Manual

Application:	GWTEventService
Software version:	1.2
Document version:	1.2
Last document update:	2011-10-15

Project home:	http://gwteventservice.googlecode.com/
Project owner:	strawbill UG (haftungsbeschränkt)
Contact:	info@strawbill.com

Contents

1 Introduction

GWTEventService is an event-based client-server communication framework. It uses GWT-RPC and the Comet / server-push technique to publish events from the server to the clients or to exchange events between the clients. Chapter 9.2 demonstrates how the communication with Comet / server-push works. The client side offers a high-level API with opportunities to register listeners to the server like to a GUI component. Events can be added to a domain on the server side and the listeners on the client side get informed about the incoming events. The server side is completely independent of the client implementation and is highly configurable. Domains can be defined to decide which events are important for the different contexts. A domain can be handled as a unique key for a context.

It's not nearly as difficult as it is proclaimed to be.

1.1 Advantages

- Encapsulation of the client-server communication
- High-level API with listeners and events
- Easy to use
- Only one open connection for event listening
- Reduction of server calls
- Reduction of connection peaks
- Events are returned directly when the event has occurred (instead of polling)
- Events are bundled to reduce server calls
- Server-side event filtering to reduce server calls
- Based on the GWT-RPC mechanism
- Automatic timeout recognition and handling with listening opportunities
- Extensible architecture

These advantages are not guaranteed for all cases and could also be disadvantages in some special cases compared with polling or other technologies.

2 Configuration

To get an application running with GWTEventService, the gwteventservice.jar must be included in the classpath. The gwt-xml must include an entry to inherit "*de.novanic.eventservice.GWTEventService*".

Example

```
<module>

    <inherits name='com.google.gwt.user.User'/>
    <inherits name="com.google.gwt.user.theme.standard.Standard"/>

    <inherits name="de.novanic.eventservice.GWTEventService"/>

    <entry-point class='de.novanic.gwteventservice.demo.conversationapp.client.DemoConversationAppUI'/>
    <servlet path="/conversationport" class='de.novanic.gwteventservice.demo.conversationapp.service.ConversationServiceImpl'/>

    <stylesheet src="DemoConversationAppUI.css"/>

</module>
```

If you want to execute the tests, you need to include the libraries JUnit 3.x or greater (http://www.junit.org) and EasyMock 2.x or greater (http://www.easymock.org). For more information about building GWTEventService yourself and setup the project (with sources), please take a look in the developer guide of GWTEventService.

To configure GWTEventService you can use “eventservice.properties” or define servlet-parameters with the web-descriptor. The location of the properties file must be attached to the classpath of the server side application. The default values and the description of the times can be seen in the following table.

Alternatively, a configuration loader can be defined if you want to load the configuration from another source. The interface “*ConfigurationLoader*” must be implemented and must be attached as a custom loader. You can take a look in the developer guide of GWTEventService to experience more about the usage of custom configuration loaders.

Overview about the configurable times

Time in milliseconds / property	Default value	Description
Min. waiting time “time.waiting.min”	0 (ms)	Time to wait at minimum (even when events have already occurred)
Max. waiting time “time.waiting.max”	20000 (ms)	Time to wait at maximum (canceled when an event occurred)
Timeout time “time.timeout”	90000 (ms)	Time till timeout (when the time is reached, the user will be removed from listening for events)

With that configuration possibilities, the server side behavior can be influenced very flexibly. If desired, it is even possible to run GWTEventService in polling mode if min. waiting time is set to a value greater than zero and max. waiting time is set to zero (server side polling mode).

The properties can be written equal to the name in the table above (without the quotes) or additional with the prefix “eventservice.”.

2.1 Connection strategies

At this version two connection strategies are supported by GWTEventService: Long-Polling and Streaming. The two connection strategies are compared in chapter 9.2 to highlight the various advantages. Long-Polling is configured as the default connection strategy.

Connection strategy property	Connector class
eventservice.connection.strategy.server.connector	Long-Polling (default): de.novanic.eventservice.service.connection.strategy.connector.longpolling.LongPollingServerConnector Streaming: de.novanic.eventservice.service.connection.strategy.connector.streaming.StreamingServerConnector
eventservice.connection.strategy.client.connector	Long-Polling (default): de.novanic.eventservice.client.connection.strategy.connector.DefaultClientConnector Streaming: de.novanic.eventservice.client.connection.strategy.connector.streaming.GWTStreamingClientConnector

2.2 Shared sessions

Multiple / shared sessions are necessary when the access to your application must be possible with more than one browser instance per client (for example when your application must be controllable by various browser tabs). The shared session support can be activated for GWTEventService but isn't defined as default because the sending of user-specific events and the configuration of event filtering is more

complicated and not downward-compatible. The reason is that the client id has to be transferred with every request and the requests of your own services can not and shouldn't be controlled by the framework.

The support of shared sessions can be activated with the following properties.

Connection id generation property	Connection id generator
eventservice.connection.id.generator	de.novanic.eventservice.service.connection.id.SessionExtendedConnectionIdGenerator

3 Get started / Usage Guide

Here you can find a small tutorial to see how to get started with GWTEventService. Some UML diagrams to understand and get an overview about the API can be found in chapter 9. The get started instructions should help you to define your first domains and to write your first own listener and events. It's a little bit work to define all the classes and to adjust the client and server side classes, but you should get a clean architecture and design. Let's start to think, define and write.

3.1 Think of your needed events

You may know listeners and events of different GUI frameworks like Swing or AWT. The client API of GWTEventService works nearly the same way, but with the difference that you are registering the listeners imaginable to the server. That listening method can theoretically resolve all polling calls to the server. You need to think of and define the events which your client side has to know about. For example a conversation application will at least need a *UserJoinEvent*, *UserLeaveEvent* and a *NewMessageEvent*. The events can be simple Beans or POJOs and can theoretically get all serializable values which are available on the server side. The *Event* interface is only a marker interface and there is logically no default implementation. The domain can't knowingly be got from the *Event*, because the listener shouldn't be aware of the different domains. If the listener has to differentiate between the different domains, it is a clear signal that you should define different listeners. But when you really need the domain in your events, nothing stops you defining your own abstract event... To learn more about events you can take a look into chapter 6 (Events), but that isn't required to understand that get started guide.

3.2 Define your domains

Domains are needed to distinguish the various contexts. Domains can be understood as a unique key / id and it is nothing magic or complicated. For example you could have a *UserJoinEvent* which is triggered when a user joined a game and you want to recognize that in a game list and on a waiting page. In that case you could define a “game-domain” where all events important for the game go to (something like *GameStartEvent* and *UserLeaveEvent*, etc.). Now you have the idea to write a conversation plugin for your application. In that case you will also need a *UserJoinEvent* and *UserLeaveEvent*, but it shouldn't be handled the same way like in the game context. To re-use the two events (or also the listener if it is factually possible) you can define another domain like “conversation-domain” to avoid conflicts between the same events in different contexts. More information about domains can be found in chapter 5 (Domains).

