

## Widget API

### Motivation

The Widget API was designed with the following goals in mind: To hide the complexity of the Agent Application from the widgets; to present small and consistent interface to the widgets; and to maintain a stable interface for the widgets.

### Overview

The Widget API is split in two major subsystems:

- An **event system** used to broadcast various messages from the services to the widgets and for widget-to-widget communication
- The **interfaces**, providing specific points of access to the services and the core functionality

### Widget API

External widgets are generally hosted on a different domain and cannot access the Widget API directly. To bypass the domain barrier the Agent application's core will open a messaging channel with the browser's Messaging API.

The code below demonstrates one way to subscribe to and handle messages from the API:

```
const origin = 'the origin of the agent application';

let port;

window.addEventListener('message', message => {
  // Make sure that the channel comes from the correct source:

  if (message.origin !== origin) return;

  // Setup the communication channel:

  if (!port) {
    port = message.ports[0];
    port.onmessage = receiver;
  }
});
function receiver(message) {
  ...
}
```

The payload of the messages is contained in `message.data`. The core will add a `message.data.type` property to all messages too.

Requests to the Widget API can only be send trough the provided port:

```
port.postMessage(message);
```

### Interfaces

To get a property or to call a method of the Widget API the `{call, args}` message format should be used, where `call` is the path to the method (or property) in the API.

In case of a method call, `args` is an array of all required arguments for the method call.

Example:

```
port.postMessage({
  call: 'tab.setTitle',
```

```
args: ['new title']
```

```
});
```

If the method returns a result, it will be sent to the external widget by the **report.onmessage** handler in the format {name, value, type}, where name is the name of the requested property or method, value is the value of the property or the result of the call, and type will be the string 'result'.

Example response to a `getOption` call:

```
{
  name: 'widget.getOption',
  value: 'https://demo.puzzel.com/dev/widgets/external/demo/',
  type: 'result'
}
```

If the called method doesn't return a result, no message will be sent by the core.

If the called method returns a promise the message will be sent when the promise is resolved or rejected. In case the promise is resolved a standard result message will be sent by the core, where value will contain the value of the promise. In case the promise is rejected an 'error' message will be sent:

```
{
  name: 'widget.setOption',
  value: 'Unexpected end of JSON input',
  type: 'error'
}
```

If matching a call to a result is required, the optional id could be added to the request. It will be returned back:

```
{
  call: 'tab.getOption',
  args: ['option name'],
  id: '0123456789'
}
```

Result:

```
{
  name: 'tab.getOption',
  value: 'option value',
  id: '0123456789'
}
```

The widget can also observe a property for changes by sending a {watch} message. The watch field should hold the path to the property in the Widget API.

If the value of that property changes, the core will send a {name, old, new, type} message, where name will be the same property path, old will be the value of that property before the change, new after the change, and type that will be the string 'changed'.

## Events

The external widgets can subscribe to events by sending a {subscribe, options: {once, address}} message to the core.

The subscribe field should contain the event's name. The whole options field is optional as are its properties:

the boolean once and the address string. The address has the same meaning as in

the `ExtendedEventAggregator`'s methods. The once set means that the `subscribeOnce` method will be used, i.e. the external widget will receive only a single event before the subscription terminates itself.

The events will be received with a {name, value, type} message, where name will be the name of the event, value is the payload, and type will be 'event'.

Example:

```
{
  name: 'userStatusChanged',
  value: 'System',
  type: 'event'
}
```

The complete API reference will be made available on [help.puzzel.com](https://help.puzzel.com), in the Developer section of the Knowledgebase.