® is a portal deployment platform. PortalBuilder provides core services such as customization, aggregation of content and services, personalization, presentation, security, access control and delivery. Portal pages are created based on modular templates, allowing maximum design flexibility and easy localization.
- TIBCO Ajax Message Service™ software: Ajax Message Service™ provides a scalable and reliable way to push or stream real-time information to web browsers and other applications over HTTP based on a publish/subscribe paradigm.

Typographical Conventions

The following typographical conventions are used in this manual.

Table 1 General Typographical Conventions

Convention	Use
`code font`	Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example: Use `MyCommand` to start the foo process.
`bold code font`	Bold code font is used in the following ways: • In procedures, to indicate what a user types. For example: Type **`admin`**. • In large code samples, to indicate the parts of the sample that are of particular interest. • In command syntax, to indicate the default parameter for a command. For example, if no parameter is specified, `MyCommand` is enabled: MyCommand [**`enable`** | disable]
italic font	Italic font is used in the following ways: • To indicate a document title. For example: See *TIBCO BusinessWorks Concepts*. • To introduce new terms For example: A portal page may contain several *portlets*. Portlets are mini-applications that run in a portal. • To indicate a variable in a command or code syntax that you must replace. For example: `MyCommand` *pathname*
Key combinations	Key name separated by a plus sign indicate keys pressed simultaneously. For example: Ctrl+C. Key names separated by a comma and space indicate keys pressed one after the other. For example: Esc, Ctrl+Q.
📝	The note icon indicates information that is of special interest or importance, for example, an additional action required only in certain circumstances.
💡	The tip icon indicates an idea that could be useful, for example, a way to apply the information provided in the current section to achieve a specific result.
⚠	The warning icon indicates the potential for a damaging situation, for example, data loss or corruption if certain steps are taken or not taken.

How to Contact TIBCO Support

If you are using a product that embeds TIBCO PageBus, please contact TIBCO Support Services as follows.

- For an overview of TIBCO Support Services, and information about getting started with TIBCO Product Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  https://support.tibco.com

  Entry to this site requires a user name and password. If you do not have a user name, you can request one.

TIBCO PageBus as a standalone product is offered by TIBCO Software under the terms of a BSD license. If you would like support for this product, you may purchase a product that embeds PageBus and provides support, such as TIBCO Ajax Message Service™.

For self-service support, education, and access to the TIBCO Developer Network, visit:

http://developer.tibco.com

For an overview of TIBCO Support Services and information about getting started with TIBCO Product Support, visit:

http://www.tibco.com/services/support

Chapter 1 Introduction

This chapter gives an introductory description of TIBCO PageBus, and explains how PageBus enables interactions between rich web components.

Overview

In order to achieve greater flexibility and promote broader use of available business assets, application developers are increasingly taking advantage of service-oriented architectures (SOA). This trend is contributing to the rapid growth of “enterprise mash-ups”—composite Ajax applications that interact with disparate web services across the enterprise.

In a modern service-oriented architecture, the server side is characterized by heterogeneous components that interact using well-defined, coarse-grained service interfaces. These characteristics are uniformly recognized as promoting reuse, loose coupling, and discoverability, essential qualities in an era of rapid change. On the client side, however, many enterprise web applications are monolithic (and built from scratch each time), with either too much functionality or too little functionality for many user tasks.

Simultaneously, enterprises are increasingly adopting Ajax component-based approaches as a way to build rich enterprise applications. An Ajax component strategy separates UI functionality into coarse-grained modules that can be pieced together or assembled by application developers. Ajax allows developers to access disparate web services, and create complex user interfaces.

These two trends are highly complimentary. Web applications can be constructed from different rich Ajax components, and these components can interact with each other when assembled on the page—enterprise mash-ups. When dashboards or portals are used, business users can even assemble and personalize application components on their own pages, creating mash-ups dynamically.

TIBCO PageBus provides a standards-based way to integrate Ajax components and portlets through publish-subscribe events.

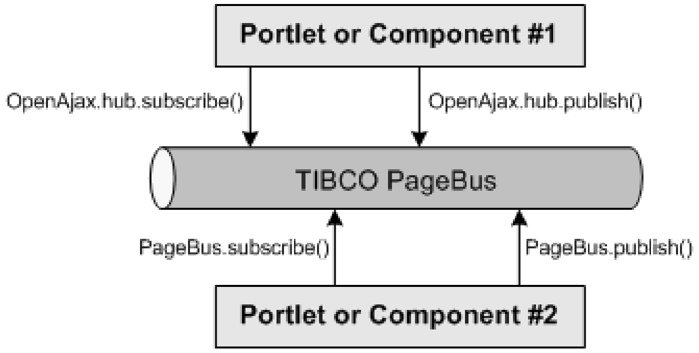
Support for the OpenAjax Hub 1.0 Standard

PageBus supports the OpenAjax Hub 1.0 specification as of December 2007. It includes a full implementation of the OpenAjax Hub API. It also provides a higher-level interface that uses the OpenAjax hub and adds the following functionality:

- PageBus Repository functionality (see Chapter 2, PageBus Repository, on page 7)
- FIFO message sequencing
- More robust error handling

Messages published using the higher-level API can be consumed by subscribers using the lower-level OpenAjax API, and vice versa.

Figure 1 Compatibility with OpenAjax Hub



TIBCO PageBus

TIBCO PageBus is a pure JavaScript, *publish-subscribe* message delivery hub that uses *subjects* to identify groups of subscribers. *Subscribers* listen (or *subscribe*) via subjects, which usually represent business- or application-level events such as a user's selection of a customer or a user's creation of a new calendar appointment. *Publishers* send (or *publish*) *messages* on subjects.

When messages are published on a subject, they are delivered to all of the subject's current subscribers. When a subscriber no longer needs to listen to traffic on a subject, it can *unsubscribe* from the subject.

Figure 2 shows a logical representation of PageBus, publishers, and subscribers.