3.3 Write your listeners and events

Now the interesting part: How can I make it work? First you will need to write your events like defined in 3.1. The events must implement the *Event* interface (*de.novanic.eventservice.client.Event*), must be available to the client- and to the server side and must be serializable, because the events are transferred from the server to the client side. A simple project structure to reach that can be seen in the chapter 3.4 (project structure). In the most projects, the events will be added from the server side, because in the most cases the events are known and triggered by the server and the events should be distributed to the clients, but it is also possible to add events directly from the client side (see chapter 3.4 – Add events). On the server side events can be added via an *EventExecutorService* (method *addEvent(Domain, Event)*). An *EventExecutorService* can be got from the *EventExecutorServiceFactory* or if you have a servlet, you can extend from *RemoteEventServiceServlet* which is an implementation of *EventExecutorService*. That's it for the server

side.

The listeners must only be accessible by the client side, because the server doesn't need to know anything about listeners. The only very logical method you need to implement is the “apply” method of the *RemoteEventListener* interface (*de.novanic.eventservice.client.listener.RemoteEventListener*). The apply method gets an *Event* as a parameter and your logic must be able to decide how to handle the event. It's a good practice to define methods in a listener interface like “*onUserJoin*” and “*onUserLeave*” which can be implemented in the real implementation of the listener. An adapter class or abstract implementation of the listener can offer the apply method implementation. It is often needed to differ the events via “instanceof” checks. That is planned to get improved with a later version of GWTEventService.

Complete code examples can be found in chapter 14.1 (Appendix: Code-Examples). Here is an example of an implemented “apply” method:

GameUserListenerAdapter

```
public abstract class GameUserListenerAdapter implements GameUserListener
{
          public void apply(Event anEvent) {
                    if(anEvent instanceof UserJoinEvent) {
                              onUserJoin((UserJoinEvent)anEvent);
                    } else if(anEvent instanceof UserLeaveEvent) {
                              onUserLeave((UserLeaveEvent)anEvent);
                    }
          }

          public void onUserJoin(UserJoinEvent aUserJoinEvent) {}

          public void onUserLeave(UserLeaveEvent aUserLeaveEvent) {}
}
```

Now you have the events and the listeners. Maybe you recognized that you need a Domain (*de.novanic.eventservice.client.event.domain.Domain*) to add an event. This is the last step: The registering of listeners and adding of events with domains. A domain can be got from the *DomainFactory* (*de.novanic.eventservice.client.event.domain.DomainFactory*) with a unique *String* (domain name). It would be a good idea to hold the domain name or the Domain as a constant, available for the client- and server side, to ensure that you access the right domains.

RemoteEventService (*de.novanic.eventservice.client.event.RemoteEventService*) can be used to register and deregister listeners (client side). The equivalent methods are named “*addListener*” and “*removeListener*”. All methods have an optional callback parameter, if you need to know when the change is really active.

Something special and optional is the *EventFilter* (*de.novanic.eventservice.client.event.filter.EventFilter*). *EventFilter instances* can be added from client side and are processed on server side. With an *EventFilter* the server side can decide if an event is really important for the user or not. Please take a look into the chapter 8 (server side event filtering) when you want to learn more about filtering of events on the server side.

3.4 Add events

Events can be added from the client- and from the server side. The two different ways are described in the following two sub-chapters.

3.4.1 Add events from the server side

On the server side, events can be added via the *EventExecutorService* which can be got from the *EventExecutorServiceFactory*. If you are using a servlet, you may extend from *RemoteEventServiceServlet* which is an implementation of *EventExecutorService* and offers all methods to add events directly. The *EventExecutorService* can be initialized per client (with the client id) to execute all actions directly with the client. This client id is used to add user-specific events and to register *EventFilter* instances for the client and

isn't necessary to simply add domain global events. To initialize it with the client id, the client id could be transferred via the *ClientHandler* or the request (*HttpServletRequest*) can be used for the initialization (as a parameter for *EventExecutorService#getEventExecutorService()*).

Code snippet	

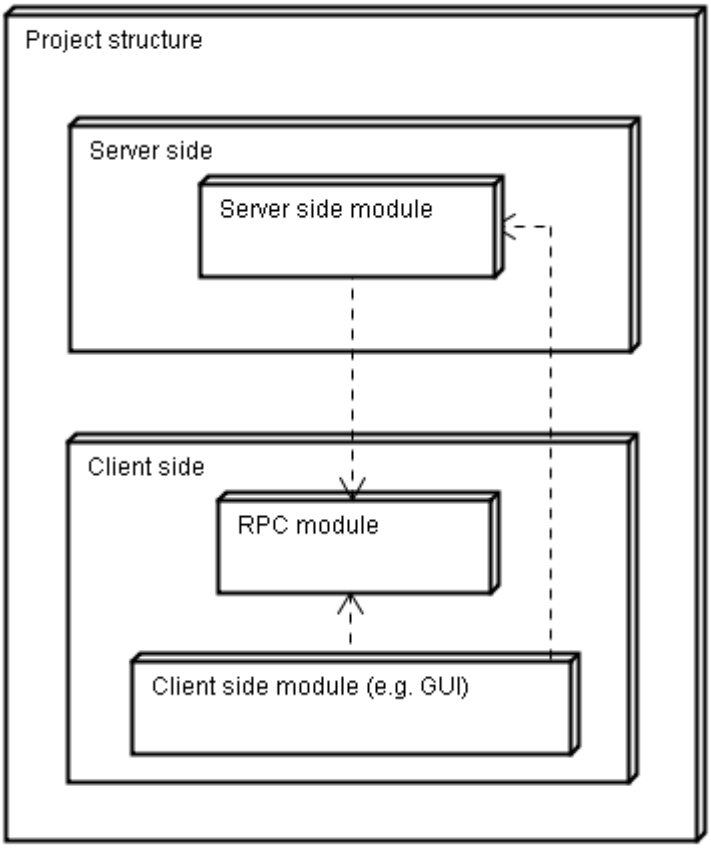
```
EventExecutorServiceFactory theSF = EventExecutorServiceFactory.getInstance();
EventExecutorService theService = theSF.getEventExecutorService(<client id>);
theService.addEvent(<Domain>, <Event>);
```

3.4.2 Add events from the client side

It is possible to add events directly from the client side. The RemoteEventService interface which is also used to add listeners offers a *addEvent(...)* method which expects a domain and an event. User-specific events can be added with the usage of the user-specific domain which is described in chapter 5.1 (User-specific domain).

3.5 Project structure

The architecture of GWTEventService requires that *Event* classes can be accessed from the server and from the client side. That doesn't cause any problems, when your project does only consist of one module, but when the client and server side code is split into different modules, you need to have at least one module between the server and client side code. When your project currently doesn't support that requirement you could derive your project structure from the following structure. It is divided into three modules. The server side module can be compiled completely independent from the client side modules (without any GWT limitations). The client side module could contain your client side GWT components (GUI) and the listeners. The RPC module is between the server side and the client side module and must also be compatible to the JRE emulation library of GWT. There aren't any dependency cycles with that structure of three modules.



