

“WebPontis” (Latin-inspired): WebBridge — a link between Unity and the Web.

Version: 1.0.0

Document version: 1.0

IEVO

OVERVIEW	5
Why not SendMessage?.....	6
On the Unity side:.....	6
On the JavaScript side:.....	6
WebPontis benefits:.....	7
SPECIFICATIONS	8
QUICK START	9
Component-based (No Code) Approach.....	9
Event sending.....	9
Event receiving.....	12
Code-based Approach.....	15
Event sending.....	15
Event receiving.....	16
DISPATCHING EVENTS	17
Component-based Approach.....	17
Supported Payload Types.....	17
No payload.....	17
Basic types.....	18
Structs.....	18
Adding the Component.....	18
Test from the Inspector (Editor Only).....	21
Code-based Approach.....	22
Supported Payload Types.....	22
No payload.....	22
Primitives.....	22
Arrays.....	22
Structs.....	22
Custom objects.....	22
Event Dispatching.....	23
No Payload.....	23
Primitive Payload.....	23
Custom Payload.....	24
Custom Payload Requirements.....	25
RECEIVING EVENTS	26
Component-based Approach.....	26
Life Cycle Mode.....	27
Modes.....	27
Supported Payload Types.....	27
Adding the Component.....	28
Code-based Approach.....	32
Supported Payload Types.....	32
No payload.....	32

Primitives.....	32
Arrays.....	32
Structs.....	32
Custom objects.....	32
Receiving Events.....	33
No Payload.....	33
Primitive Payload.....	33
Custom Payload.....	34
Custom Payload Requirements.....	34
Removing Listeners.....	36
How to remove listeners.....	36
Remove all listeners (not recommended).....	37
EVENT NAMES RECOMMENDATIONS.....	38
Common Naming Styles.....	38
1. kebab-case (default).....	38
2. Namespace-style with colons :.....	38
What JS frameworks use.....	39
Recommendations.....	39
Avoid!.....	39
USING TYPED EVENT NAMES.....	40
IOutboundEvent.....	40
IInboundEvent.....	41
Benefits of Typed Events.....	41
Why Use struct Instead of class?.....	42
CONFIGURATION.....	43
Basic options.....	43
Log Level.....	43
Channels.....	44
Target Origin (for outgoing events).....	47
Allowed Origins (for incoming events).....	47
No-Code Approach.....	48
Options which can be configured.....	48
Using the Prefab.....	49
Optional: Create or Edit Settings Asset.....	49
Code-based Approach.....	50
Log Level.....	50
Channels.....	51
Target Origin (for outgoing events).....	51
Allowed Origins (for incoming events).....	51
Replace the JSON Serializer.....	52
Limitations of JsonUtility.....	52
When to use a custom serializer like Newtonsoft.Json.....	52

How to replace it.....	53
Switching back to Unity's default.....	53
CUSTOM LOGGER.....	54
When to use a custom logger.....	54
How to replace it.....	54
Reset to default.....	56
CUSTOM INTEROP SERVICE.....	57
How it works.....	57
Replacing the Interop Layer.....	57
Example Use Cases.....	58
WebSocket-based interop.....	58
Native plugin bridge.....	59
CUSTOM MESSAGE FORMAT.....	60
Default message format (postMessage).....	60
How to customize message structure.....	61
DEBUGGING TOOLS.....	63
EventListenersMonitor.....	63
DebugEventSender.....	64
DebugEventReceiver.....	65
WebPontis Forwarder.....	66
How to use it.....	67
Supported flows.....	67
Useful for:.....	67
Browser Integration.....	68
Plugin Location.....	68
Installation Instructions.....	68
For Chromium-based browsers (Chrome, Edge):.....	68
For Firefox:.....	68
Usage.....	70
Browser Plugin Settings.....	71
Port Settings.....	71
Message Formatters.....	73
Outbound Formatter.....	73
Inbound Formatter.....	73
Controls.....	74
CONTACTS.....	74

OVERVIEW

WebPontis is a Unity plugin that helps you send and receive events between your Unity app and JavaScript. It works in WebGL/WebGPU builds and inside the Unity Editor.

WebPontis removes the usual pain of integrating Unity and HTML/JavaScript — no more awkward **SendMessage** hacks or dealing with raw JSON manually. Instead, it provides a clean, type-safe, object-oriented event system that feels natural in both C# and JavaScript.

You can use **WebPontis** entirely without writing any code, just by using the built-in components. But if you prefer working in code, the API for sending and receiving events is very simple and mirrors the JavaScript side.

To receive events from JavaScript, use:

```
EventBridge.AddEventListener(...)
```

To send events to JavaScript, use:

```
EventBridge.DispatchEvent(...)
```

You can work with many data types, like **string**, **bool**, **int**, **float**, **arrays**, **Vector3**, or even **custom objects**. EventBridge handles all the serialization and communication for you.

To **make development and debugging easier**, the plugin includes several helpful tools:

- **WebPontisForwarder** – redirects events between the browser (requires installing the browser extension) and the Editor.
- **EventListenersMonitor** – shows which events are currently being listened to.
- **DebugEventSender** – allows you to manually send test events.
- **DebugEventReceiver** – logs and displays all received events in real time.

These tools are available in the Unity Editor under the **Tools > WebPontis** menu.

Why not SendMessage?

Unity provides a basic way to receive messages from JavaScript in WebGL builds using `SendMessage(...)`, but it comes with several limitations:

On the Unity side:

- Only works with methods on **MonoBehaviour** components.
- Requires hardcoded `GameObject` and method names.
- Only supports **string** parameters.
- No type safety — any typo or mismatch causes silent runtime errors.
- Doesn't work in the Unity Editor — testing becomes harder and slower.

On the JavaScript side:

- You have to wait for the Unity instance to load.
- You must get a reference to **unityInstance**, which can vary.
- Sending messages depends on global state and timing.
- Communication is fragile.
- Hard to scale.

As a result, communication looks like this:

```
// JavaScript
unityInstance.SendMessage("GameManager", "OnEvent", "payload");
```

```
// Unity (MonoBehaviour only)
public void OnEvent(string payload)
{
    Debug.Log("Received: " + payload);
}
```

With **WebPontis**, everything is simpler, safer, and editor-friendly:

```
// JavaScript
window.dispatchEvent(new CustomEvent("my-event", {
  detail: "payload"
}));
```

```
// Unity
EventBridge.AddEventListener<string>("my-event", data =>
{
  Debug.Log("Received: " + data);
});
```

WebPontis benefits:

- No MonoBehaviour required!
- No need to know GameObject names.
- Works in WebGL/WebGPU and the Editor.
- Type-safe.
- Fully event-driven.
- Integrates naturally with HTML and JavaScript event systems.

SPECIFICATIONS

- No external dependencies.
- Works in **Unity Editor** and **WebGL/WebGPU builds**.
- Compatible with **Unity 2019.4** and above.
- Supports all major event data types (primitive, arrays, JSON, Vector3).
- Editor tools included (no need to write debug UIs yourself).

QUICK START

Component-based (No Code) Approach

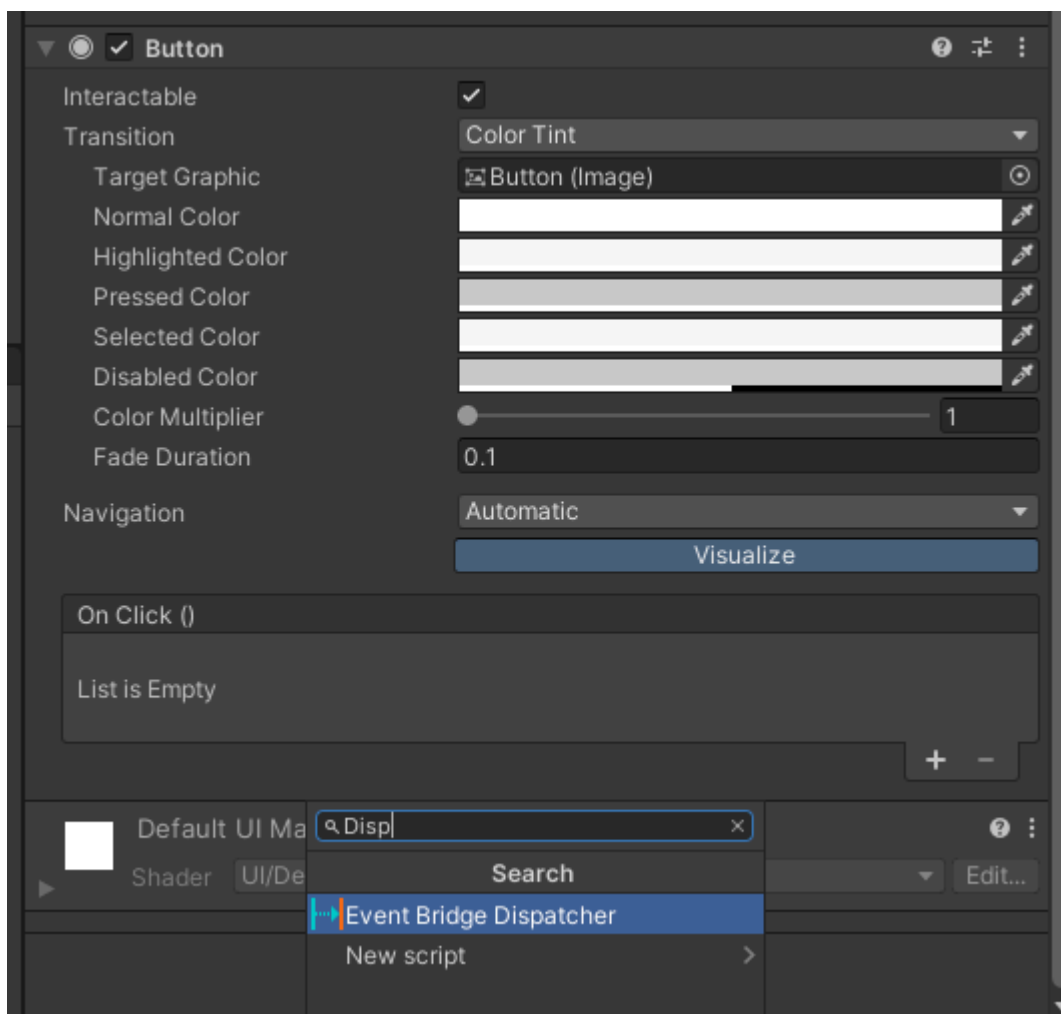
Example: 01_QuickStart_NoCode

You can use **EventBridge** without writing any code by using the built-in components.