Figure 2 PageBus, Publishers, and Subscribers

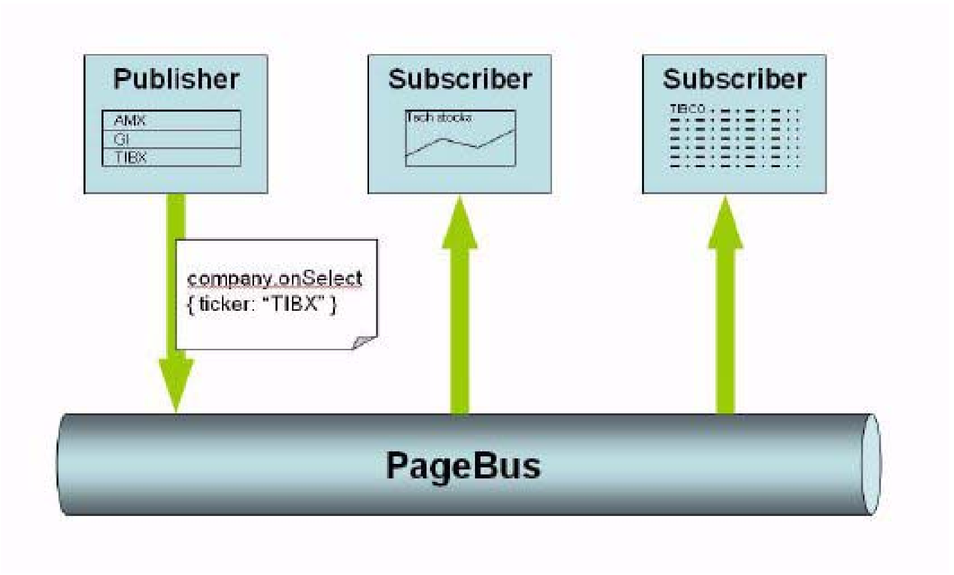
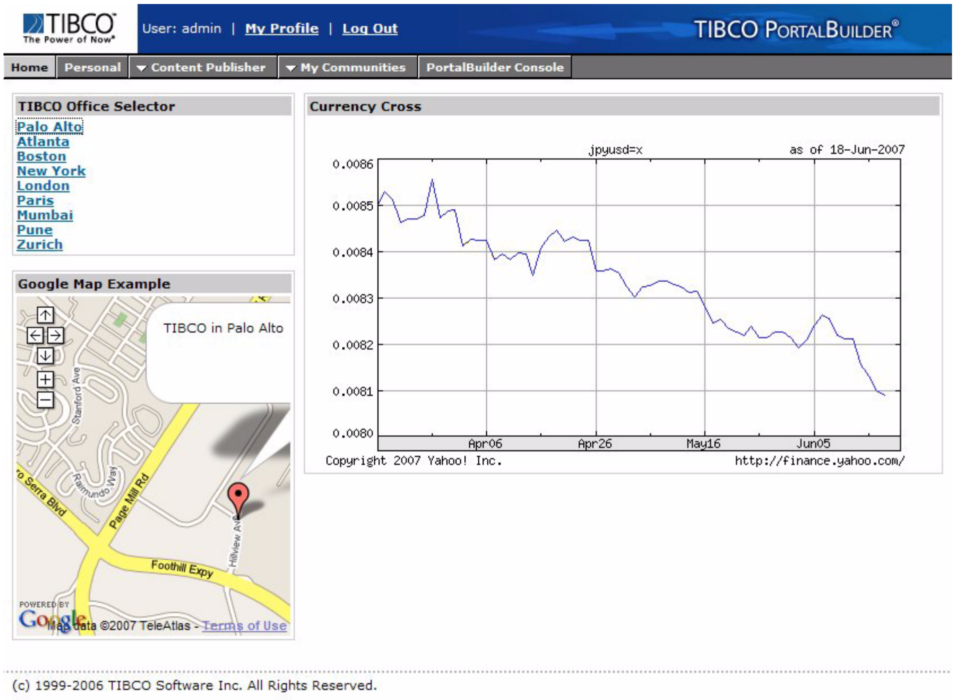


Figure 3 illustrates PageBus publish-subscribe messaging between three portlets in an enterprise portal. In this figure, the upper left portlet contains a list from which the user chooses one of TIBCO's worldwide offices. The other two portlets then display information related to the selected office. When a user clicks a city in the TIBCO Office Selector portlet, that portlet publishes a message on the subject `eg.geo.Location.onSelect`. The other two portlets, Currency Cross and Google Map Example, are subscribers to `eg.geo.Location.onSelect`. Whenever these subscribers receive a message, they perform whatever action is appropriate to their function. As the figure below shows, Currency Cross displays a graph of recent exchange rates for the local currency, while Google Map Example maps the location of the selected office.

Figure 3 Portlets and Messaging



Chapter 2 PageBus Repository

Topics

Using the PageBus Repository

Application components on a page may need to store information in a way that allows other components to look it up on demand. The components that use this information may need to be notified when this information is updated.

PageBus 1.2 implements this common pattern by providing a stateful name-value repository that is integrated with the publish-subscribe message bus. When repository entries are added, updated, or removed, the repository automatically publishes messages. Thus, if a repository entry under the name `com.example.foo.bar` is updated, an update notification is automatically published on `com.example.foo.bar`. This functionality allows components to store values for use by other components, while at the same time ensuring that if the values change, the consumers of these values will be notified.

The repository makes it easy for application components to interact statefully while remaining loosely coupled and event-based.

The repository supports two operations: store and query.

Store stores a value in the repository under a subject name and then publishes a message advertising the change. The subject name on which the message is published is the same as the name of the retistry entry. The message payload includes name and new value.

Query finds all repository entries matching a specified subject name. For each matching entry, `query` invokes a callback function. If the subject contains wildcard tokens then there might be multiple matches.

Chapter 3 TIBCO PageBus Interface Reference

Topics

Overview

TIBCO PageBus consists of two components:

- PageBus API, which includes two objects—PageBus and Subscription.
- Subjects that are used by the APIs to send messages as well as subscribe to them.

This section describes each of the components and how to use them.

PageBus API

The PageBus object in the PageBus API is a *singleton* object that you can use to send and receive messages between your Ajax components.

When using the PageBus API, there must be a *single* instance of the PageBus object. If you include `pagebus.js`, this single instance is automatically created by `pagebus.js`. To subscribe to a subject, you must define a callback function and pass it into `PageBus.subscribe.` See Subscription Callback on page 22. The callback function is used by PageBus to handle messages that are published onto the subscribed subject.