Example reference
The displayed project structure can be seen in the demo project “DemoConversationApp”.

4 Connection-Handling

The connection-handling is completely solved internal of *RemoteEventService* (*de.novanic.eventservice.client.event.RemoteEventService*). GWTEventService creates only one connection to the server for event listening. The connection is opened when a listener is added and closed when the last listener is removed. Therefore it's important that listeners are removed when they become unneeded. When a listener is never removed, the connection is still open as long as the client is active (until timeout or browser closed). The client logs activate and deactivate messages to control the connection-handling. There are different ways to remove an unneeded listener: The listener can be removed directly, all listeners of a domain can be removed (the listener object isn't needed) or all listeners can be removed. Look at the *RemoteEventService* in the class diagram (chapter 9.1) and the corresponding JavaDoc for more information. For more information take a look at the description of the various connection strategies in chapter 9.2.

Technical hint: GWTEventService executes an initialization server call to load the configuration and to avoid session handling issues.

4.1 Automatic reconnect

GWTEventService starts reconnect attempts automatically when a connection problem occurred. The number of automatic reconnect attempts can be configured with the property “eventservice.reconnect.attempt.count”. For compatibility reasons this feature is deactivated by default (default value is zero), but the recommended setting is value “2” (two reconnect attempts will be executed on a connection failure). The recommended setting is included within the default eventservice.properties file which may be used to setup new projects.

5 Domains

Domains are a central element of GWTEventService to divide the registered users into the various contexts of interest. That helps to make an application with various contexts more scalable and defined events can be re-used. A domain can be understood as a unique id and can be simply created with one call to the *DomainFactory*. The domain is used on the client side to register listeners to the domain and on the server side to add events to the domain. That is why a domain should be held as a constant and must be accessible from the client and from the server side. The *String* for the domain creation or the *Domain* itself should be held as a constant to ensure that you access the right domains.

Code snippet	

```
Domain theDomain = DomainFactory.getDomain("your_context");
```

5.1 User-specific domain

It is also possible to use GWTEventService without domains. In that case the user-specific domain (DomainFactory#USER_SPECIFIC_DOMAIN) can be used to register listeners and to add events. The user-specific domain makes sense when the application or some parts of the application only triggers user-specific events. In that case the server side can add user-specific events (no domains are required for user-specific events) and on the client side a listener can be registered with the user-specific domain. User-specific events are described in chapter 6.1 (user-specific events).

6 Events

An event implementation must implement the *Event* interface (*de.novanic.eventservice.client.Event*) and must be available to the client- and to the server side. A simple project structure to reach that can be seen in

chapter 3.4 (project structure). The *Event* interface is only a marker interface and logically there is no default implementation available. The events can be simple Beans or POJOs and can theoretically get all serializable values which are available on the server side. The event itself must also be serializable itself, because it is transferred from the server to the clients. There are a few special rules for serializable types (see http://code.google.com/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideSerializableTypes for more information). Common made mistakes are for example that the class doesn't have a default constructor (no arguments) or that final variables are not transferred.

6.1 User-specific events

An event can also be added user-specifically instead of domain-specifically. This causes that the event can be added on the server side and will only be received by the specified user / client. User-specific events can be received with every listener without considering the domain of the registered listener. The listener to receive user-specific events can be added with the user-specific domain. The user-specific domain is described in chapter 5.1 (user-specific domain). The user id which is required to add user-specific events is equal to the session id.

7 Unlisten / timeout listening

GWTEventService cleans-up the internal user management system itself, when a timeout occurs or when all listeners are removed. That information is provided with *UnlistenEvent* objects and applications which are using GWTEventService can catch that information with an *UnlistenEventListener*. An *UnlistenEventListener* can be registered the same way like other *RemoteEventListener* implementations, with the difference that *RemoteEventService#addUnlistenListener(...)* must be called instead of *RemoteEventService#addListener(...)*. An unlisten event listener is defined globally and not domain specifically, to catch also global unlisten events like a connection timeout which could affect more than one domain at the time. There are various scopes, instead of domains, to define or specify the interests of the unlisten listener. One part of the main unlisten concept is, that only unlisten events of users/clients can be received, when the listening user is at least registered to one same domain like the removed user. A more global scope shouldn't and mustn't be required, because it would cause too many unimportant unlisten events.

7.1 Scopes

There are three different scopes to customize the event occurrences (*local*, *unlisten* and *timeout*). The default scope is *unlisten* and can be changed with an optional parameter of the registration of an unlisten listener. The scopes are built on different levels. Scopes on higher levels contain all event occurrences of scopes on lower levels.

Here is a short overview about the various scopes. Details about the single scopes are described in the sub-chapters.

Overview about the various scopes

Scope	Level	Contained Scope	Description	Server calls
LOCAL	1	-	Caused by connection issues between client and server. Unlisten events of the local scope are only generated for the own client and not published to other users, but a timeout will be recognized on the server side and published for the timeout scope.	no
UNLISTEN	2	LOCAL	Caused by calls of unlisten or by removing listeners (for single domains and for all domains).	yes
TIMEOUT	3	LOCAL, UNLISTEN	Caused by timeout detections on the server side, for example when the browser of a user/client has been closed.	yes

7.1.1 Local

When the unlisten listener is registered with the local scope, unlisten events will only be triggered when the own connection is interrupted. That could for example occur when the connection to the server is lost, caused by a technical network issue or when the server is currently terminated or restarted. Unlisten listeners registered with that scope can simply be tested by pulling the network connector and should get an unlisten event directly. An Unlisten listener with the local scope isn't registered to the server side and causes no server calls because it inspects only the own connection of the client side. When the user/client doesn't reactivate the connection before the timeout time is exceeded, the user/client will be removed from the server side, a timeout will be recognized and an unlisten event for the timeout scope will be produced.

7.1.2 Unlisten

The unlisten scope contains the local scope and offers additional unlisten events which are produced by unlisten calls or by removing listeners. The unlisten event contains the user id and the unlistened domain. When all listeners are removed by one call (domain global unlisten), all unlistened domains are contained. Instead of the local scope, the unlisten event will be produced on the server side and published to all users/clients who have an unlisten listener with the unlisten scope registered.

7.1.3 Timeout

The timeout scope contains the unlisten scope and the local scope, and offers additional unlisten events which are produced by timeouts. The timeout is detected on the server side, when the configured timeout time is exceeded for a user/client. The generated unlisten event will be published like the unlisten scope, to all users/client who have an unlisten listener with the timeout scope registered. When the user/client was registered to more than one domain, the unlisten event contains all unlistened domains.

7.2 Custom unlisten events

For every user/client one custom unlisten event can be registered. That unlisten event must implement the interface *de.novanic.eventservice.client.event.listener.unlisten.UnlistenEvent* or must extend from the *DefaultUnlistenEvent* and can be registered with an optional parameter of *RemoteEventService#addUnlistenListener(...)*. That unlisten event will be held on the server side and sent to all users/clients where an unlisten listener is registered. A user/client can only send one unlisten event before the user/client information is cleaned-up on the server side. That is why only one custom unlisten event is required. When no custom unlisten event is registered, a generic unlisten event with the user id and the unlistened domains will be published. The custom unlisten event can contain any information which is serializable, like all other events of GWTEventService.