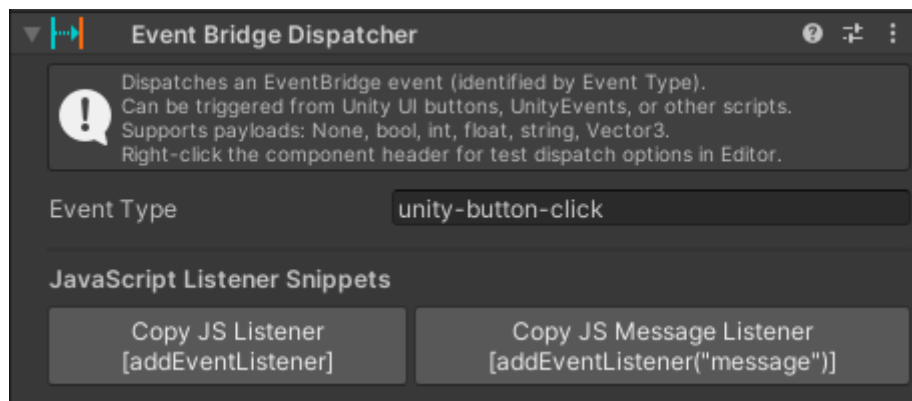
Event sending

Let's start by sending a message from Unity to JavaScript — triggered by a UI button click.

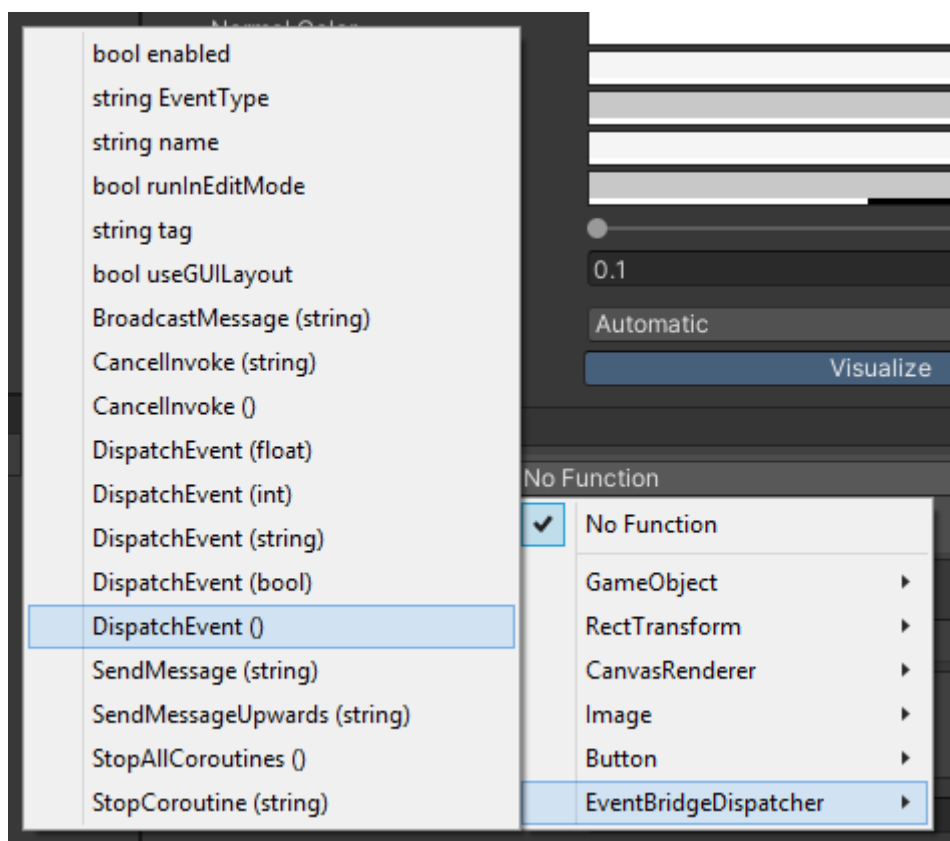
1. In the **Hierarchy**, select a UI **Button** (or create a new one).
2. Add the **EventBridgeDispatcher** component to the same GameObject.



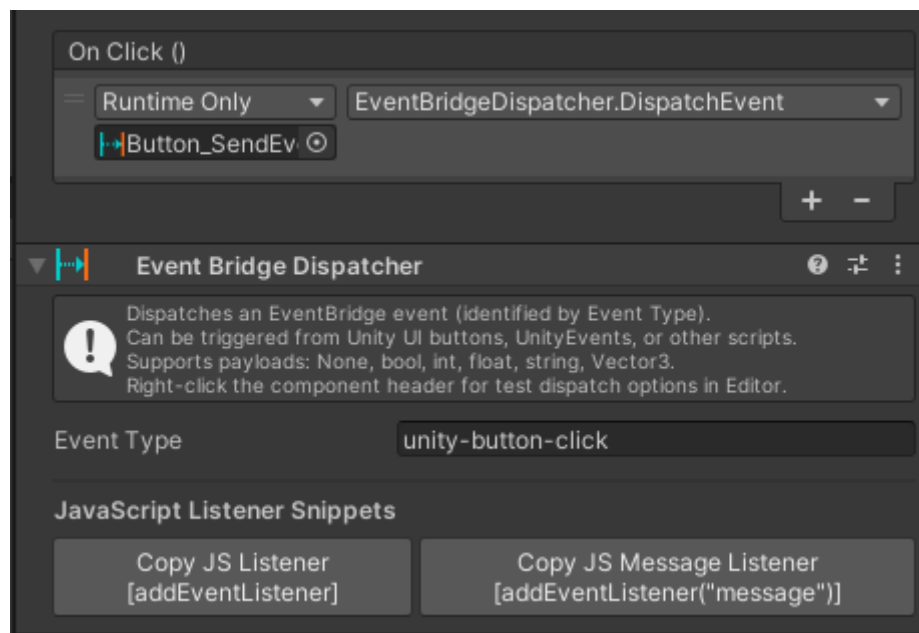
- In the **Event Type** field of the **EventBridgeDispatcher**, enter:
unity-button-click



- In the **Button** component, scroll to the **On Click ()** section.
- Click "+", drag the GameObject with the **EventBridgeDispatcher** into the object field.
- From the function dropdown, choose:
EventBridgeDispatcher → DispatchEvent()



Now when the button is clicked, an event called **unity-button-click** will be sent to JavaScript.



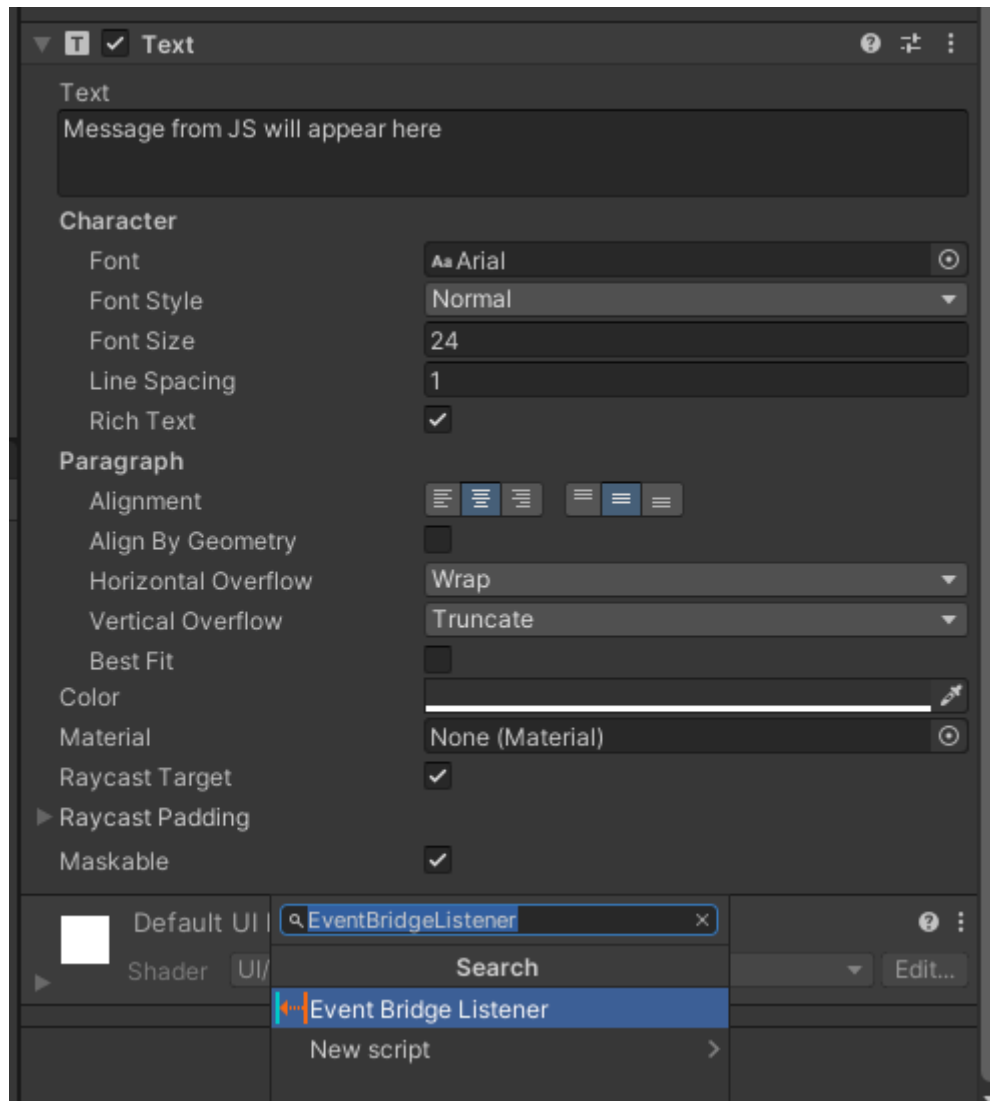
To receive the message in JavaScript, add the following code to your HTML or JavaScript:

```
// JavaScript
window.addEventListener("unity-button-click", () => {
    alert("Button click from Unity!");
});
```