The Subscription object is returned by `PageBus.subscribe`. It is the handle to your subscription, and you can use it to unsubscribe from the subscription.

The table below shows the operations you can perform with the PageBus object.

Table 2 PageBus Object

Operation	Comment
publish	Publish a message on a subject. See PageBus.publish on page 13 for more information.
subscribe	Subscribe to a subject, specifying a callback function to be invoked when a message is published on the subject. See PageBus.subscribe on page 14 for more information.
unsubscribe	Cancel a subscription on a subject. See PageBus.unsubscribe on page 16 for more information.
version	Return the version number. See PageBus.version on page 17 for more information.
store	Store a value in the PageBus repository. See PageBus.store on page 18 for more information.

Table 2 PageBus Object

Operation	Comment
query	Find all PageBus repository entries that match a specified name and invokes callback functions to deliver these values to the application. See PageBus.query on page 20 for more information.

Subjects

Subjects are period-delimited strings. For example: `eg.customer.onSelect` and `log.info`. The periods divide the subject strings into *tokens*. For example, `eg.customer.onSelect` has three tokens: `eg`, `customer`, and `onSelect`.

Subscribers can subscribe to specific subjects, or they can subscribe to *wildcard subjects* that match multiple subjects. There are two ways to specify a wildcard subject:

- Single-asterisk (*) indicates a single-token wildcard, which matches all tokens at exactly the indicated position. For example, `a.*.c` matches `a.b.c` and `a.thing.c`
- Double-asterisk (**) indicates a multiple-token wildcard, which matches all subsequent tokens from its position onward. For example, `a.**` matches `a.b.c.d` and `a.e`; and `**` by itself matches all legal subjects.

Subject Rules

Subject names must adhere to the following rules:

- Subjects must be *non-null*.
- A subject must consist of dot-delimited tokens. For example:
  - `Hello`
  - `hello.1.2.3`
- A subject can contain any number of tokens. For example:

  `This.is.AN.example.of.A.very.very.long.subject`
- A subject token can contain any number of JSON-compliant characters except for the asterisk character (*) and the period character (**.**) . The backslash (\),

control characters, and quotation marks (") are FORBIDDEN in JSON strings and therefore in PageBus subjects.

If you use the period in a token, such as `he.llo`, it separates the token into two tokens.

For example:

```
FORBIDDEN: a.b.c*d.e
FORBIDDEN: a.b**.c
FORBIDDEN: a.b\c.d
FORBIDDEN: a.b"c."xyz
```

We strongly recommend that you AVOID using in subjects any character that has special meaning in the JavaScript language, such as space, apostrophe ('), colon (:), angle brackets (< >), and others.

- A subject must *not* begin with the underline (_) character.
- Empty tokens are forbidden. For example:
  - — FORBIDDEN: `a.b.c..d.e`
  - — FORBIDDEN: `a.b.c.`
  - — FORBIDDEN: `.a.b.c`
- The single-asterisk (*) and double-asterisk (**) wildcards can be used only for subscription and *not* for publication.
- The double-asterisk (**) wildcard must *not* be followed with any token.

PageBus.publish

Function

This section describes the *publish* operation.

Description

The publish operation publishes one message on a subject. The message is delivered to all current subscribers whose subscriptions match the subject.

If there are multiple subscribers, the order in which the message is delivered to the various subscribers is arbitrary.

If `PageBus.publish` is called twice, so that two messages are published in sequence, then subscribers will always receive the message that was published first before they receive the second message.

Returns

Null

Parameters

Name	Type	Description
`subject`	string	Subject name on which to publish the message. This must *not* be a wildcard subject.
`message`	any	Message content. The reference to the original message is received by all subscribers (rather than a copy).

Throws

`PageBus.BadName`	The specified subject name did not follow the subject rules. See Subject Rules on page 11.
`PageBus.StackOverflow`	Probable infinite publisher-subscriber loop detected and interrupted. This happens if a publish operation causes a callback that publishes again, which causes another callback in an infinite recursion. `PageBus.publish` detects the cycle after about 20 iterations and throws this exception.

Remarks

All PageBus functions can be safely called from within the subscription callback function. See Subscription Callback on page 22 for more information on the callback function.

PageBus.subscribe

Function

This section describes the *subscribe* operation.

Description The subscribe operation subscribes to a subject and passes in the reference to a callback function. The callback function is invoked with the subscriberData and message whenever someone publishes a message on the subscribed subject. Subscribed subjects can be wildcard subjects.

Returns Subscription

Parameters

Name	Type	Description
`subject`	string	Subject name to which to create the subscription.
`scope`	object	If the value is non-null, the object specified here becomes the context of the callback function. In other words, when the JavaScript `this` keyword is used in the callback function, it will point to this object. If the value is null, then the object `window` is used as the default context of the callback function.
`callback`	function	Callback function for PageBus to invoke when a message is published on the subscribed subject. See Subscription Callback on page 22 for the required function signature. This parameter must *not* be null.
`subscriberData`	any	User-defined. Null values are permitted. This is useful when the subscriber needs to access or update any data members when the callback function is invoked.
filter	function	Optional callback function. Returns a boolean. If present, and having a non-null value, it is called before the subscriber callback. If the filter function returns true, then the subscription's callback function is called. If it returns false, the callback is not called.

Throws

`PageBus.BadName`	The specified subject name did not follow the subject rules. See Subject Rules on page 11.
`PageBus.BadParameter`	A null value was specified for the parameter `callback`.

Remarks `PageBus.subscribe` can be safely called from within the subscription callback function. See Subscription Callback on page 22 for more information on the callback function.

The `filter` Parameter

The `filter` parameter is optional. It corresponds to the OpenAjax Hub 1.0 `subscribe` **`filter`** parameter. The signature of the `subscribe` function when this parameter is present is is:

```
PageBus.subscribe(name, scope, callback, subscriberData, filter)
```

The filter parameter is a callback function. If present, if is called before the subscriber callback. If has the following signature:

MyFilterFunc(subject, message, subscriberData)

and returns a Boolean. If the filter function returns true, then the subscription's callback function is called. If it returns false, the callback is not called.