7.3 Rules

- Unlisten events which are published on scopes of lower levels will also be sent to scopes of higher levels.
- Unlisten event listeners won't be removed automatically when all event listeners are removed for single domains, because a listener for a single domain could only be removed temporarily and the unlisten listener couldn't be re-registered automatically after that. The unlisten listener will be removed when all listeners are removed (*RemoteEventService#removeListeners(...)*) or must be removed manually with *RemoteEventService#removeUnlistenListener(...)*.

8 Server side event filtering

GWTEventService provides a mechanism to filter events directly on the server side when the events occur. That is very useful in order to specify the interests more precisely within a domain. That task is processed by *EventFilter* objects. For every client/user domain combination only one *EventFilter* can be registered, but they are chainable appropriately to the filter pattern. That means *EventFilter* objects can be attached to other *EventFilter* objects to build filter chains which are concurrently processable. A domain defines the compact context and an *EventFilter* has the task to filter the events of a domain for the client/user depended interests.

Maybe some events that occur within a domain are only important for single users/clients. The advantage of server side event filtering is that the events get filtered directly on the server side which can save unnecessary server calls / connection cycles and can preserve the bandwidth. An *EventFilter* can be specified for every user/client domain combination. That means, the users/clients within a domain can get different *EventFilter* objects and every user/client can have different *EventFilter* objects according to the various domains.

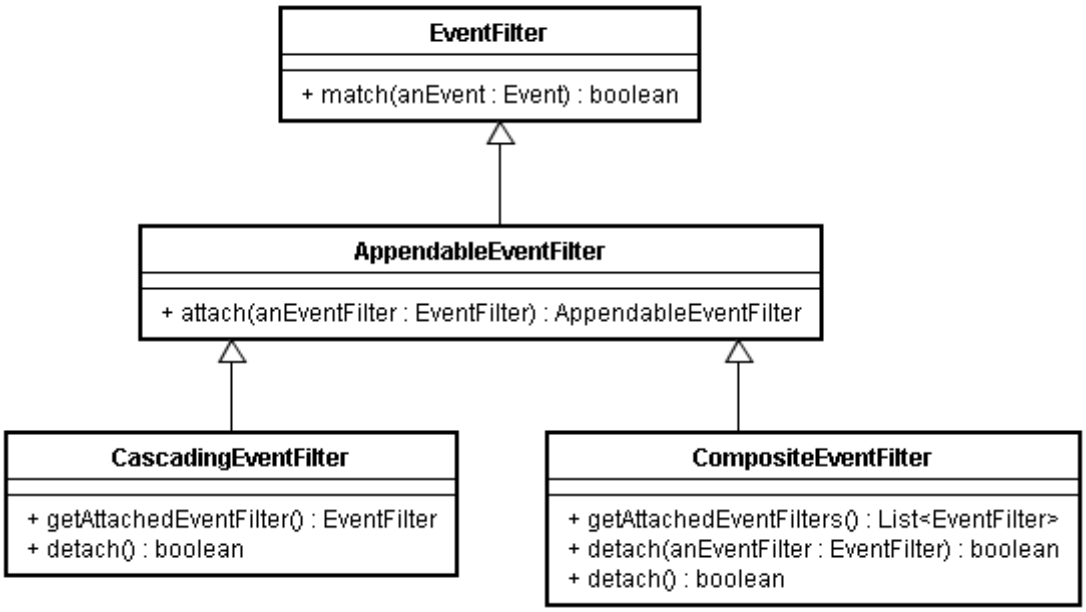
Example reference
The DemoConversationApp defines a conversation domain and filters the events for conversation channels within the domain. That is used to guarantee that the registered user can only receive messages from users of the same channel.

8.1 Registering of EventFilter objects

An *EventFilter* can be set from the client and from the server side. The *RemoteEventService* of the client side offers a method to register an *EventFilter* separately (*RemoteEventService#registerEventFilter(...)*). The *EventFilter* can also be set with an optional parameter directly when a listener is registered to a domain. That is the better option, when it is possible to create the *EventFilter* at the time, because it saves a server call compared to the separately method. Of course deregister methods are also available according to the register methods.

The server side classes *EventService*, *RemoteEventServiceServlet*, *EventExecutorService* and *EventRegistry* provide methods to register *EventFilter* objects to user domain combinations.

8.2 EventFilter-API overview



8.3 Chaining of EventFilter objects

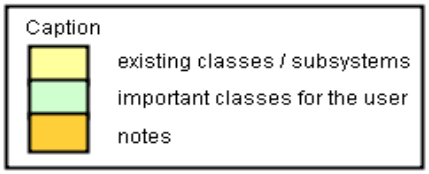
There are two types that support the chaining of *EventFilter* objects. When an *EventFilter* implementation implements the *AppendableEventFilter* interface, another *EventFilter* can simply be attached to that *EventFilter*. The attached *EventFilter* will be returned as an instance of *AppendableEventFilter and* another *EventFilter* can be attached again, to build a filter chain. That option can save memory, because the filter chain is built with memory saving references, but has a few limitations to the usability (attaching and getting

of attached *EventFilter* objects) and concurrent processing isn't effectively possible. The class *DefaultEventFilter* is the default implementation of *AppendableEventFilter* and *CascadingEventFilter*. When an *EventFilter* class extends from *DefaultEventFilter* and the appending of *EventFilter* objects is needed, the *EventFilter* implementation must call the overridden *DefaultEventFilter#match(...)* method (with *super.match(...)*).

