Echo

Echo
November 2010

Backplane

Abstract

Backplane is a framework to facilitate interaction between multiple independent client- and server-side parties in the context of a browser session.

Table of Contents

1. Introduction

In essence Backplane is a message distribution system where messages are delivered reliably, in order and in nearly real-time fashion. The framework may be used in different scenarios which build on top of the transport-level semantics described in this document.

1.1 Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Overview

A Backplane setting consists of the following components:

- A Backplane server
- A Client-side Backplane library (a Javascript library executed by an end user's browser)
- Backplane-enabled widgets
- Backplane clients

A Backplane server is an independent orchestrator of the message interchange between the parties and may serve multiple independent *buses*.

2.1 Buses

A bus is essentially a namespace to allow multiple customers to be served off a single server. A bus is referenced by a short identifier (e.g. "customer.com", "organization.org"). Bus identifiers are known to all relevant parties beforehand.

It is assumed that a relationship of trust exists between all clients granted permission to post messages to a specific bus.

2.2 Channels

Any message entering the Backplane is routed to a *channel* and gets delivered to all consumers of that channel.

Channels are:

- allocated within buses
- referenced by unique (typically unguessable) identifiers (URLs)
- non-persistent and expire after being inactive (no messages posted) for specific amount of time (e.g. 30 minutes)

Here is an example of a Backplane channel identifier:

```
http://backplane.customer.com/v1/bus/customer.com/
channel/8ec92f459fa70b0da1a40e8fe70a0bc8
```

3. Backplane Messages

A Backplane message is a JSON object with the following predefined fields:

source	the source of the message, MUST be a URL ("http://client.com")
type	message type, string
payload	arbitrary object carrying data specific to the particular message type

A Backplane message MAY be wrapped in a Magic Envelope [draft-panzer-magicsig-00]. Consumers MUST be able to process (unwrap) such messages (note that unwrapping a magic envelope is not the same thing as signature verification).

4. Backplane Server

4.1 Channels

Channels are created implicitly once a Backplane message get posted to a channel that doesn't yet exist. It is up to the posting party to come up with the *channel name* given the security constraints of the setting (e.g. in a simple setups one can get away with predefined channel names; in another scenario channel names will have to be unguessable, etc).

Channel name MUST only contain characters from the base64url [RFC4648.section5] character set.

4.2 Message Retrieval Conventions

The server maintains a "window" of most recent messages for each of the existing buses and channels. The exact amount of messages that fit into the window depends on the server implementation or policy.

All API methods dealing with message retrieval observe the principles outlined below. A retrieval request may contain an optional 'since' parameter, referring to a message previously seen by the caller.

If the 'since' parameter is...

...omitted, the whole window worth of messages is returned.

...present and refers to a message...

...currently residing in the window, the server returns all messages that are more recent than the referenced one.

...which is no longer in the window, the whole window worth of messages is returned.

In order to avoid omissions, a client needs to invoke the method often enough (specific recommendations will be based on the server implementation or policy).

4.3 Message Frames

Whenever a Backplane server needs to communicate metainformation along with a message, the Backplane message is wrapped into a "frame". The Backplane frame is a JSON object with the following fields:

message	the Backplane message
channel_name	Name of the channel in a bus the message was posted to (can be used to construct Channel ID of the message)
id (string)	an arbitrary identifier assigned by the server to this message for the purpose of later referencing

4.4 API

A Backplane server provides the following API.

4.4.1 Get All Messages In A Bus

Endpoint	`/v1/bus/<BUS_NAME>`
HTTP Method	GET
Security	HTTPS with basic HTTP authentication
Parameters	since (optional): message identifier (see Message Retrieval Conventions (Section 4.2) above)
Returns	a list of Backplane frames

This method is used by Backplane clients to receive all messages distributed on the specific bus <BUS_NAME>.

4.4.2 Get Channel Messages

Endpoint	`/v1/bus/<BUS_NAME>/channel/<CHANNEL_NAME>`
HTTP Method	GET
Security	none
Parameters	since (optional): message identifier (see Message Retrieval Conventions (Section 4.2) above)
Returns	a list of Backplane frames

This method is used by Backplane-enabled widgets to receive messages broadcast to a specific channel.

4.4.3 Post Messages To A Channel

Endpoint	`/v1/bus/<BUS_NAME>/channel/<CHANNEL_NAME>`
HTTP Method	POST
HTTP Request Body	a list (JSON array) of Backplane messages to post to the channel
Security	HTTPS with basic HTTP authentication

This method injects a message (or multiple messages) to the channel. If channel <CHANNEL_NAME> doesn't exist, it is created.

5. Client-side Backplane library API

A Backplane Javascript library runs in an end user's browser and mediates communication between Backplane-enabled widgets on the page and the Backplane server.

Only one instance of the Backplane library on a given page is possible. The library has to be the first to load on the page to make it possible for other scripts to use its subscription functionality.