If the filter parameter is not provided, e.g., `PageBus.subscribe(name, scope, callback, subscriberData)`, or if `null` is specified for the filter function, then no filter function is associated with the subscription and behavior is exactly as it was in PageBus 1.1.

Because the filter function is optional, PageBus 1.1 applications will continue to work on 1.2. There is no need to modify the 1.1 code.

PageBus.unsubscribe

Function

This section describes the *unsubscribe* operation.

Description The unsubscribe operation cancels a subscription, using the Subscription object returned by `PageBus.subscribe` as a handle to the subscription.

Returns Null

Parameters

Name	Type	Description
`subscription`	Subscription	The Subscription object returned by the `PageBus.subscribe` operation.

Throws

`PageBus.BadParameter`	The value of the subscription parameter is not a valid subscription. Note that no exception will be thrown if `PageBus.unsubscribe` is called on a subscription that has already been unsubscribed.

Remarks `PageBus.unsubscribe` can be safely called from within the subscription callback function. See Subscription Callback on page 22 for more information on the callback function.

PageBus.version

Data Member

This section describes the *version* data member.

Description A string constant (X.Y.Z) that indicates the version of the PageBus API.

Value The string "1.2.0".

PageBus.store

Function

This section describes the *store* operation.

Description The store operation stores a value in the PageBus repository under the specified name. It then publishes a message on the specified name to notify any subscribers of the change. If the value is null, then the store operation removes the specified repository entry.

Returns Null

Parameters

Name	Type	Description
`subject`	string	Name under which value is stored in repository, and on which repository then publishes notification message. Must *not* be a wildcard subject or null.
`value`	any	The value that is to be stored. Typically object or string.
`props`	object	Optional properties object. May contain an optional `quiet` property, described in the table below.

The following table describes the **`quiet`** property, which is an optional property of the optional **`props`** properties object:

Name	Type	Description
`quiet`	boolean	If present and true, does not publish a notification message.

Throws

`PageBus.BadName`	The specified subject name did not follow the subject rules for publishers. See Subject Rules on page 11.

`PageBus.StackOverflow`	Probable infinite publisher-subscriber loop detected and interrupted. This happens if a publish operation causes a callback that publishes again, which causes another callback in an infinite recursion. `PageBus.store` detects the cycle after about 20 iterations and throws this exception.

Remarks Can be safely called from within subscriber callback function or a repository `query` callback function.

When storing an object value, store a new object each time, just as you would do if you were publishing the object value. Don't update the existing value. Don't change the properties of a stored object without calling `store()`; such changes won't result in `publish` notifications

The optional `props` object may contain properties that affect how the function behaves. If the object looks like this:

```
{ quiet: true }
```

then the repository will not publish a notification message when the update completes. This might be used if an OpenAjax Hub subscriber is calling `store()` to automatically cache values published by an ordinary publisher; in this case we would not want the repository to redundantly republish every message.

PageBus.query

Function

This section describes the *query* operation.

Description

The query operation finds all PageBus repository entries that match the specified name and invokes a specified callback function to deliver these values to the application. The interface and application-visible behavior resemble those of `PageBus.subscribe`, except that only the querying component has its callbacks invoked.

Query callback functions have essentially the same signature as subscribe callback functions, except that a query callback function returns a Boolean value. If the callback returns `false`, then the query aborts and no additional results are processed. Otherwise, the query continues until there are more results to pass back.

After all of the results have been delivered via the query callback, the callback is invoked one last time with the subject `com.tibco.pagebus.query.done` to indicate that no more results are available.

Returns

Null

Parameters

Name	Type	Description
`subject`	string	Subject name. May include wildcards.
`scope`	object	If the value is non-null, the object specified here becomes the context of the callback function. In other words, when the JavaScript `this` keyword is used in the callback function, it will point to this object. If the value is null, then the object `window` is used as the default context of the callback function.
`callback`	function	Callback function for PageBus to invoke for each query result.
`data`	any	User-defined. Null values are permitted. This is useful when the subscriber needs to access or update any data members when the callback function is invoked.
`filter`	function	

Remarks Can be safely called from within subscriber callback function or a repository query callback function.

PageBus uses the subscriberData, scope, and filter parameters of `PageBus.query` in the same way that it uses the corresponding parameters of `PageBus.subscribe`.

Query Result Callback

This section describes the signature of a callback function that is passed into `PageBus.query()`.

Description A function must be implemented with the parameters below and passed into `PageBus.query` as the callback parameter. It then acts as the subscriber's message handler to handle messages that match the query subject.

Returns Boolean

If true, then continue to invoke callbacks and deliver results. If false, abort the query; no further results will be delivered.

Parameters

Name	Type	Description
`subject`	string	Subject name under which value is stored.
`value`	any	Value stored under the specified name.
`data`	any	data parameter passed into `PageBus.query()`.

Subscription Callback

This section describes the function signature of the subscription callback function.

Description

A function must be implemented with the parameters below and passed into `PageBus.subscribe` as the `callback` parameter. It then acts as the subscriber's message handler to handle messages that match the subscribed subject.

Returns

Null

Parameters

Name	Type	Description
`subject` (Read only)	string	Reference to the string subject on which the message was published.
`message` (Read only)	any	Reference to the message that was published. Do *not* modify the data, as all other subscribers receive the same reference.
`subscriberData`	any	Reference to the subscriberData parameter that was specified by the subscriber in `PageBus.subscribe`. If `subscriberData` is a reference to an object, the object members are writable. If it is an array, elements can be added or removed.

Remarks

`PageBus.subscribe`, `PageBus.unsubscribe` and `PageBus.publish` can be safely called from within the callback function.

Callbacks *should not* throw exceptions. If a callback does throw an exception, PageBus will handle it gracefully (see below) but as a rule, callbacks should catch and handle their own exceptions. Exceptions should *never* be used as a mechanism for communicating between subscriber and publisher.

Message Delivery Sequence for Recursive Publish

PageBus enforces FIFO delivery of messages. If the delivery of a message results in the publication of a second message, all subscribers who receive both messages will receive the first message before they see the second one.

If a subscriber callback calls `PageBus.publish` recursively, `PageBus.publish` ensures FIFO delivery of the messages. It finishes delivering the first message to subscribers before it begins to deliver the recursively published message. Delivery of the recursively published message is deferred until the first message has been delivered.