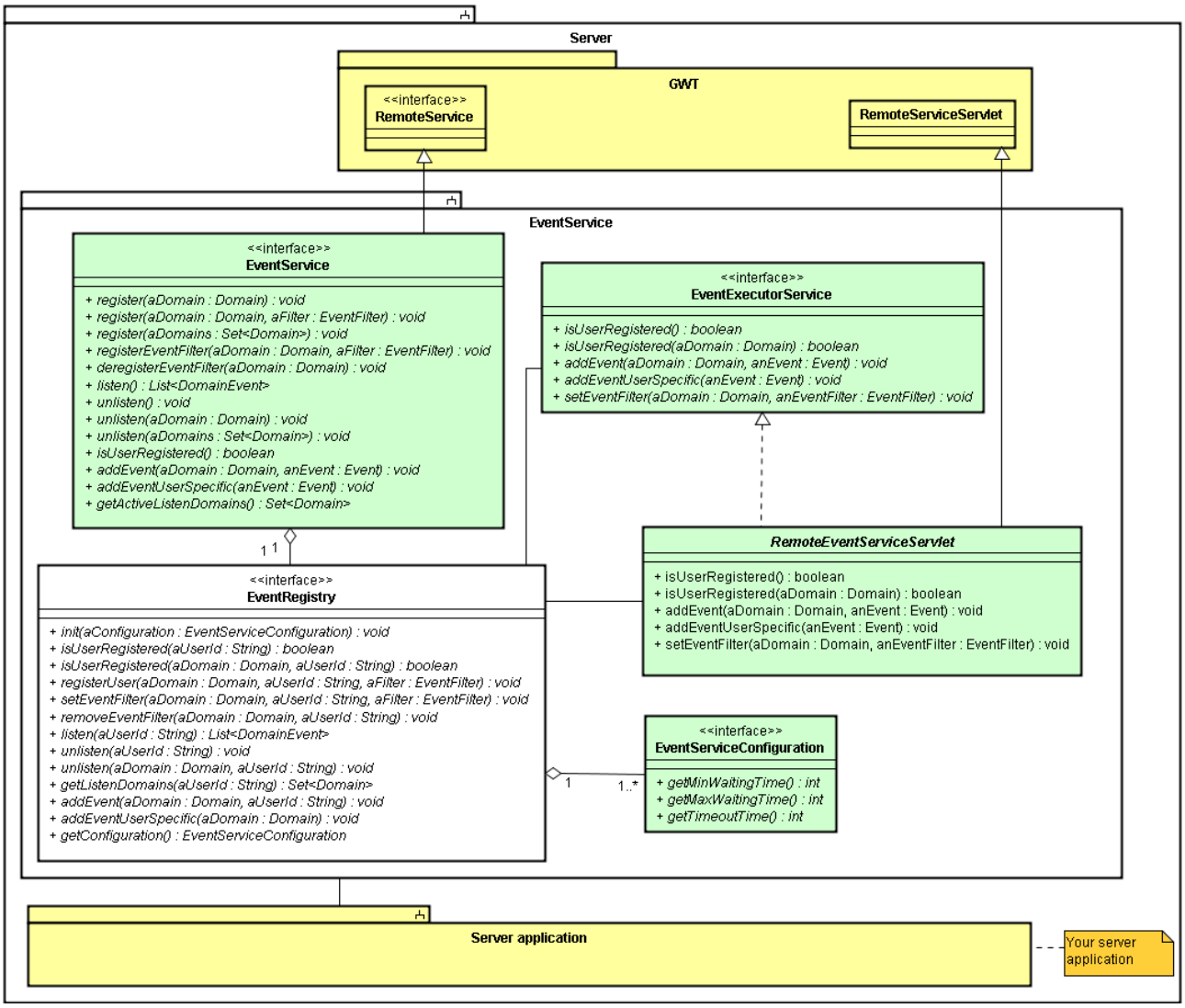
The other option is to use a *CompositeEventFilter*. A composite filter chain has the advantages, that it is easy to build by using the *EventFilterFactory* (*EventFilterFactory#connect(...)*) and it is easy to access the attached *EventFilter* objects by *CompositeEventFilter#getAttachedEventFilters()*. Additionally the attaching *EventFilter* objects must only implement the *EventFilter* interface instead of *AppendableEventFilter*. That solution has the disadvantage that a few more memory is necessary, caused by the internal structure and the top level composite *EventFilter* must be used for executing all sub *EventFilter* objects. The default implementation is *DefaultComposteEventFilter*, but a composite *EventFilter* can also be created with the above described *EventFilterFactory*.