The library provides the following API (all methods are static):

```
/**
 * Initializes the backplane library
 *
 * @param {Object} Params - hash with configuration parameters.
 *   Possible hash keys:
 *     serverBaseURL (required) - Base URL of Backplane Server
 *     busName (required) - Customer's backplane bus name
 *     channelName (optional) - custom specified channel name
 */
Backplane.init(Params);

/**
 * Subscribes to messages from Backplane server
 *
 * @param {Function} Callback - Callback function which accepts backplane
 messages.
 * @returns Subscription ID which can be used later for unsubscribing.
 */
Backplane.subscribe(Callback);

/**
 * Removes specified subscription
 *
 * @param {Integer} Subscription ID
 */
Backplane.unsubscribe(SubscriptionID);

/**
 * Returns channel ID (like http://backplane.customer.com/v1/bus/
customer.com/channel/8ec92f459fa70b0da1a40e8fe70a0bc8)
 *
 * @returns Backplane channel ID
 */
Backplane.getChannelID();

/**
 * Notifies backplane library about the fact that subscribers are going
 * to receive backplane messages within specified time interval.
 *
 * @param {Integer} TimeInterval - Time interval in seconds
 * @param {Array} MessageTypes (optional) - a list of expected message
 types
 */
Backplane.expectMessagesWithin(TimeInterval, MessageTypes);
```

5.1 Initialization

Backplane is initialized using the `Backplane.init` method.

During initialization the library generates a random *channel name* unless information about one for the specified *bus name* already exists in the `backplane-channel` cookie. If client-side generation of the

channel name is considered non-secure in a particular setting, the implementation may perform a request to obtain a channel name from the server side and pass it as optional `channelName` parameter. Channel name stored in the `backplane-channel` cookie takes precedence over the one passed via the `channelName` parameter.

After the initialization the library stores the current channel name in the `backplane-channel` cookie set against complete domain name of currently opened page. The cookie is set for 5 years in advance and keeps information about association of bus names to channel names (to support possibility to use the library with several different bus names on the same domain). The information about the association is stored in a serialized form.

Here is an example of cookie that stores association of bus names `example.com` and `example.org` to the corresponding channel names `123` and `456`:

```
backplane-channel=example.com:123|example.org:456
```

After the channel ID has been determined, the library performs first reading of messages from a channel, discards all of them (remembering only the identifier of the very last one) and starts polling the server for new messages since the latest message. This way the library is guaranteed to push to subscribers only those messages which arrived after the library had been fully initialized.

5.2 Subscription Management

The library provides a method for widgets to set up notification callbacks: `Backplane.subscribe`. The method returns subscription id which can be later used for unsubscribing using the `Backplane.unsubscribe` method.

After the initialization the library starts polling the server for new events. All incoming events are delivered to the widgets that have registered callbacks with the library.

5.3 Hints

For performance reasons the Backplane library polls the server quite infrequently (once a minute or so). Since Backplane events usually take their origin on the client side (e.g. the user clicking a button), widgets on the page are in a position to hint the library that a Backplane message may be soon delivered. Upon the receipt of such hint (via `expectMessagesWithin` method), the library temporarily changes the polling frequency (to once a second or so) and gradually increases it back to the default polling interval.

The `expectMessagesWithin` method can accept an optional list of expected message types.

If the library accepts *any* message from the passed message types list, it gradually returns to the slower polling frequency mode;

If messages with only one type are expected, the second argument may be specified as a string;

Each call of the method adds passed message types to the list of expected message types. In other words, if a user calls the method with a message type "type1" and then performs one more call with a message type "type2", the library will run in the fast polling mode until it received messages of *all* the types or until the libray reached the maximum allotted waiting time interval.

5.4 Usage Example

```
Backplane.init({
    serverBaseURL: "http://backplane.customer.com/v1",
    busName: "customer.com"
});

var escSubscription = Backplane.subscribe(function(backplaneMessage) {
    alert(backplaneMessage.payload);
});

// We can ask the library to perform more frequent polling
// if a widget, for example, expects a message from Backplane pretty soon
// using the expectMessagesWithin method which accepts
// time interval of possible message arrival in seconds
Backplane.expectMessagesWithin(10);

// The method can accept an option list with expected message types.
// The library stops fast polling when it receives a message of
// either type.
Backplane.expectMessagesWithin(10, ["type1", "type2"]);

// Subsequent calls extend the list of expected message types.
// The library stops fast polling only after it has received
// a message of type1 or type2 AND a message of type3 or type4.
Backplane.expectMessagesWithin(10, ["type3", "type4"]);

Backplane.unsubscribe(escSubscription);

// If a widget needs Backplane channel ID it can get it using the
// getChannelID method
Backplane.getChannelID();
```

6. References

6.1 Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997, <http://tools.ietf.org/html/rfc2119>.

6.2 Informative References

[RFC4648.section5] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", October 2006, <http://tools.ietf.org/html/rfc4648#section-5>.

[draft-panzer-magicsig-00] Panzer, J. and B. Laurie, "Magic Signatures", February 2010, <http://salmon-protocol.googlecode.com/svn/trunk/draft-panzer-magicsig-00.html#anchor3>.

Author's Address

Echo