This behavior is only guaranteed when using `PageBus.publish`. If you call `OpenAjax.hub.publish`, messages are not necessarily delivered in sequence.

In other words, if delivery of a message M1 causes a call to `PageBus.publish(s, M2)`, then all subscribers see M1 before M2. But if delivery of M1 causes a call to `OpenAjax.hub.publish(s, M2)` then applications cannot assume anything about the delivery order; some subscribers may receive M2 before M1.

Errors in Callbacks

As a best practice, subscriber callbacks *should* catch and handle all exceptions; they should *not* throw exceptions. In particular, callbacks must *never* use exceptions to communicate with the publisher.

If a callback throws an exception, `publish` does the following:

1. It catches the exception.
2. It immediately publishes a message on the subject `com.tibco.pagebus.error.callbackError`. Subscribers to this subject will immediately be invoked (bypassing the usual FIFO behavior of `PageBus.publish`). Application code can subscribe to this subject in order to be notified when a callback throws an exception.
3. It finishes delivering the original message to the original subscribers.
4. It returns normally, without throwing an exception.

This behavior achieves several goals:

1. PageBus delivers messages reliably to all subscribers. A single dysfunctional callback does not prevent delivery to other subscribers.
2. Callbacks can never use exceptions to communicate with the publisher.

The payload message published on `com.tibco.pagebus.error.callbackError` is a JavaScript object with the following structure:

```
{
  name: "<string>",  // the PageBus subject associated with the
                     // callback invocation that threw the error
  error: "<string>" // the Error.message value>
}
```

The name and error are escaped using `escape( )`.

Application components can subscribe to `com.tibco.pagebus.error.callbackError`. They will then be notified when `PageBus.publish` catches an exception thrown by a subscriber callback. Handlers for such messages might do things like display alert messages, send the browser immediately to an error page, or replace a faulty component with an error box. Of course, the best practice is for callbacks to catch and handle their own exceptions.

When subscribing to `com.tibco.pagebus.error.callbackError`, the callback function passed into subscribe must *never* throw exceptions.

Chapter 4 Example Code

Topics

Example Code

This section provides examples that demonstrate how to use PageBus operations in your code.

Initialization

Before using PageBus, you must link to `pagebus.js` in the `<head>` area of your web page.

```
<html>
<head>
<title>Home</title>

<script
src="/sta-pb/PortalTemplates/UtilityScriptlets/pagebus/pagebus.js"
                             language="JavaScript1.2"></script>
...
```

PageBus.publish

The following JavaScript code demonstrates the `PageBus.publish` operation.

```
// Create the message
var message = { caseid: "654321",
                workitem: "WI-123",
                summary: "Here is some info." };

// Publish the message using PageBus
window.PageBus.publish("eg.workItem.onSelect", message);
```

PageBus.subscribe

The following JavaScript code demonstrates the `PageBus.subscribe` operation.

```
// Create a scope object
var myScope = {};

// Create a subscriberData.
// This could be an object or simply a string, or null.
var subscriberData = { whatever: "I", want: "to put here" };
```

```
// Subscribe to a subject.
var mySubscription = window.PageBus.subscribe(
                                         "eg.workItem.onSelect",
                                         myScope,
                                         myFunc,
                                         mySubscriberData);
```

PageBus.unsubscribe

The following JavaScript code shows how a subscription can be cancelled.

```
// mySubscription is the Subscription object returned
// by PageBus.subscribe (see PageBus.subscribe).
window.PageBus.unsubscribe(mySubscription);
```

Subscription Callback

The following JavaScript code demonstrates the subscription callback function.

```
function myFunc(subject, message, subscriberData) {

   // Do something with message, subscriberData,
   // and possibly subject.

}
```

Publishing Messages in Subscription Callback

The following JavaScript code demonstrates how to publish a message in the subscription callback function.

```
function myCallbackLogger(subject, message, subscriberData) {
   try {
      var aMessage = { foo: "bar", baz: 2, boo: { a: 1 } };
      window.PageBus.publish("My.Logger", aMessage);
   }
   catch(err) {
      // ... do something ...
   }
}
```

PageBus.store

The following example demonstrates the `PageBus.store` operation.

```
// Create a value
var message = { xyz: 123, abc: "def" };
// Store it
window.PageBus.store("com.example.sample", message);
// Clear it
window.PageBus.store("com.example.sample", null);
```

Subscribing to Update Notifications from PageBus.store

The following example demonstrates subscribing to update notifications from `PageBus.store`.

```
// Subscribing to the subject
var s = window.PageBus.subscribe("com.example.foo.*.bar", myScope,
mySubCallback, "My Data", null);
```

PageBus.query

The following example demonstrates the `PageBus.query` operation.

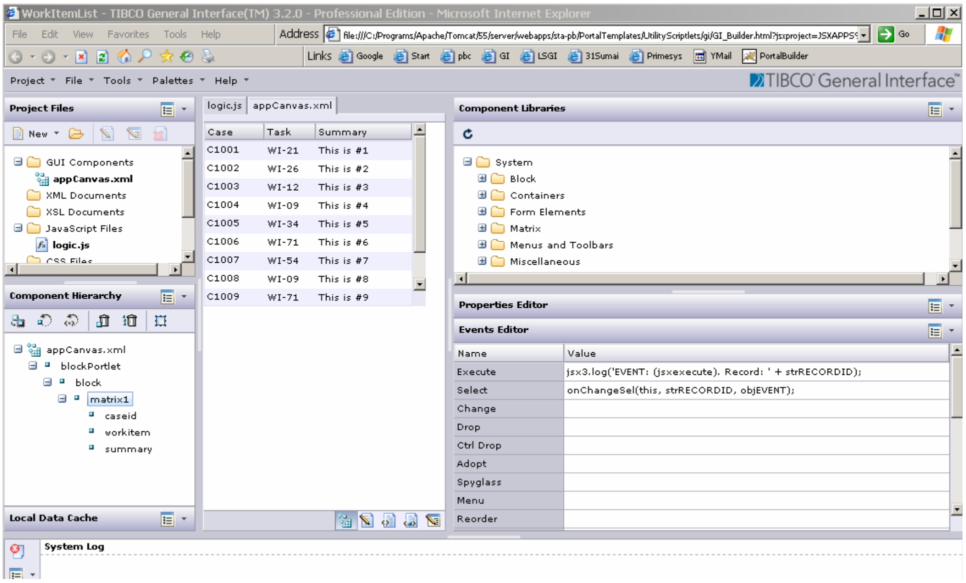
```
var myScope = window;
var myData = "example";

myCallback = function(s, val, d) {
  alert(val);
  return true;
}

// Query 1
window.PageBus.query("com.example.sample", myScope, myCallback,
"Some Data", null);

// Query 2
window.PageBus.query("com.example.foo.*.bar", myScope, myCallback,
"Some Data", null);

mySubCallback = function(s, val, d) {
  alert(val);
  return true;
}
```

Publisher Example in TIBCO General Interface

This section demonstrates how to publish messages using PageBus in TIBCO General Interface™.