9 API - Overview

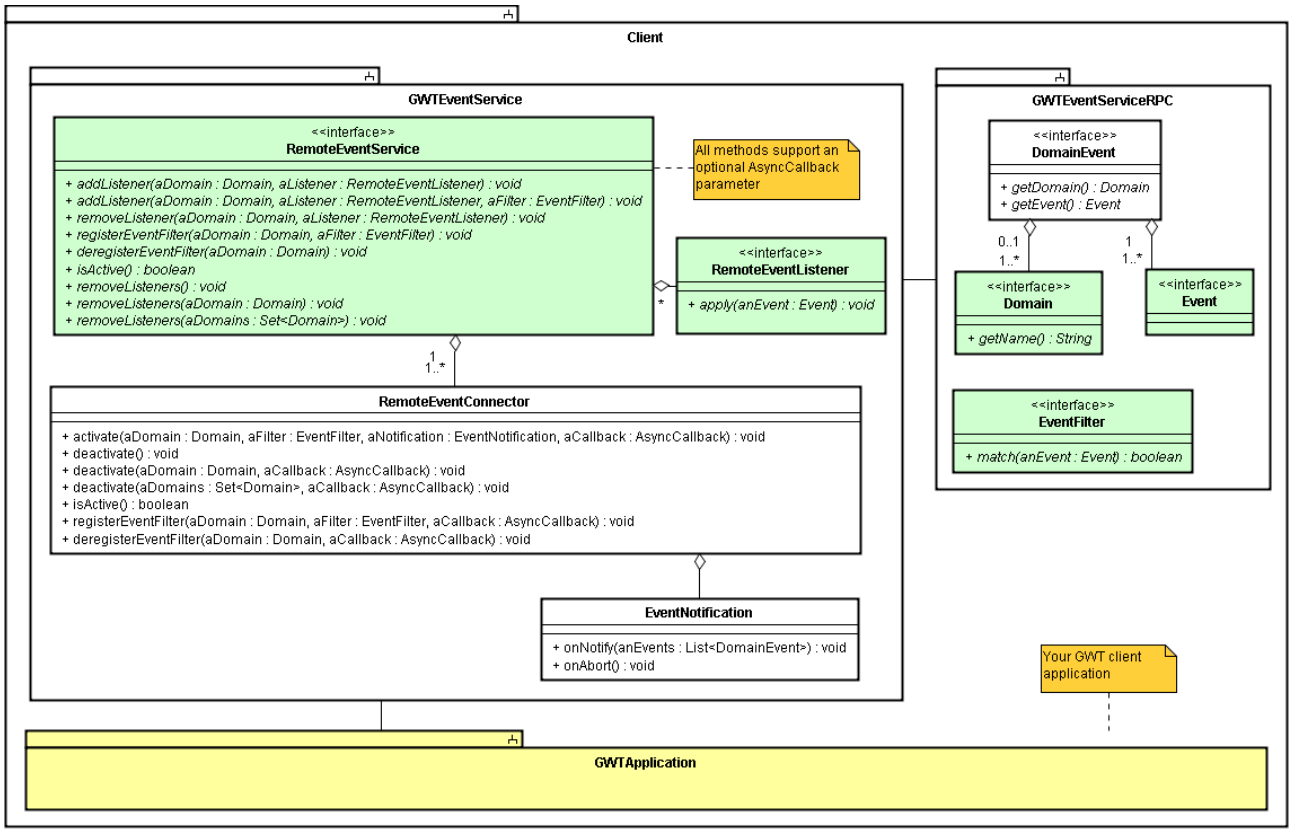
9.1 API class diagram



9.1.1 Server side



9.1.2 Client side



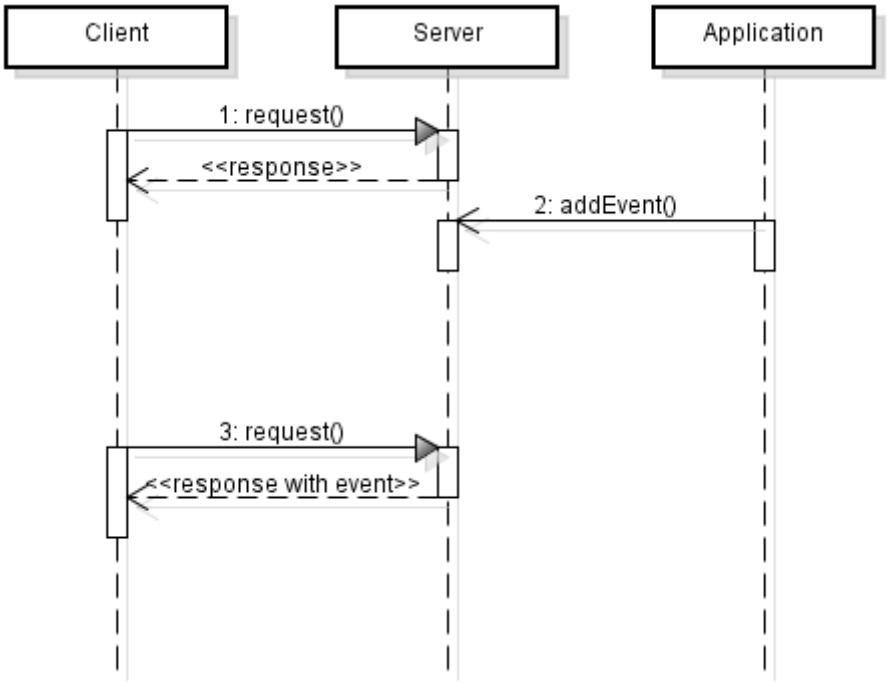
9.2 Polling compared to Comet / server-push

Polling is mostly used in simple web applications which need to check some server side states. Polling is an easy, but inefficient way to check the server side for occurred events, because polling means that a request is sent every *x* seconds to the server. Therefore, events are recognized lately and a lot of requests are sent from every client.

Comet / server-push is a solution to reduce the expensive server calls and to send the occurred events directly back to the client (without delays). There are two popular strategies to realize server-push. The first one is called long-polling and offers a maximum of client and server compatibility. The other strategy which is also supported by GWTEventService is streaming. Streaming requires client/browser specific implementations, but is more accurate when a lot of events are sent continuously (for example when more than one event per second per client is sent). Both strategies are supported by GWTEventService and long-polling is defined as the default strategy of GWTEventService due to the maximum of compatibility. Read chapter 9.2.3 for more information according streaming and chapter 2.2 to see how to activate streaming, if your decision is the streaming strategy.

9.2.1 Event listen strategy - Polling

The following sequence diagram shows the workflow of regular polling. The client sends a request in a defined interval to check whether an event is available and the result is returned directly. Events which occur between the requests / within the interval are received lately, caused by the fixed check interval. Another disadvantage could be that more requests are sent than necessary.

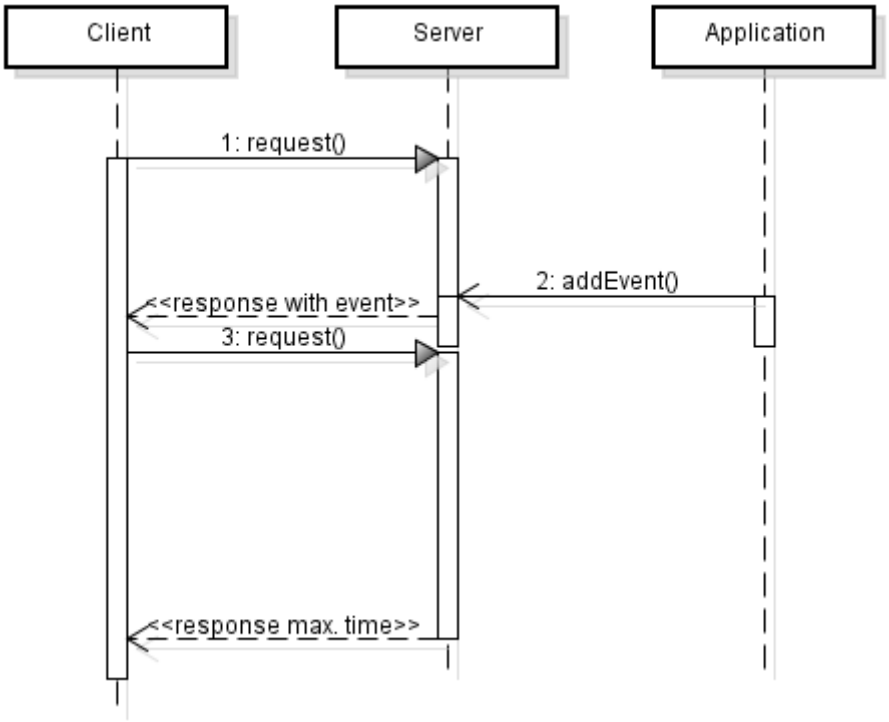


9.2.2 Server-push strategy – Long-Polling (default)

When Long-Polling is configured (default), the client sends a request to the server, the server waits till an event occurs and sends the event directly back to the client via the response. After that, the client starts the next request directly, to activate the event listening again. When events are occurred during the connection cycle, all events are returned directly via the response and the cycle starts again. GWTEventService allows to configure a max. waiting time to limit the server side waiting sequence to avoid possible timeout detections.

The advantages and disadvantages are easy to notice. It offers a maximum of client and server compatibility, because clean and normal request and response cycles are used and the client recognizes the server just like a slow server. Therefore there are no client- or server-specific implementations required. The connection cycles could be a disadvantage in some projects when a lot of events must be sent to the same clients within a few seconds, because that causes continuously connection cycles.

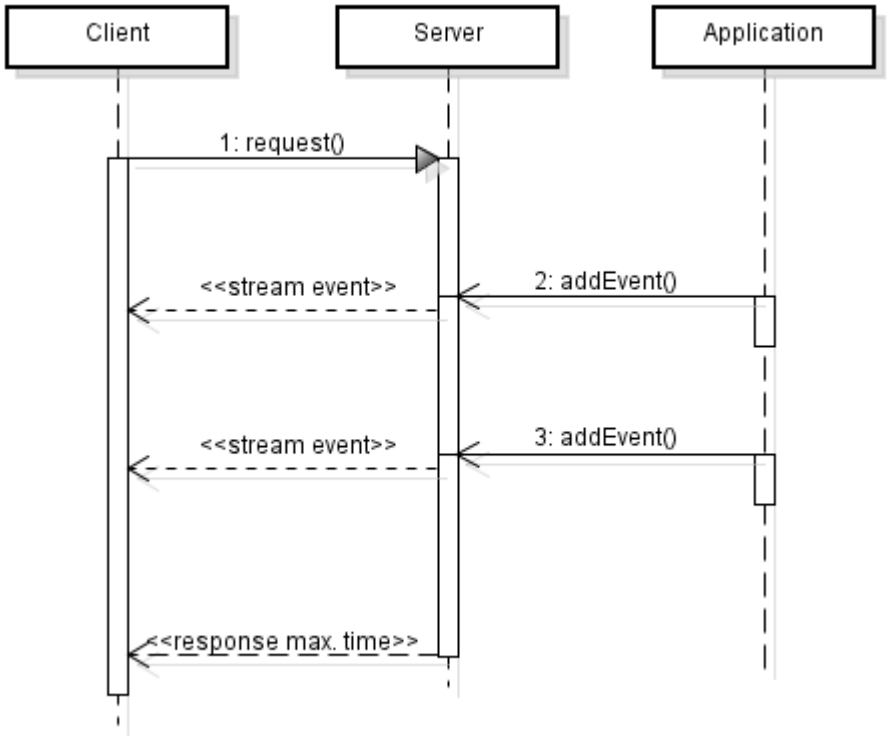
The following sequence diagram demonstrates the workflow of Long-Polling.