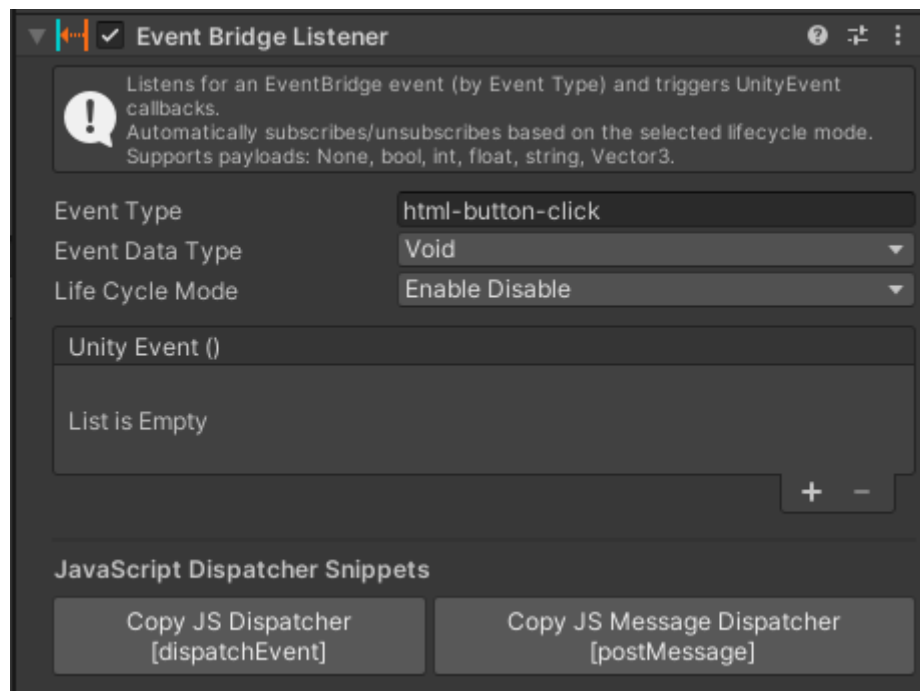
Event receiving

Now let's receive a message from HTML/JavaScript into Unity.

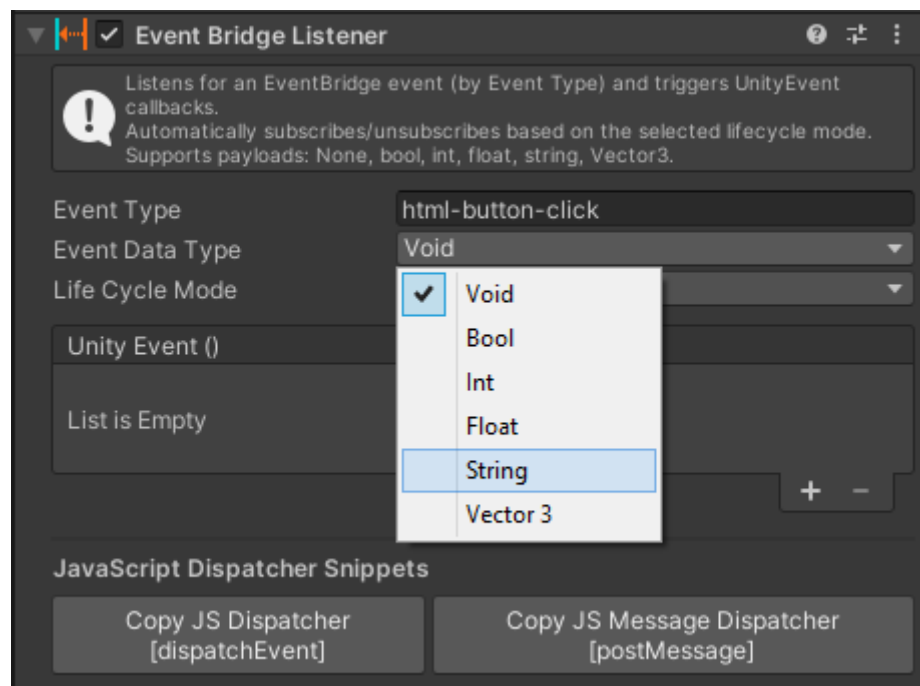
1. In the **Hierarchy**, select a **Text** component (e.g., **Text** from Unity UI) or create a new one.
2. Add the **EventBridgeListener** component to the same GameObject.



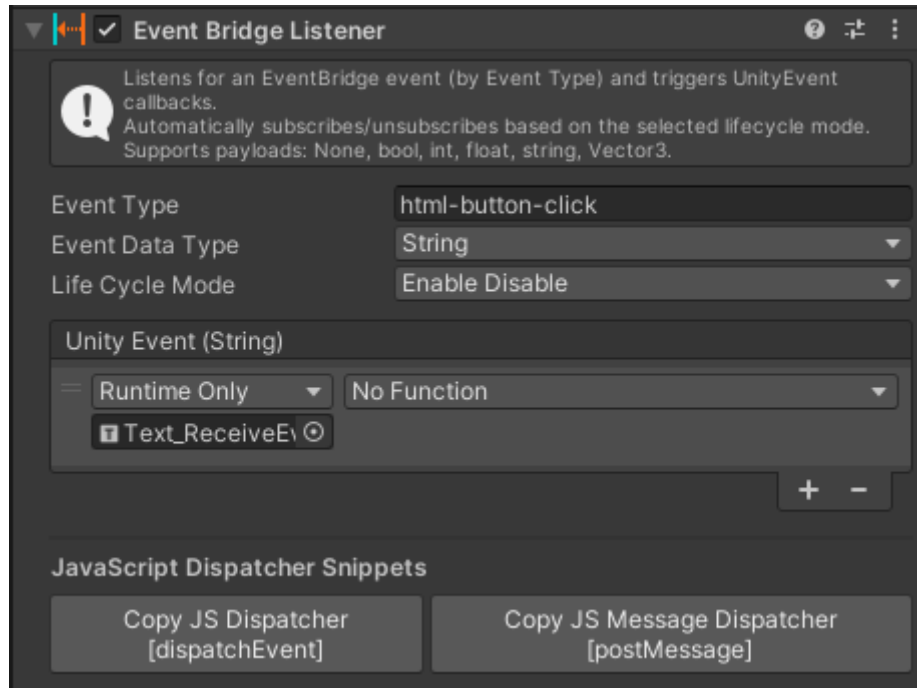
3. In the **Event Type** field, enter: **html-button-click**



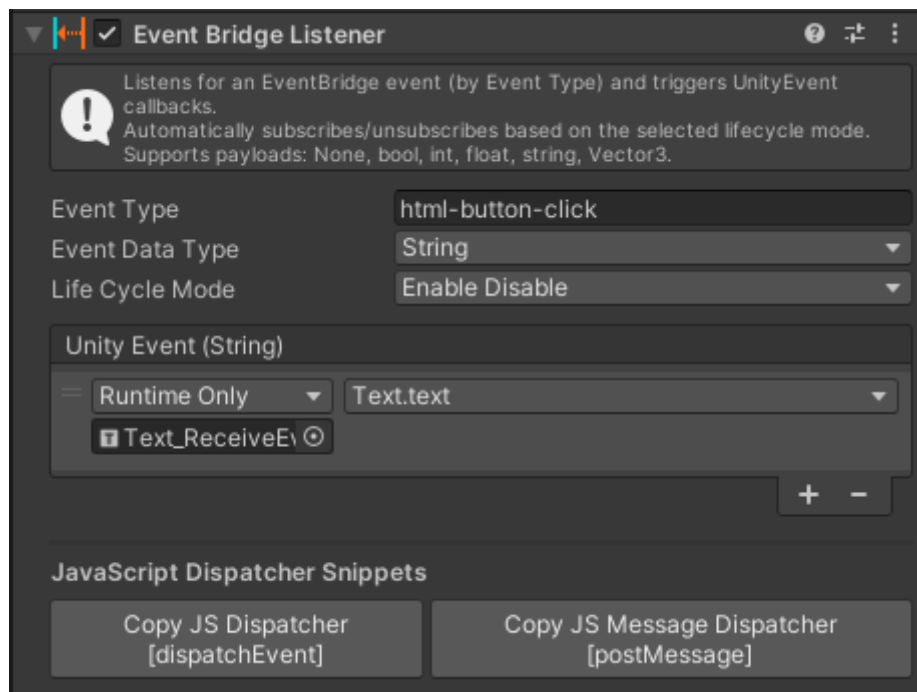
4. Set **Event Data Type** to **String**.



5. In the **Unity Event** section, click "+" to add a response.
6. Drag the **Text** component into the object field.



7. From the function dropdown, choose:
Text → **text (string)**



This will update the text when a message is received from JavaScript.

To send the event from JavaScript, use:

```
// JavaScript
window.dispatchEvent(new CustomEvent("html-button-click", {
  detail: "Hello from JavaScript!"
}));
```

That's it! Now we have simple two-way communication between the Unity application and JavaScript.

Code-based Approach

Example: [02_QuickStart_Code](#)

Namespace: IEVO.WebPontis.EventSystem

Two-way communication can be implemented in just a few lines of code.

Event sending

To send a text message from Unity to JavaScript, simply write:

```
// Unity
EventBridge.DispatchEvent("unity-string-message", "Hello from
Unity!");
```

On the JavaScript side, you can receive it like this:

```
// JavaScript
window.addEventListener("unity-string-message", function (e) {
  alert(e.detail);
});
```

Event receiving

To receive a message in Unity from HTML/JavaScript, you can write:

```
// Unity
EventBridge.AddEventListener<string>("html-string-message", data
=>
{
    Debug.Log(data);
});
```

Sending the message from HTML/JavaScript looks like this:

```
// JavaScript
window.dispatchEvent(new CustomEvent("html-string-message", {
    detail: "Hello from HTML!"
}));
```

With only two lines of code in Unity, you've set up a working two-way connection with JavaScript.

DISPATCHING EVENTS

WebPontis allows you to send events from Unity to JavaScript.

This can be done in two ways:

- by using the **EventBridgeDispatcher** component (no coding required),
- or by calling **EventBridge.DispatchEvent(...)** directly in code.