Figure 4 shows General Interface Builder with a component called matrix1 that displays a list of work items and publishes a message when a record is selected in the list.

Figure 4 PageBus Publisher in General Interface Builder


WorkItemList - TIBCO General Interface(TM) 3.2.0 - Professional Edition - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address file:///C:/Programs/Apache/Tomcat/55/server/webapps/sta-pb/PortalTemplates/UtilityScriptlets/gi/GI_Builder.html?jsxproject=JSXAPPS
Go
Links Google Start pbc GI LSGI 31Sumai Primesys YMail PortalBuilder
Project File Tools Palettes Help
TIBCO General Interface
Project Files
New
GUI Components
appCanvas.xml
XML Documents
XSL Documents
JavaScript Files
logic.js
CSS Files
Component Hierarchy
appCanvas.xml
blockPortlet
block
matrix1
caseid
workitem
summary
Local Data Cache
logic.js
appCanvas.xml
Component Libraries
System
Block
Containers
Form Elements
Matrix
Menus and Toolbars
Miscellaneous
Properties Editor
Events Editor
System Log

Case	Task	Summary
C1001	WI-21	This is #1
C1002	WI-26	This is #2
C1003	WI-12	This is #3
C1004	WI-09	This is #4
C1005	WI-34	This is #5
C1006	WI-71	This is #6
C1007	WI-54	This is #7
C1008	WI-09	This is #8
C1009	WI-71	This is #9

Name	Value
Execute	jsx3.log('EVENT: (jsxexecute). Record: ' + strRECORDID);
Select	onChangeSel(this, strRECORDID, objEVENT);
Change	
Drop	
Ctrl Drop	
Adopt	
Spyglass	
Menu	
Reorder	

In Figure 4, the matrix1 component is selected in the Component Hierarchy palette (bottom left), and the Events Editor palette (bottom right) shows the event handlers for the matrix1 component. You can see that a Select event handler has been added to the matrix1 component. It executes the function `onChangeSel`.

The complete `logic.js` file for matrix1 is shown below. As you can see, `onChangeSel` simply creates a message object and publishes it.

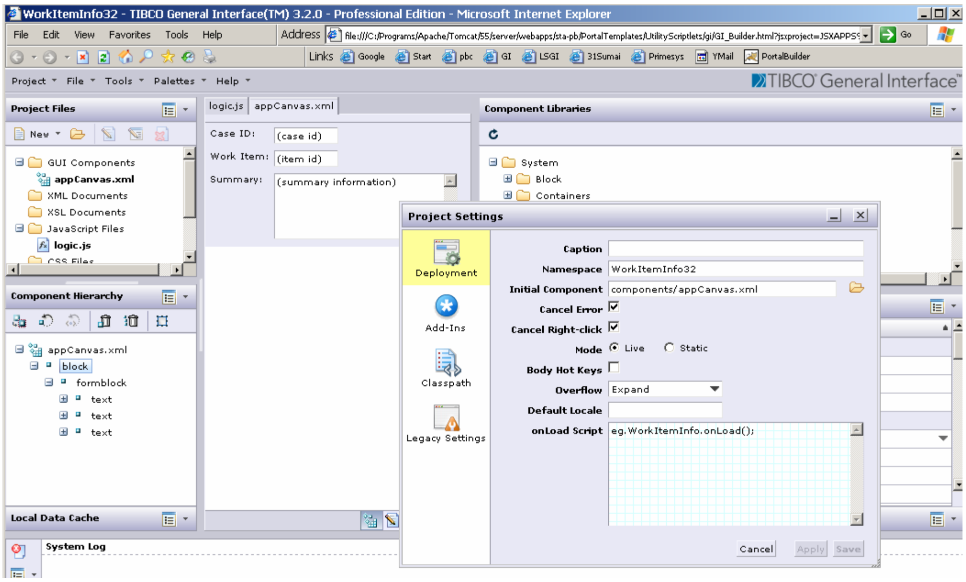
```
jsx3.lang.Package.definePackage(
  "eg.WorkItemList",                   // the full package name
  function(WorkItemList) {             // short package name, good
                                       // to use in this

    WorkItemList.getServer = function() {
      // should be the same as namespace in
      // Project -> Deployment Options
      return WorkItemList32;
    };

    /**
     * This method will publish to the topic. Any clients
     * subscribed to PageBus will receive the message.
     */
    onChangeSel = function(matrix, strId, guiEvent) {

      var selectedNodes = matrix.getSelectedNodes();
      if (selectedNodes == null)
        return;
      var objRecord = selectedNodes.getItem(0);

      var cid = objRecord.getAttributeNode("jsxid").getValue();
      var wi = objRecord.getAttributeNode("jsxtext").getValue();
      var su = objRecord.getAttributeNode("summary").getValue();
      var ev = { jss: "eg.WorkItemSummary",
                 jssv: "1.0.0",
                 caseid:cid,
                 workitem:wi,
                 summary:su };

      window.PageBus.publish("eg.workItem.onSelect", ev);
    }
})
```

Subscriber Example in TIBCO General Interface

This section demonstrates how to subscribe to messages using PageBus in General Interface™.

Figure 5 shows General Interface™ Builder with a component that listens for PageBus messages and displays the fields from these messages.