9.2.3 Server-push strategy – Streaming

Streaming is a server-push strategy which works without a lot of requests / connection cycles and therefore it is useful when a lot of events per client are expected (for example more than one event per second per client). One request is sent to the server and when an event occurs, it is sent back without sending the full response. The GWTEventService implementation allows to set a max. waiting time to restart the connection when no event has occurred within the max. waiting time. That can help to avoid any timeout recognitions, but can also be deactivated by setting the max. waiting time to zero when it isn't desired.

The disadvantage of streaming is, that the client/browser needs a client-/browser-specific implementation (some people would call it hack) to let the connection open and to handle the incoming events. GWTEventService uses the hidden iframe technique like many other comet frameworks, because that is a solution which can be defined quite universal and should work with all common browsers. However, some browsers may display a loading indicator because the browser recognizes that the loading is still in progress (which is true). That will be improved with future versions of GWTEventService by replacing the implementations with browser-specific solutions one after another.



To activate streaming in GWTEventService (notice long-polling is defined as default) you should set the two connection strategy properties which are described in chapter 2.2.

10 Configuration loading

That chapter describes the possibilities to load configurations for GWTEventService.

10.1 Default configuration loading sequence

There is a default sequence to load the GWTEventService configuration. At first, the properties file “eventservice.properties” will be searched. When that properties file isn't found at the classpath, the configuration parameters of the web-descriptor are taken. That means an eventservice.properties can be put at the classpath to overwrite the configuration parameters of the web-descriptor. When no configuration parameters / servlet-parameters are defined in the web-descriptor, the default configuration is loaded (min. time zero seconds, max. time 20 seconds, timeout time 90 seconds). That sequence can be manipulated

and extended with custom configuration loaders and configuration levels. See chapter 10.2 (Extending the configuration loading).

The default sequence to load the GWTEventService configuration is demonstrated below.

Position	ConfigurationLoader	ConfigLevel	Default search location
1	PropertyConfigurationLoader	DEFAULT (5000)	/classes/eventservice.properties
2	WebDescriptorConfigurationLoader	DEFAULT (5000)	/WEB-INF/web.xml
3	DefaultConfigurationLoader	HIGHEST (10000)	-

10.2 Extending the configuration loading

It is possible to manipulate or extend the configuration loading. For example to load the configuration from a remote database or an external XML file. That can be done by implementing a *ConfigurationLoader*. *ConfigurationLoader* is a simple interface which provides a load method and a method to check if the configuration is available. That custom loader can be embedded in the loading process with *EventServiceConfigurationFactory* which provides methods to add, remove and replace *ConfigurationLoader* (*de.novanic.eventservice.config.loader.ConfigurationLoader*) registrations. *ConfigurationLoader* instances must be registered to a *ConfigLevel* to define the order. There are five predefined levels (LOWEST, LOW, DEFAULT, HIGH and HIGHEST). As seen in the table of chapter 5.1 (Default configuration loading sequence), there are two configuration loaders at DEFAULT level and the *DefaultConfigurationLoader* is registered to the HIGHEST level. An ConfigurationLoader which is registered at a higher level (after *DefaultConfigurationLoader*) it hasn't an effect, because the *DefaultConfigurationLoader* is always available. When a custom *ConfigurationLoader* should for example be checked before searching for a properties file, it is recommend to register it to the LOW or LOWEST level or to a finer defined level which can be created with *ConfigLevelFactory*.

11 Build with Maven

The global Maven project configuration can be found in the root directory of GWTEventService and can be executed by Maven when you would like to build GWTEventService manually. The build can be started for example with “mvn clean install” which is one of the mostly used commands to build a Maven project. It builds the whole project sources with documentation, demo applications, release-notes, RoadMap, etc., runs the tests and creates the core- and developer-release archives. One of the advantages of Maven is that the dependent libraries are gathered automatically by Maven, so no library directory has to be setup manually. The most IDEs can handle the Maven projects. That means the project can be integrated within a few steps.

As mentioned above, there are two various archives released with GWTEventService. The core release only contains the absolutely necessary data to get an application using GWTEventService to run. Here you can see the differences between the core- and the developer-release in detail.

	core release	developer release
jars	yes	yes
licence	yes	yes
about	yes	yes
system requirements	yes	yes
release-notes	yes	yes
default configuration	yes	yes
sources	no	yes
javadoc	no	yes
documentation	no	yes
demo application	no	yes
RoadMap	no	yes

12 Deployment

The deployment of a GWT application which uses GWTEventService is very easy when your application is already prepared for deploying. That short article describes the deployment with Apache Tomcat (http://tomcat.apache.org/), but is also comparable to other web servers.

The JAR files of GWTEventService must simply be put into the “lib” directory of your WAR archive (or exploded directory) and a servlet mapping must be added to your web module deployment descriptor (web.xml). The following two figures show the structure of a WAR file containing the GWTEventService library and a web.xml with the important part of the appropriate servlet mapping.

WAR structure

- WAR
  - <your module path> (generated sources)
  - META-INF
  - WEB-INF
    - classes (compiled application, shouldn't contain any files of GWTEventService)
    - lib (libraries which are used by the application)
      - gwt-servlet.jar
      - **eventservice.jar**
      - **eventservice-rpc.jar**
      - **gwteventservice.jar**
    - web.xml

web.xml

```
<web-app>

  ...

  <servlet>
    <servlet-name>EventService</servlet-name>
    <servlet-class>
      de.novanic.eventservice.service.EventServiceImpl
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>EventService</servlet-name>
    <url-pattern>/<your-GWT-module-path>/gwteventservice</url-pattern>
  </servlet-mapping>

  ...

</web-app>
```

12.1 Deployment of the demo application

When one of the demo applications is built via Maven (for example via “mvn clean install”), a WAR file will be generated to the target directory. The created WAR file can be deployed normally to your web server. After

deploying the WAR file, you should reach the application with the following URL. The “democonversation” part of the URL (name of the WAR file) may not be necessary/allowed, depending on the configuration of your web server and the web server port could vary.