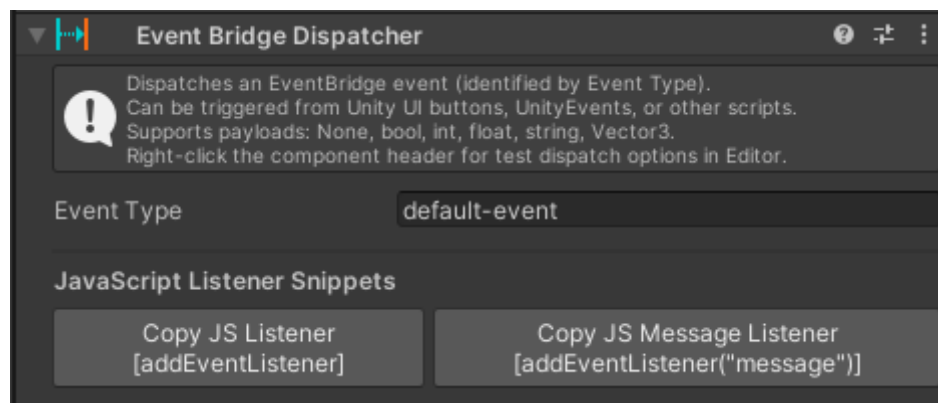
You can send events without data, or include payloads such as numbers, strings, arrays, Vector3, or custom objects. All messages are automatically serialized and delivered to the browser via CustomEvent.

Let's look at both approaches.

Component-based Approach

Example: [01_QuickStart_NoCode](#)

The **EventBridgeDispatcher** component lets you send an event from any GameObject — most commonly a UI button.



Supported Payload Types

The **EventBridgeDispatcher** component supports the following payload types:

No payload

- **None** (empty event)

Basic types

- **bool, int, float, string**

Structs

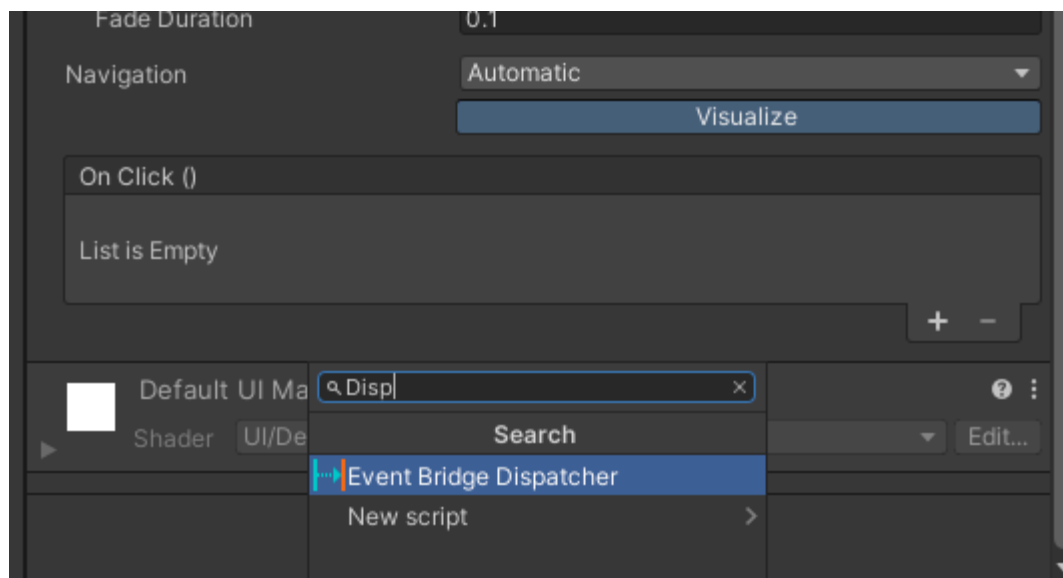
- **Vector3**

💡 **Note:** Internally, **float** values are converted to **double** to match the Web serialization format.


💡 **Note:** For arrays, custom objects, or other advanced data types, use the [Code-based Approach instead](#).

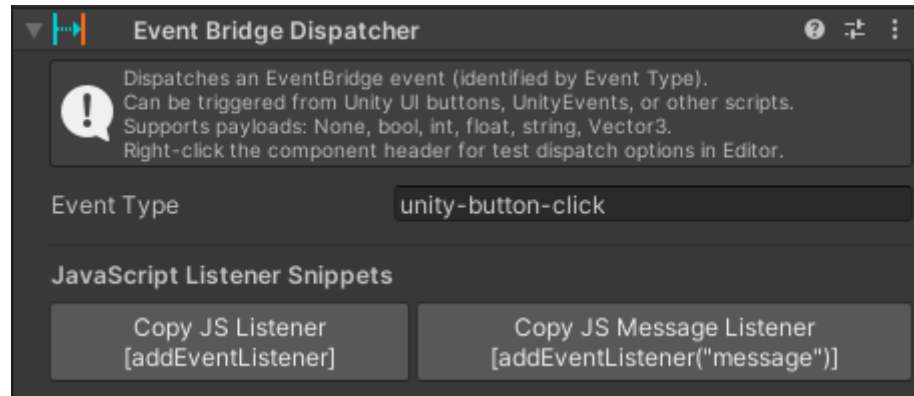
Adding the Component

1. In the **Hierarchy**, select a UI **Button** (or create a new one).
2. Add the **EventBridgeDispatcher** component to the same GameObject.

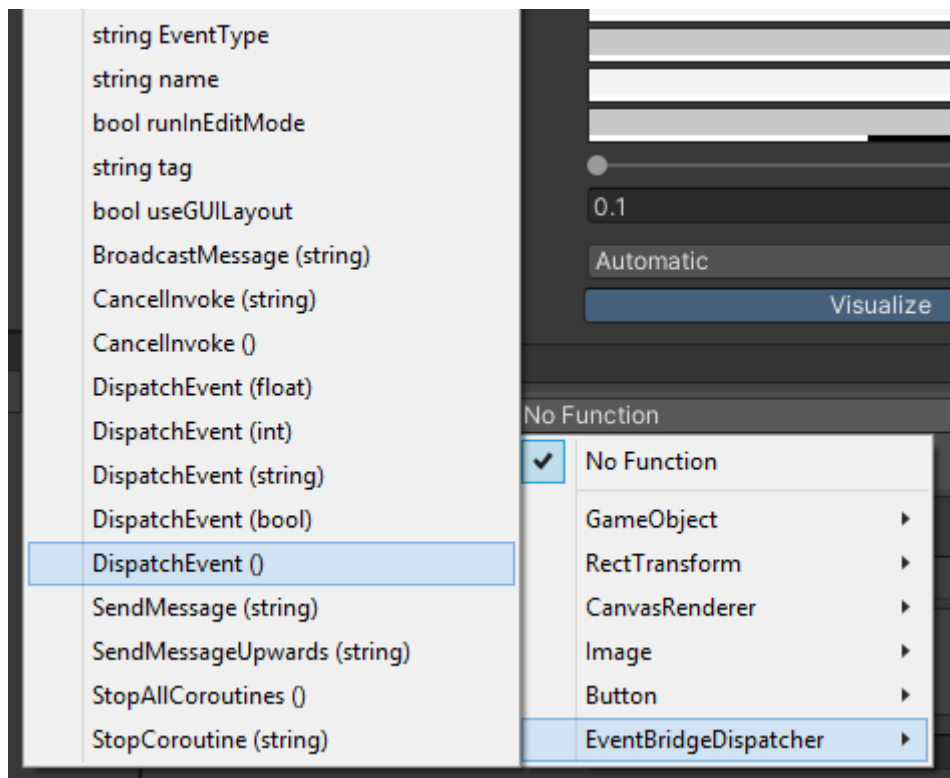


3. In the **Event Type** field of the **EventBridgeDispatcher**, enter:
unity-button-click

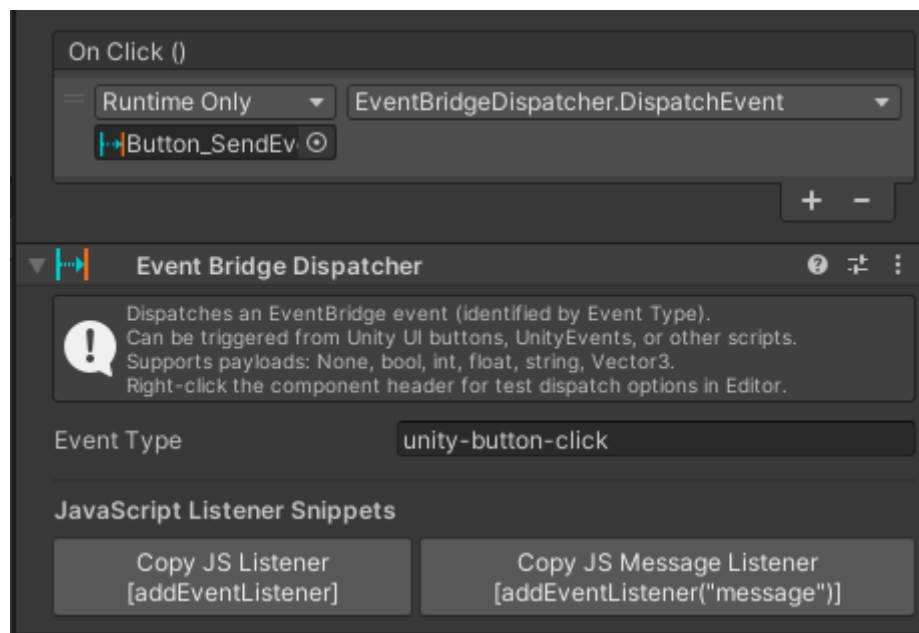
 **Note:** For event naming best practices, see [EVENT NAMES RECOMMENDATIONS](#)



4. In the **Button** component, scroll to the **On Click ()** section.
5. Click "+", drag the GameObject with the **EventBridgeDispatcher** into the object field.
6. From the function dropdown, choose:
EventBridgeDispatcher → DispatchEvent()



Now when the button is clicked, an event called **unity-button-click** will be sent to JavaScript.