Figure 5 PageBus Subscriber in General Interface Builder



Case ID:	(case id)
Work Item:	(item id)
Summary:	(summary information)

Caption	
Namespace	WorkItemInfo32
Initial Component	components/appCanvas.xml
Cancel Error	✓
Cancel Right-click	✓
Mode	[✓ Live] Static
Body Hot Keys	
Overflow	Expand
Default Locale	
onLoad Script	eg.WorkItemInfo.onLoad();

In Figure 5, a function called `eg.WorkItemInfo.onLoad` has been added to the onLoad Script field in the Project Settings palette. Functions in the onLoad Script field are executed when the component is initialized. Therefore, you can insert a function that creates a PageBus subscription in this field.

The complete `logic.js` file for this General Interface component is shown next. As you can see, `eg.WorkItemInfo.onLoad` simply subscribes to the subject "eg.workItem.onSelect", and the function call includes the callback function called `eg.WorkItemInfo.onMessage`, which outputs the message content in the respective text fields.

Note that the short package name , `WorkItemInfo`, is used in the `logic.js` code instead of the full package name, `eg.WorkItemInfo`.

```
jsx3.lang.Package.definePackage(
  "eg.WorkItemInfo",                    // Full package name
  function(WorkItemInfo) {              // Short package name

    WorkItemInfo.getServer = function() {
// Should return the value of the Namespace field in the
// Deployment tab of the Project Settings palette.
      return WorkItemInfo32;
    };

    WorkItemInfo.onLoad = function() {
      // Subscribe to PageBus subject
      window.PageBus.subscribe("eg.workItem.onSelect",
                               WorkItemInfo.onMessage,
                               WorkItemInfo);
    };

    /*
     * This is the callback for the PageBus subscription.
     */
    WorkItemInfo.onMessage = function(subject, message) {

      // Get the text input that will be set to the case id
      // received in the message
      var inp = WorkItemInfo.getServer().getJSXByName("caseidin");
      // Now set its value
      inp.setValue(message.caseid);

      inp = WorkItemInfo.getServer().getJSXByName("workitemin");
      inp.setValue(message.workitem);

      inp = WorkItemInfo.getServer().getJSXByName("summaryin");
      inp.setValue(message.summary);
    };

})
```

Recommended Message Structure

This section describes the recommended message structure when publishing messages. While you can use other message structures, extensions provided by TIBCO or other vendors may take advantage of this common structure.

The recommended message structure is shown below:

```
{
  jss:  "org.pagebus.msg.Message",
  jssv: "0.9.1",
  head: {HeaderName1: HeaderValue1, HeaderName2: HeaderValue2, ...},
  body: yourMessageBody
}
```

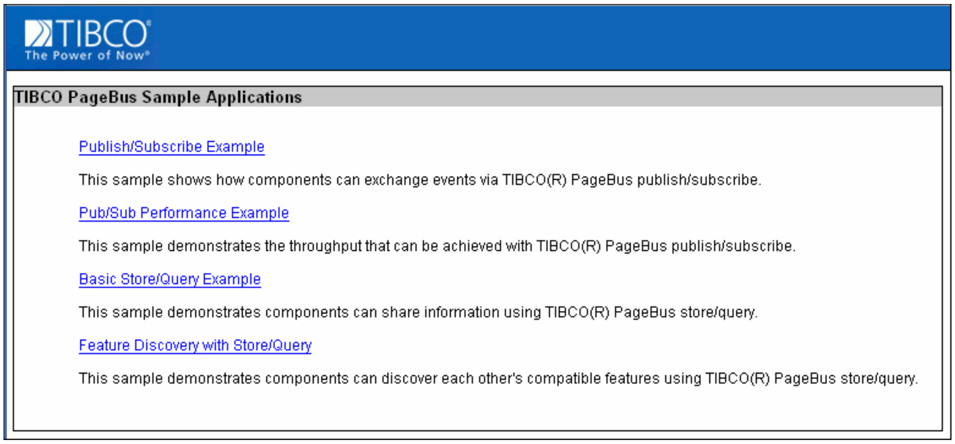
See Table 3 below for the format of each data member in the recommended message structure.

Table 3 Recommended Message Data Member

Member	Type	Comment
`jss`	string	The string "org.pagebus.msg.Message".
`jssv`	string	Version of this message class definition. The current version is "1.0.0".
`head`	object	Object that specifies a set of headers. This member is optional. If present, it must be an object. It can include any number of headers, and it also can be empty.
`body`	any	Message body. It is typically either an XML or JSON string or a JavaScript object.

Samples

Samples are found in the samples directory included with PageBus. The index.htm file in this directory provides links to the samples and descriptions of each:



TIBCO PageBus Sample Applications

Publish/Subscribe Example

This sample shows how components can exchange events via TIBCO(R) PageBus publish/subscribe.

Pub/Sub Performance Example

This sample demonstrates the throughput that can be achieved with TIBCO(R) PageBus publish/subscribe.

Basic Store/Query Example

This sample demonstrates components can share information using TIBCO(R) PageBus store/query.

Feature Discovery with Store/Query

This sample demonstrates components can discover each other's compatible features using TIBCO(R) PageBus store/query.

For more information and the latest samples, visit the PageBus resource center:

http://www.tibco.com/devnet/pagebus.

Chapter 5 Best Practices and Frequently Asked Questions

This section provides some guidance for developers and architects who intend to make use of TIBCO PageBus, as well as answering a number of common questions.

Topics

Application Design Strategy

TIBCO PageBus can help you build composite web applications that are much more modular than ones that are built with traditional approaches. Some additional techniques will help you maximize the reusability and manageability of these applications.