URL

http://localhost:8080/democonversation/DemoConversationAppUI.html

protocol	server name	:	port	/	deployment name / WAR file	/	GWT module start page
http://	localhost	:	8080	/	democonversation	/	DemoConversationAppUI.html

13 RoadMap

GWTEventService 1.0 – Initial main feature release (**released since 2008-12-31**)

GWTEventService 1.1

- Timeout listening (new UnlistenEventListener)
- EventFilter enhancements
- Optional configuration with the web descriptor (Web Container Config XML)
- Revised configuration loading
- Lock free programming
- A lot of refactorings

GWTEventService 1.2

- Preparations for new connection strategies (and support for streaming)
- Possibility to add events directly from the client side
- Simplified listener-event mapping
- Support for annotations
- Automatic reconnect on connection loss
- Maven integration

GWTEventService 1.3

- Domain registry to improve the handling of domains
- Support for subdomains

GWTEventService 2.x

- Rule engine for event bundle definitions
- New puffer engine for waiting connections / clients
- Observable interface with reference implementations for Collections and Maps
- Extended internal command handling
- Historization of events

GWTEventService 3.x

- GWTEventService-Console (for monitoring and configuration)

The road map has the state of 2010-02-28. To get the current road map, see
http://code.google.com/p/gwteventservice/wiki/StartPage

14 Conclusion / Feedback

We hope GWTEventService will fit all your needs to replace polling with a generic solution and supports you to clean-up the architecture of client-server communication with GWT. We are looking forward to get some feedback, suggestions or questions about the manual or the GWTEventService framework. You can contact us via the GWTEventService forum or user group or via mail: info@strawbill.com

15 FAQ

Q:	*Events* or internal data of *Events* are not correctly transferred
A:	Consider that *Events* must be serializable and require some special rules. For example the class does require a no-argument constructor and variables declared as final are not transferred. For more information see: http://code.google.com/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideSerializableTypes
Q:	Event listening doesn't work correctly with more than one web browser instance on the same client
A:	That is a general problem of web browsers / web applications, because the browser uses the same session for all web browser instances. To get it work you can run two or more different web browser applications (for example the GWT Hosted Mode Browser and your standard web browser or two other different web browsers when you have installed more than one). GWTEventService does also support multiple/shared sessions to avoid that problem. See chapter 2.2 for more information and how to activate the multiple/shared session support.
Q:	The build script of GWTEventService doesn't find GWT classes.
A:	The environment variable “GWT_HOME” must be set correctly. See the chapter “Build with Apache Ant” in the developer guide of GWTEventService for more information.

16 Appendix

16.1 Code examples

Here are some code examples to demonstrate the implementation of events and listeners. *UserJoinEvent* and *UserLeaveEvent* are subclasses of *GameUserEvent*. Chapter 14.2 is referencing to a complete demo application using GWTEventService.

GameUserListener

```
package de.novanic.gwteventservice.manual.example;

import de.novanic.eventservice.client.event.listener.RemoteEventListener;
import de.novanic.gwteventservice.manual.example.event.UserJoinEvent;
import de.novanic.gwteventservice.manual.example.event.UserLeaveEvent;

public interface GameUserListener extends RemoteEventListener
{
          void onUserJoin(UserJoinEvent aJoinEvent);

          void onUserLeave(UserLeaveEvent aLeaveEvent);
}
```

GameUserListenerAdapter

```
package de.novanic.gwteventservice.manual.example;

import de.novanic.eventservice.client.event.Event;
import de.novanic.gwteventservice.manual.example.event.UserJoinEvent;
import de.novanic.gwteventservice.manual.example.event.UserLeaveEvent;

public abstract class GameUserListenerAdapter implements GameUserListener
{
          public void apply(Event anEvent) {
                    if(anEvent instanceof UserJoinEvent) {
                              onUserJoin((UserJoinEvent)anEvent);
                    } else if(anEvent instanceof UserLeaveEvent) {
                              onUserLeave((UserLeaveEvent)anEvent);
                    }
          }

          public void onUserJoin(UserJoinEvent aUserJoinEvent) {}

          public void onUserLeave(UserLeaveEvent aUserLeaveEvent) {}
}
```

GameUserEvent

```
package de.novanic.gwteventservice.manual.example.event;

import de.novanic.eventservice.client.event.Event;
import de.novanic.gwteventservice.manual.example.Game;
import de.novanic.gwteventservice.manual.example.User;

public class GameUserEvent implements Event
{
          private Game myGame;
          private User myUser;

          public GameUserEvent() {}

          public GameUserEvent(Game aGame, User aUser) {
                    setGame(aGame);
                    setUser(aUser);
          }

          public Game getGame() {
                    return myGame;
          }

          public void setGame(Game aGame) {
                    myGame = aGame;
          }

          public User getUser() {
                    return myUser;
          }

          public void setUser(User aUser) {
                    myUser = aUser;
          }
}
```

Example class of usage

```
package de.novanic.gwteventservice.manual.example;

import de.novanic.eventservice.client.event.RemoteEventService;
import de.novanic.eventservice.client.event.RemoteEventServiceFactory;
import de.novanic.eventservice.client.event.domain.Domain;
import de.novanic.eventservice.client.event.domain.DomainFactory;
import de.novanic.gwteventservice.manual.example.event.UserJoinEvent;
import de.novanic.gwteventservice.manual.example.event.UserLeaveEvent;

public class UsageClass
{
  private static final Domain GAME_DOMAIN = DomainFactory.getDomain("gamedom");

  public void init() {
    //do something

    RemoteEventServiceFactory theEventServiceFactory =
                              RemoteEventServiceFactory.getInstance();
    RemoteEventService theEventService =
                              theEventServiceFactory.getRemoteEventService();

    theEventService.addListener(GAME_DOMAIN, new GameUserListenerAdapter() {
      public void onUserJoin(UserJoinEvent userJoinEvent) {
        //do something with the new user
      }

      public void onUserLeave(UserLeaveEvent userLeaveEvent) {
        //do something with the leaved user
      }
    });

    //do something
  }
}
```

16.2 Demo application

There is a very simple demo application called “HelloGWTEventService” for GWTEventService available which shows the general usage of GWTEventService. The server side sends every five seconds an event and the client side displays the occurred events in a list.

There is also a more complex demo application called “DemoConversationApp” available. It shows a small conversation application with a channel system to demonstrate the “*EventFilter*” functionality. There is only one domain for the conversation context and an *EventFilter* to divide the conversation into different channels. The *EventFilter* is processed on the server side and checks the channels to decide if an event is important for the particular user. If not, the connection is still active, otherwise the events are returned to the client/user. Global events like “*NewChannelEvent*” or “*CloseChannelEvent*” are processed by the *EventFilter* without a check for a channel.

The most important functionality is contained in the class “*ConversationControl*” (*de.novanic.gwteventservice.demo.conversationapp.client.conversation.control.ConversationControl*) and should be a good entry point to study the usage in practice.

Project-SVN-URL: http://gwteventservice.googlecode.com/svn/trunk/

Demo-SVN-URL: http://gwteventservice.googlecode.com/svn/trunk/demo/