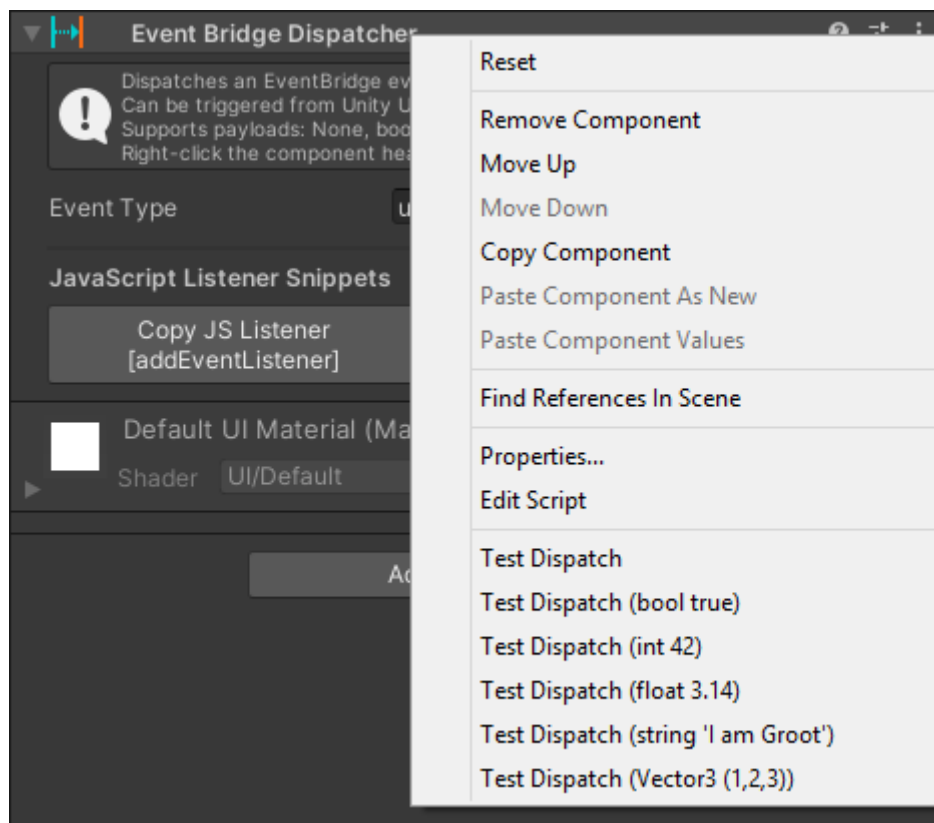


 **Note:** You can also call **dispatcher.DispatchEvent(...)** from your own scripts.

Test from the Inspector (Editor Only)

For quick testing in the Unity Editor, right-click the component header and choose a **Test Dispatch** option from the context menu.

This lets you simulate sending an event with predefined test values (e.g., **true**, **42**, **"I am Groot"**, **Vector3(1,2,3)**), without writing any code or linking to the UI.



Code-based Approach

Example: 02_QuickStart_Code; 03_TypedEventNames; 04_CustomPayload;
10_TextureToHtml;

Namespace: IEVO.WebPontis.EventSystem

You can also dispatch events directly from code using

EventBridge.DispatchEvent(...).

This approach gives you full flexibility and supports all available data types.

Supported Payload Types

The code-based approach supports the full set of payload types available in

EventBridge:

No payload

- **None** (empty event)

Primitives

- **bool, sbyte, byte, short, ushort, int, uint, long, ulong**
- **float, double**
- **string**

Arrays


- **byte[], int[], long[], float[], double[]**
- **string[]**

Structs

- **Vector3**

Custom objects

- Any **[Serializable]** C# object.


 **Note:** Internally, **float** values are converted to **double** to match the Web serialization format.


Event Dispatching

To send an event from Unity to JavaScript, use the **EventBridge.DispatchEvent(...)** method.

You can dispatch:

- a simple event without data,
- or include a payload (number, string, array, object, etc.).

 **Note:** For **Event Type** best practices, see [USING TYPED EVENT NAMES](#)

 **Note:** For event naming best practices, see [EVENT NAMES RECOMMENDATIONS](#)

No Payload

```
// Unity
EventBridge.DispatchEvent("ui:clicked");
```

```
// JavaScript
window.addEventListener("ui:clicked", () => {
  console.log("Button click from Unity!");
});
```

Primitive Payload

```
// Unity
EventBridge.DispatchEvent("score:update", 100);
EventBridge.DispatchEvent("msg:hello", "Hello from Unity!");
```

```
// JavaScript
window.addEventListener("score:update", (e) => {
    console.log("Score: ", e.detail);
});

window.addEventListener("msg:hello", (e) => {
    console.log("Message: ", e.detail);
});
```

Custom Payload

Example: 04_CustomPayload

You can dispatch custom C# objects as event payloads. This is useful when you want to send structured data, such as player info, configurations, or any complex state.

To send a custom object:

1. Define a **[Serializable]** class or struct.
2. Pass it as a payload to **DispatchEvent(...)**.

```
// Unity
[System.Serializable]
public class ChatMessage
{
    public string sender;
    public string text;
}

var message = new ChatMessage {
    sender = "Alice",
    text = "Hi!"
};

EventBridge.DispatchEvent("chat:message", message);
```

On the JavaScript side:

```
// JavaScript
window.addEventListener("chat:message", (e) => {
  const msg = e.detail;
  console.log(`${msg.sender}: ${msg.text}`);
});
```

Custom Payload Requirements

- Your custom type **must be marked with [Serializable]**.
- Internally, the object is serialized to JSON using the active **IJsonService**.

If the type is not serializable, **EventBridge** will log an error and throw a **NotSupportedException**.

💡 **Note: WebPontis** uses Unity [JsonUtility](#). You can override the JSON serializer using **EventBridge.SetJsonService(...)** if you need advanced serialization (e.g., dictionaries, polymorphism) — see [Replace the JSON Serializer](#) for more information.

RECEIVING EVENTS

WebPontis allows you to receive events in Unity that were sent from JavaScript.

This can be done in two ways:

- by using the **EventBridgeListener** component (no code required),
- or by calling **EventBridge.AddEventListener(...)** in C# code.

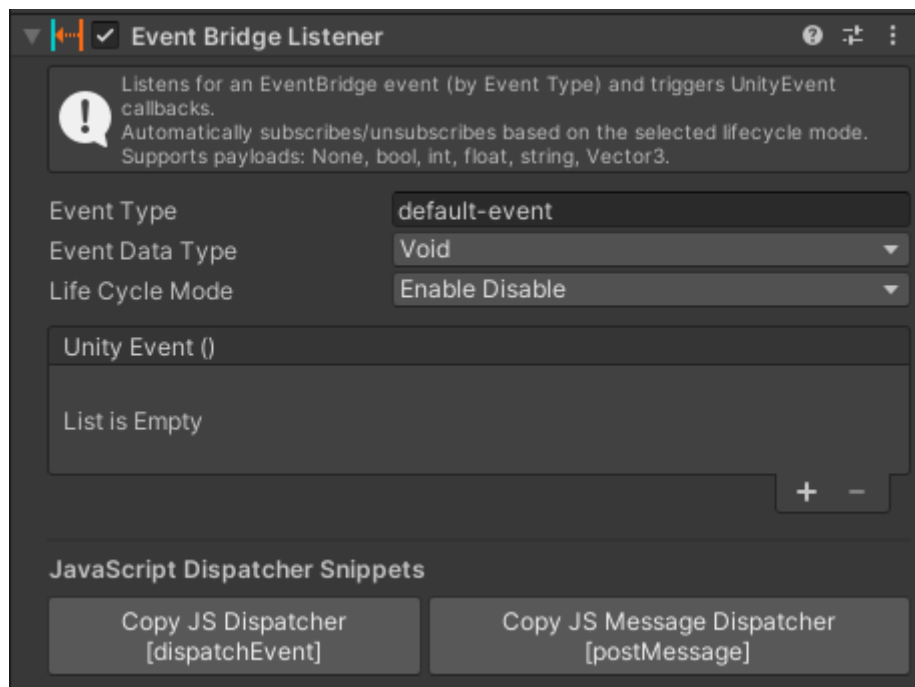
Events can contain no payload (just a trigger), or carry data (string, number, object, etc.).

Let's look at both approaches.

Component-based Approach

Example: 01_QuickStart_NoCode

The **EventBridgeListener** component allows you to respond to events from JavaScript and trigger UnityEvents inside the Inspector.



Life Cycle Mode

The **EventBridgeListener** component includes a **Life Cycle Mode** option that controls **when** the listener is active.

Modes


- **OnEnable / OnDisable (default)**
The listener is registered when the GameObject is enabled, and automatically removed when it is disabled.
This is suitable for UI elements, screens, or other dynamic content.
- **Awake / OnDestroy**
The listener is registered in **Awake()** and stays active until the GameObject is destroyed.
Useful for persistent objects that shouldn't lose their subscriptions (e.g., singletons, root systems).

You can choose the mode via the **Life Cycle Mode** dropdown in the Inspector.