- Architect application components as coarse-grained, rich portlets or Ajax components rather than fine-grained controls.
- Each rich portlet or Ajax component should have a well-defined PageBus interface, providing an abstraction boundary between the component and other elements of the page and hiding the UI and implementation of the component. For example, if a rich portlet publishes messages on the subject `eg.customer.onSelect`, then it should not matter to subscribers if the messages were published in response to a button click, a node selection, or the receipt of an Ajax response from a web service.
- Subjects and messages should relate to the business or application level rather than the DOM level to ensure implementation-independence of portlets and Ajax components. Because messages relate to the application level, avoid making them correspond to DOM events. For example, `eg.customer.onSelect` is an appropriate subject name, but not `link.click`.
- Establish a shared concept model and set of event messages. Rich components from different developers and organizations can use these standard models and messages to interact with each other.
- Subject strings should *not* be scattered around the code. Try to isolate them and configure them as component properties. Doing so will allow you to modify them easily without having to track down every instance of your subject strings. It will also enable you to integrate components efficiently.
- If different application components initialize at different rates, a subscriber might not be initialized until after a given publisher has published a value. If this would result in improper behavior, then the publisher should probably use `PageBus.store` rather than `PageBus.publish`, and the subscriber should *both* subscribe to the notification subject and perform a `PageBus.query` to obtain the current value(s) for each matching subject.
- If a publisher needs to discover certain information about subscribers (see the "Feature Discovery Store/Query" sample), then the subscribers can use `PageBus.store` to advertise this information, and the publisher can use `PageBus.query` and `PageBus.subscribe` to discover the information and/or track changes.

PageBus Versus Raw JavaScript APIs

PageBus is an extremely useful tool, but it is not the only tool at your disposal. While interfaces between Ajax components and other page components are frequently event-based, there are also many cases in which more tightly coupled interfaces are required.

Use PageBus-based events when:

- Components may appear alone or in groups.
- Interactions are primarily event-based.
- One-to-many, many-to-one, or many-to-many communication is possible.
- Component implementations should be relatively decoupled.
- Brokers or proxy services may be used.

Use Raw JavaScript APIs when:

- Components always appear in a single grouping and configuration.
- Interactions are primarily request-response.
- Interactions are tightly-coupled and one-to-one.
- Application components are using fine-grained, internal events. *Internal events* are events that are propagated only within a single application component and are not exposed or visible outside of that component.

Use the Two Approaches Together

Change notifications can use the publish-subscribe interface in PageBus, while queries for detailed data may use either a common cache or a request-reply API. For example, a graph can be told to plot a set of points using PageBus, but at the same time it may be allowed to retrieve additional information about each point by calling a JavaScript function.

Performance Considerations

This section briefly describes how PageBus performs, and the overhead in different browsers. Design considerations are also discussed.

PageBus Performance

Table 4 shows the approximate time it takes PageBus to deliver 100,000 messages from one publisher to one subscriber on a fast 2.33 GHz laptop computer, where the subscriber does nothing with the messages that it receives. It shows that, on average, PageBus takes approximately 30 microseconds to deliver a message from the publisher to the subscriber.

Table 4 PageBus Performance Data

Browser	Time (sec.)
Firefox	3
Internet Explorer 6	4
Opera	2

However, even if users could actually see 100,000 changes per second, browsers can't render data to the screen this quickly. Mozilla Firefox 1.5 renders 5,000 values in about 3 seconds on the same computer. So the cost of the render operation dwarfs the cost of a PageBus message delivery. If you use TIBCO PageBus wisely, the performance limiting factor in your applications will be your applications' rendering or AJAX communication logic rather than the performance of the PageBus.

Design Considerations

Assuming that your application involves both business logic and sophisticated rendering, the limiting factors in an Ajax application's performance will almost certainly be the amount of rendering *and* the amount of network I/O that the application performs. Performing unnecessary rendering and network I/O can drastically impede an Ajax application's performance.

You should consider the following factors when planning the message publishing rate in your web application:

- Browser's performance (speed of rendering to screen)
- Amount and complexity of rendering logic in the callback function

- Amount of network I/O logic in the callback function
- Network delays from I/O logic
- Application performance with concurrent processes
- Application performance on slow computers

Security and Mash-Ups

Any time you allow JavaScript from untrusted parties to run on a web page that contains sensitive information, there is a potential security risk. This is because JavaScript on a web page can access any JavaScript function on that page, traverse and update the DOM objects, and potentially communicate with any server using Ajax.

Use of PageBus neither increases nor decreases the potential security threat associated with running untrusted JavaScript code.

When building mash-ups, you should avoid running untrusted JavaScript code when the page contains sensitive information. You may want to think of each page as a security zone, and control the types of Ajax components that are allowed in that zone.

Subject Naming

We recommend that you use Java package notation when designing subject names. For example:

```
com.tibco.portal.foo.bar
com.cisco.etc.etc
org.myorg.etc.etc.etc.etc
```

This technique avoids subject name clashes between organizations.

Frequently Asked Questions

This section answers frequently asked questions.

How do we Troubleshoot Applications using PageBus?

It is very difficult to troubleshoot monolithic applications. By enabling the development of more modular applications and cleaner interfaces, PageBus can help you to create applications that are easier to troubleshoot. In addition, the specific architecture of PageBus makes interfaces based on PageBus inherently easier to troubleshoot than "raw" scripting interfaces.

To troubleshoot your applications based on PageBus, you can take advantage of the fact that PageBus allows for any number of subscribers. For example, you can add a "Logger" subscriber that listens for messages on some subjects and logs them to a table on the page. Such loggers can be simple or elaborate. They are typically used only in development and testing phases and are removed later, since they do not affect how the other components operate.

How do we Test Applications using PageBus?

You can use the same method described in the previous section to test your components based on PageBus by attaching record-and-playback components to your application components in order to confirm that message traffic generated by the components complies with expectations.

How do we Integrate PageBus with TIBCO Ajax Message Service?

TIBCO Ajax Message Service is pre-integrated with TIBCO PageBus. Please see the Ajax Message Service product documentation for relevant information.

Can Subjects be Configured?

Yes. While PageBus subjects can be created on the fly by application components just prior to subscribe and publish calls, some application components may obtain subjects from application-specific configuration objects.

Does PageBus Require a Portal?

No. While PageBus may be packaged with TIBCO PortalBuilder or other TIBCO software, it does not require a portal in order to function.

Rich Internet application (RIA) components that interact using PageBus do not need to be portlets. However, PageBus provides an ideal browser-based, event-based communication mechanism for rich portlets. The ability of the PageBus to support zero to many subscribers or publishers on a subject is very useful in a portal page that contains an unspecified number of related portlets.

Does PageBus Require TIBCO General Interface?

No. PageBus does not require TIBCO General Interface in order to function.

TIBCO PageBus BSD License

Third Party Software License Agreements

The following are the software licenses for the Third Party Software provided in connection with the software.

Apache License, Version 2.0, January 2004

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.
"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.