Supported Payload Types

The **EventBridgeListener** component supports the following data types:

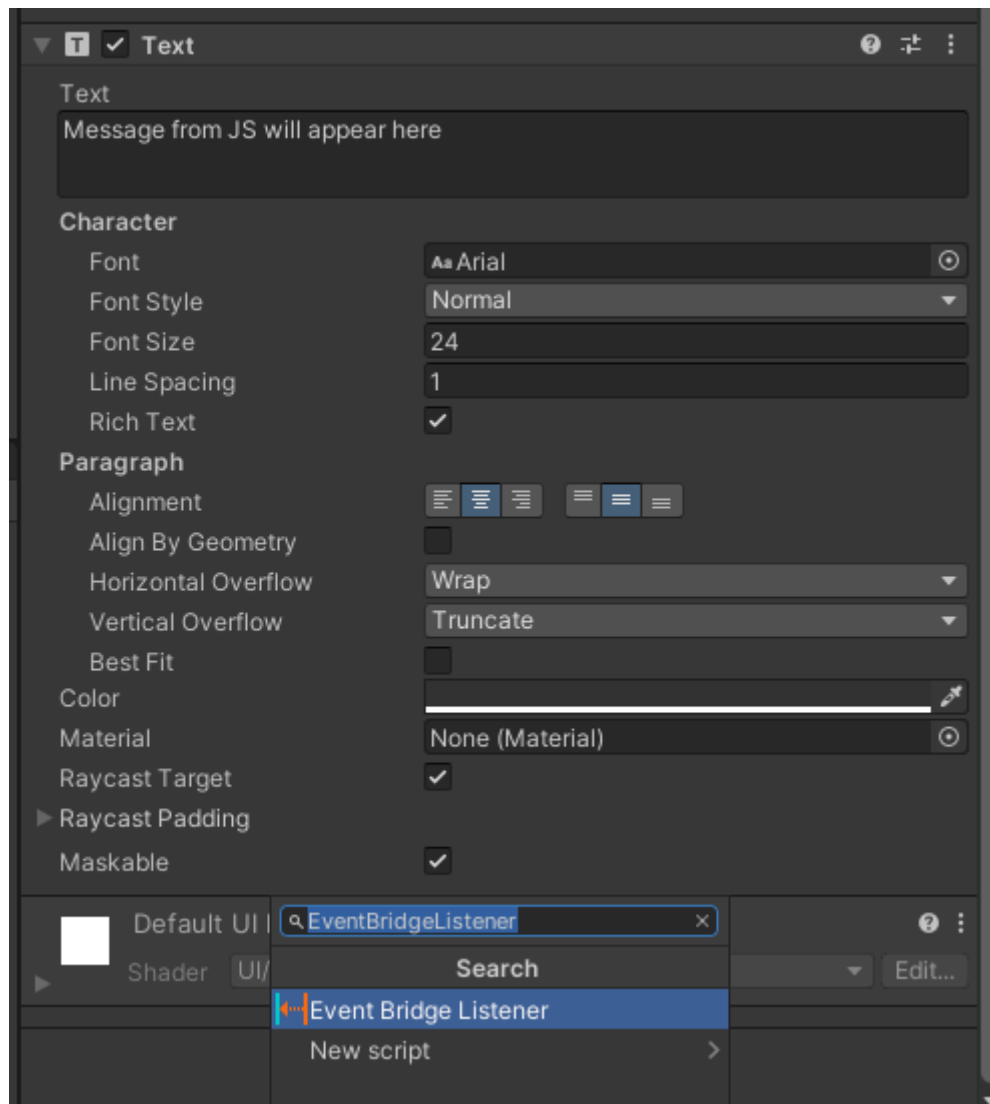
- **None** (no payload)
- **String**
- **Bool**
- **Int**
- **Float**
- **Vector3**

 **Note:** Internally, **float** values are converted to **double** to match the Web serialization format.


 **Note:** For arrays, custom objects, or other advanced data types, use the [Code-based Approach instead](#).

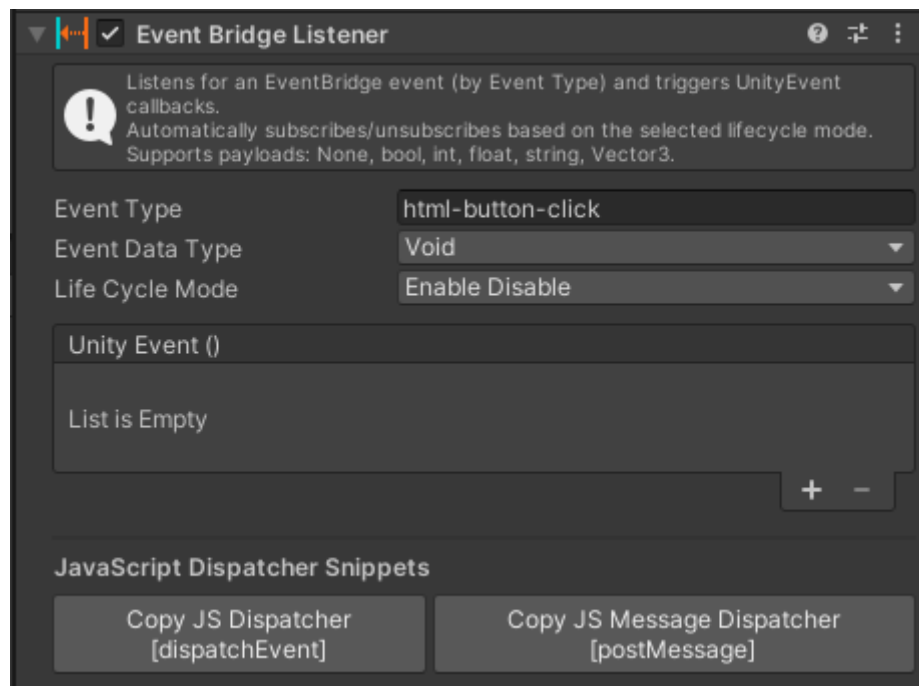
Adding the Component

1. Select a GameObject with a UI **Text**, **TMP_Text**, or any MonoBehaviour method you want to trigger.
2. Add the **EventBridgeListener** component to the same GameObject.

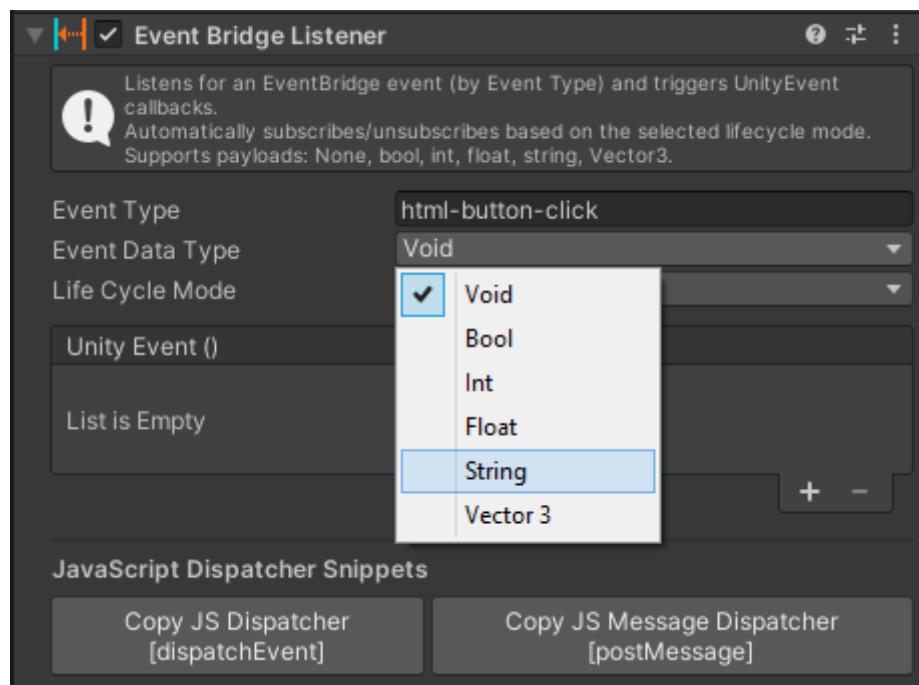


3. In the **Event Type** field, enter: **html-button-click**

 **Note:** For event naming best practices, see [EVENT NAMES RECOMMENDATIONS](#)

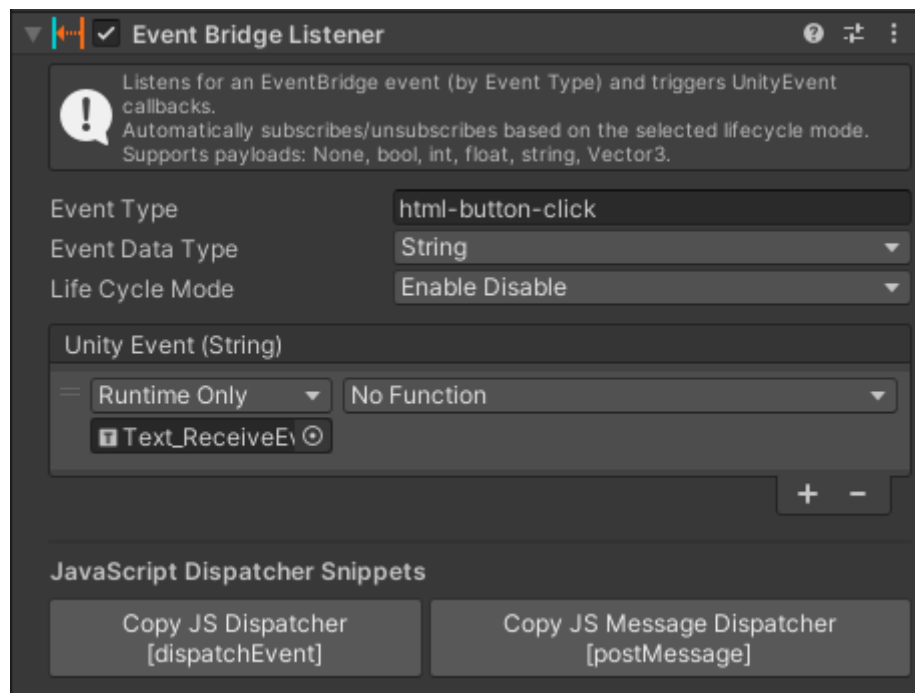


4. Set **Event Data Type** to **String**.

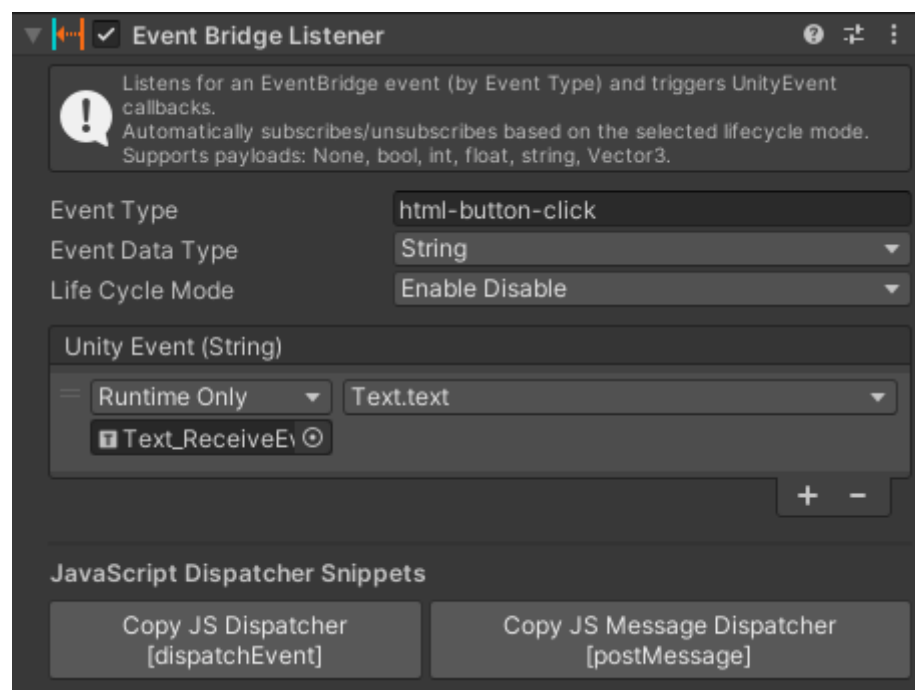


5. In the **Unity Event** section, click "+" to add a response.

6. Drag the **Text** component into the object field.



7. From the function dropdown, choose: **Text** → **text (string)**



When the event is received from JavaScript, the Unity Event will be invoked.

To send the event from **JavaScript**, use:

```
// JavaScript
window.dispatchEvent(new CustomEvent("html-button-click", {
  detail: "Hello from JavaScript!"
}));
```


Code-based Approach

Example: 02_QuickStart_Code, 03_TypedEventNames, 04_CustomPayload

Namespace: IEV0.WebPontis.EventSystem

The code-based approach provides full flexibility and supports all payload types.

Use **EventBridge.AddEventListener(...)** to register a listener for any event from JavaScript, and **EventBridge.RemoveEventListener(...)** to unregister it when it's no longer needed.

 **Note:** Don't forget to remove listeners when they are no longer needed, see [Removing Listeners](#).

Supported Payload Types

No payload

- **None** (empty event)

Primitives

- **bool, sbyte, byte, short, ushort, int, uint, long, ulong**
- **float, double**
- **string**

Arrays

- **byte[], int[], long[], float[], double[]**
- **string[]**

Structs

- **Vector3**

Custom objects

- Any **[Serializable]** C# object.

Receiving Events

No Payload

```
// Unity
EventBridge.AddEventListener("html:ready", () => {
    Debug.Log("HTML is ready!");
});
```

```
// JavaScript
window.dispatchEvent(new CustomEvent("html:ready"));
```

Primitive Payload

```
// Unity
EventBridge.AddEventListener<string>("html:user:name", name => {
    Debug.Log("Username: " + name);
});
```

```
// JavaScript
window.dispatchEvent(new CustomEvent("html:user:name", {
    detail: "UserName"
}));
```

Custom Payload

Example: 04_CustomPayload

```
// Unity
[System.Serializable]
public class ChatMessage
{
    public string sender;
    public string text;
}

EventBridge.AddEventListener<ChatMessage>("chat:message", msg => {
    Debug.Log($"{msg.sender}: {msg.text}");
});
```

JavaScript:

```
// JavaScript
window.dispatchEvent(new CustomEvent("chat:message", {
    detail: {
        sender: "Alice",
        text: "Hello!"
    }
}));
```

Custom Payload Requirements

- Your custom type **must be marked with [Serializable]**.
- Internally, the object is serialized to JSON using the active **IJsonService**.

If the type is **not serializable**, **EventBridge** will **log an error and throw a **NotSupportedException****.

💡 **Note:** **WebPontis** uses Unity [JsonUtility](#). You can override the JSON serializer using **EventBridge.SetJsonService(...)** if you need advanced serialization

(e.g., dictionaries, polymorphism) — see [Replace the JSON Serializer](#) for more information.

Removing Listeners

Listeners registered via `EventBridge.AddEventListener(...)` stay active until explicitly removed.

To remove listener use `EventBridge.RemoveEventListener(...)`.

If you don't remove them, they will **remain in memory** and continue to receive events — even if the related object is destroyed.

This is especially important in dynamic systems (e.g., UI screens, popups, runtime objects).

How to remove listeners

If you registered a listener using a delegate:

```
// Unity
void OnEnable()
{
    EventBridge.AddEventListener<string>("html:user:name",
    HandleUsername);
}

void OnDisable()
{
    EventBridge.RemoveEventListener<string>("html:user:name",
    HandleUsername);
}

void HandleUsername(string name)
{
    Debug.Log("Name: " + name);
}
```

If the listener was anonymous (lambda) (**not recommended**), you **cannot** remove it later:

```
// Cannot be removed
EventBridge.AddEventListener<string>("some-event", data => {
    Debug.Log(data);
});
```

Remove all listeners (not recommended)

You can also clear all registered listeners with

```
// Unity
EventBridge.RemoveAllEventListeners();
```

But this removes **everything** across all event types. Use with caution.

 **Best practice:** Always pair **AddEventListener** with **RemoveEventListener**, especially if used inside **MonoBehaviour.OnEnable / OnDisable**.

EVENT NAMES RECOMMENDATIONS

Consistent, well-structured event names help keep Unity–JavaScript communication clean, scalable, and easy to maintain.

Let your naming match your architecture — whether flat or hierarchical — and focus on clarity and consistency.

Common Naming Styles

1. kebab-case (default)

Most widely used in plain JavaScript and browser events.

```
ui-button-click  
html-loaded  
player-scored
```

2. Namespace-style with colons :

Used to organize events by domain or source. Inspired by conventions from libraries like **socket.io**, **analytics.js**, and some analytics/event tracking systems.

```
unity:ui:button-click  
html:form:submitted  
game:player:joined
```

- Improves readability in large systems
- Useful for filtering or routing by prefix (**unity:***, **html:***, etc.)
- Works well with browser **CustomEvent**

⚠ Note: Avoid mixing styles (e.g. **ui_button:click** or **Unity:Click-Event**)

What JS frameworks use

Framework / Context	Common Event Style
Plain JS / DOM	kebab-case
Vue.js	kebab-case for component events
React (Synthetic Events)	camelCase internally, but kebab-case often used for <code>CustomEvent</code>
socket.io	namespaced events: <code>chat:message</code> , <code>user:login</code>
Firebase / Analytics	kebab-case , sometimes with /
Unity WebGL (default SendMessage)	raw method names (discouraged)

Recommendations

- Use **kebab-case** by default for simplicity and compatibility.
- Use **namespace-style(:)** if you want to organize events into logical groups or modules (e.g. `unity:action:level-start`).
- Keep names lowercase and consistent.
- Make names meaningful, not just structural.

Avoid!

- camelCase (`playerScored`) — not standard in JS events.
- snake_case (`player_scored`) — less common in the front-end.
- Spaces or special characters (`player scored!`, `event@type`)
- Mixing separators: `ui_button:click-event`

USING TYPED EVENT NAMES

Namespace: `IEV0.WebPontis.EventSystem.Events`

Example: `03_TypedEventNames`

To avoid hardcoding event names as strings and make your code more maintainable and type-safe, you can define your own event types by implementing the **IInboundEvent** and **IOutboundEvent** interfaces.

Each typed event must provide a unique **Type** identifier — this will be used as the actual event name in the system.

 **Note:** For naming best practices, see [EVENT NAMES RECOMMENDATIONS](#)

IOutboundEvent

Use this interface for events **sent from Unity to JavaScript** (outbound):

```
// Unity
public struct EventOut_PlayerScored : IOutboundEvent
{
    public string Type => "game:player:scored";
}
```

Then dispatch the event like this:

```
// Unity
EventBridge.DispatchEvent<EventOut_PlayerScored>();
```

Or with data:

```
// Unity
EventBridge.DispatchEvent<EventOut_PlayerScored, int>(100);
```

InboundEvent

Use this interface for events **received from JavaScript** (inbound):

```
// Unity
public struct EventIn_HtmlButtonClick : IInboundEvent
{
    public string Type => "html:button:click";
}
```

Then listen for this event like so:

```
// Unity
EventBridge.AddEventListener<EventIn_HtmlButtonClick>(() =>
{
    Debug.Log("HTML button was clicked!");
});
```

If the event includes data:

```
// Unity
EventBridge.AddEventListener<EventIn_HtmlButtonClick,
string>(message =>
{
    Debug.Log("Received from JS: " + message);
});
```

Benefits of Typed Events

- No hardcoded strings — safer refactoring
- Centralized control over all event names
- Cleaner code in larger projects
- Clear semantic meaning (**EventOut_***, **EventIn_***)

Why Use struct Instead of class?

Typed events should be implemented as **struct** (value types), not **class**. This is by design:

The **EventBridge** API uses generic constraints like **where T : struct, IOutboundEvent**, which ensures that:

- **No heap** allocation is required
- There's **no need to instantiate** event types manually

! Note: Only **struct**-based events are supported.

Do not use **class** for **IInboundEvent** or **IOutboundEvent** implementations.

! Note: Typed event names (**IInboundEvent**, **IOutboundEvent**) are only supported in the **code-based API**.

CONFIGURATION

EventBridge allows you to customize its behavior at runtime.

You can configure [basic options](#): logging, event routing, and messaging security — either by using a [component](#) or via [code](#).

Some **advanced options** can only be **changed only from code**, such as:

- Replacing the internal [logger](#)
- Injecting a custom [JSON serializer](#) for the **custom objects**
- Customizing the **event message format** for JS
- Injecting a custom [InteropService](#) (for non WebGL/WebGPU platforms)

Basic options


The following settings affect how the **EventBridge** system behaves:

Log Level

Controls how much debug information is written to the console.

Available levels:

- **Debug** – shows all activity including internal conversions and message flow.
- **Info (default)** – general operations like sent/received events.
- **Warning** – potential misconfigurations or unexpected behaviors.
- **Error** – only critical failures.

 **Note:** Use **Debug** while developing. Switch to **Warning** or **Error** in production.

 **Note:** To customize how logs are handled, see [CUSTOM LOGGER](#)

Channels

Determines which JavaScript mechanisms will be used when sending / receiving events.

Available options:

- **Internal** (default) – uses `UnityInstance.Module.WebPontis.EventBridge.Dispatcher.dispatchEvent` and `UnityInstance.Module.WebPontis.EventBridge.Dispatcher.addEventListener` for event dispatching and receiving.

Example: [06_Channels_Internal](#)

Best for: Internal browser apps that interact directly with Unity through `Module.WebPontis`.

Use when:

- ❖ You are building tightly-coupled logic within the same HTML page or WebGL wrapper.
- ❖ You want **maximum performance** with **zero DOM overhead**.
- ❖ You require **complete event isolation** (no leakage to DOM or other frames).

- **Events** (default) – Uses the standard DOM system via `window.dispatchEvent` and `window.addEventListener`.

Example: [01_QuickStart_NoCode](#); [02_QuickStart_Code](#); [03_TypedEventNames](#); [04_CustomPayload](#);

Best for: Most applications that want to observe Unity events in a standard, browser-friendly way. React, Vue, or any framework using DOM event listeners.

Use when:

- ❖ You prefer the **native browser event model**.
- ❖ You want to **debug events easily** via **DevTools → Event Listeners**.
- ❖ You have **multiple JS modules** or libraries that should react to Unity events.
- ❖ You want a **lightweight and loosely-coupled integration**.

💡 **Note:** This is often the **most straightforward option** — readable, observable, and easy to connect to analytics or UI layers.

- **Messages** (default) – Uses `window.postMessage` and `window.addEventListener("message"...` for communication from Unity to the same window/frame.

Example: [07_Channels_Messages](#); [05_CustomMessageFormat](#);

Best for: Integrating with third-party scripts, sandboxed iframes, or message-based architectures.

Use when:

- ❖ You need cross-component communication, but not between parent/child frames.
- ❖ You want to **decouple** event emission from listeners.
- ❖ You need a **central message router** or controller.
- ❖ You're working with **browser extensions** or **cross-layer communication**.
- ❖ You want to **customize the message structure**.

💡 **Note:** While powerful, `postMessage` requires careful origin filtering and parsing. It's a good choice for **embedded applications** or **cross-context messaging**.

- **MessagesParent** (default) – Uses `window.parent.postMessage` and `window.addEventListener("message"...` for communication from Unity to parent window.

Example: [08_Channels_MessagesParent](#);

Best for: Embedding Unity inside an **iframe** and communicating with the parent page.

Use when:

- ❖ Unity is **embedded via <iframe>**, and coordination with the host page is required.
- ❖ The **parent** page **acts as a controller**, message hub, or analytics collector.

❖ Unity sends telemetry, commands, or state updates to the outer page.

💡 **Note:** Ensure proper origin settings (**SetTargetOrigin**, **SetOrigins**) for security. This channel is essential when using **cross-frame communication**, **dashboard integration**, or browser-based orchestration.

💡 **Note:** By default, **all channels are enabled**. This ensures maximum compatibility with different setups **during development**, including iframes and embedded views.

💡 **Note:** In code these values can be combined using the bitwise OR (**|**) operator.

```
EventBridge.Configuration.SetChannels(ChannelType.Events |  
ChannelType.Messages);
```

💡 **Note:** In production, recommended **disabling unused channels** using **SetChannels(...)** to:

- Reduce overhead
- Avoid duplicate event dispatching
- Improve security and clarity

Target Origin (for outgoing events)

Output Channels: **Messages**, **MessagesParent**

Defines the allowed destination when Unity sends messages (In js via **postMessage**).

Examples:

- "*" (**default**) – allow sending to any origin (not recommended for production)
- "https://example.com" – restrict to a specific domain

💡 **Note:** Important for browser security and content isolation.

Allowed Origins (for incoming events)

💡 **Note:** Important for browser security and content isolation.

Output Channels: **Messages**, **MessagesParent**

Restricts which browser origins can send events to Unity via (In js via **postMessage**).

Examples:

- ["*"] (**default**)
- "https://my.app.com"
- "https://localhost:3000, https://staging.app.com"

If security is a concern, you should restrict this list to known trusted domains using **SetOrigins(...)**.

💡 **Note:** if the list contains "*" – meaning **all origins are accepted!**

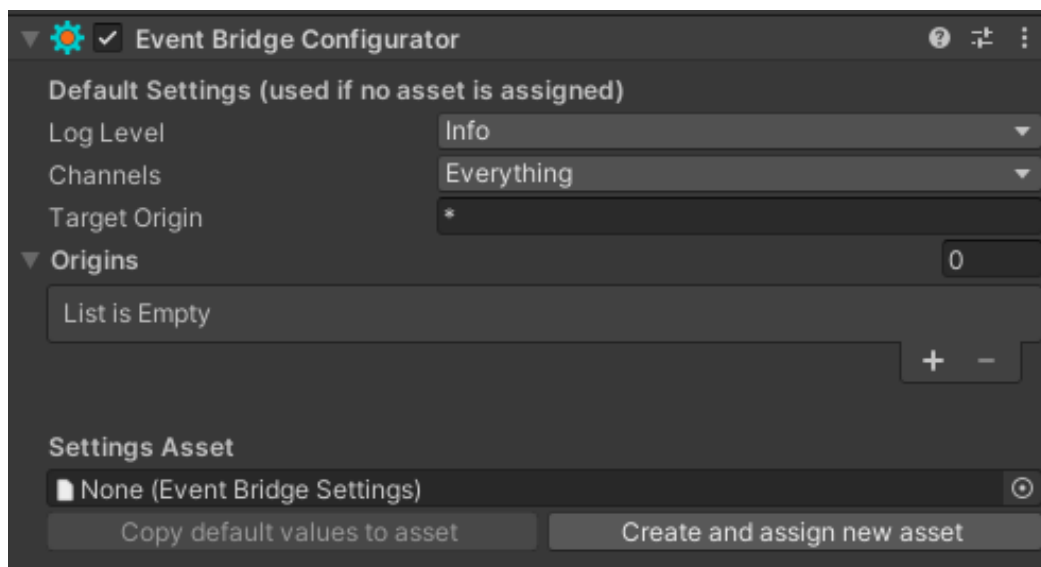
No-Code Approach

Example: 01_QuickStart_NoCode

Prefab: WebPontis / Prefabs / EventSystem / EventBridgeConfigurator

The recommended way to configure **WebPontis** in the No-Code approach is by adding the ready-to-use **EventBridgeConfigurator** prefab to your scene.

This prefab automatically applies configuration at runtime using the built-in component.




Options which can be configured

- [Log level](#)
- [Channels](#)
- [Target origin](#) for outgoing postMessage
- [Origins](#) for incoming messages

Using the Prefab

1. In your Unity project, locate the prefab at:
WebPontis / Prefabs / EventSystem / EventBridgeConfigurator
2. Drag it into your scene (e.g., into a System or Managers GameObject).
3. Select the prefab in the scene and configure it via the Inspector:
 - Use local settings, or
 - Assign a reusable **EventBridgeSettings** asset.

 **Note:** If a settings asset is assigned, it overrides the values in the Inspector.

Optional: Create or Edit Settings Asset

You can:

- Copy current Inspector values into an assigned asset,
- Create a new **EventBridgeSettings** asset directly from the Inspector.

This makes it easy to reuse configuration across scenes or environments.

Code-based Approach

Example:

Namespace: `IEV0.WebPontis.EventSystem`

You can configure WebPontis programmatically using the static API:

```
// Unity
EventBridge.Configuration
```

This approach gives you full control and is useful when:

- Settings depend on runtime conditions (e.g., environment, user profile, platform)
- You don't want to rely on scene-based configuration
- You need to override or extend behavior beyond the Inspector
- You need to setup advanced settings

Log Level

Controls how much debug information is written to the console.

```
// Unity
Debugger.SetLogLevel(LogLevel.Debug);
```

Available values:

- **LogLevel.Debug,**
- **LogLevel.Info,**
- **LogLevel.Warning,**
- **LogLevel.Error.**

See [Log Level](#) for more info.

Channels

You can combine multiple options using the bitwise OR (|) operator.

```
// Unity
EventBridge.Configuration.SetChannels(
    ChannelType.Events | ChannelType.Messages
);
```

See [Channels](#) for more info.

Target Origin (for outgoing events)

```
// Unity
EventBridge.Configuration.SetTargetOrigin("https://example.com");

// or allow all:
EventBridge.Configuration.SetTargetOrigin("*");
```


See [Target Origin \(for outgoing events\)](#) for more info.

Allowed Origins (for incoming events)

```
// Allow only specific trusted domains:
EventBridge.Configuration.SetOrigins(new[]
{
    "https://a.com",
    "https://b.com"
});

// Allow all origins (unrestricted):
EventBridge.Configuration.SetOrigins(new[] { "*" });
```

See [Allowed Origins \(for incoming events\)](#) for more info.

 **Note:** if the list contains "*" – meaning **all origins are accepted!**

💡 **Note:** If you pass `null`, the method will log a warning and skip the update. Invalid URLs will result in an error and will not be applied.

Replace the JSON Serializer

Example: [09_CustomJsonService](#)

By default, **WebPontis** uses Unity's built-in [JsonUtility](#) for serializing and deserializing **custom objects**.

This serializer is fast and works well for simple data structures, but it has several **important limitations**.

Limitations of JsonUtility

- Only supports **public fields** (not properties)
- No support for **dictionaries**
- No support for **nested polymorphic types**
- No support for **custom converters**
- Cannot handle complex or dynamic JSON structures

Example: The following object will not serialize correctly with **JsonUtility**:

```
[System.Serializable]
public class PlayerData
{
    public string id;
    public Dictionary<string, int> stats; // Not supported!
}
```

When to use a custom serializer like Newtonsoft.Json

If you need:

- Full support for complex object graphs
- Dictionary serialization
- Advanced customization (e.g., **JsonProperty**, custom converters)
- Polymorphic deserialization (base class → derived types)
- Compatibility with existing JSON formats or APIs

Then you should replace the default JSON serializer.

How to replace it

Implement your own **IJsonService**:

```
public interface IJsonService
{
    string ToJson<T>(T data);
    T FromJson<T>(string json);
}
```

Example using **Newtonsoft.Json (Json.NET for Unity)**:

```
public class NewtonsoftJsonService : IJsonService
{
    public string ToJson(object obj) =>
    JsonConvert.SerializeObject(obj);
    public T FromJson<T>(string json) =>
    JsonConvert.DeserializeObject<T>(json);
}
```

Apply it during initialization:

```
EventBridge.Configuration.SetJsonService(new
NewtonsoftJsonService());
```

Switching back to Unity's default

If needed, you can switch back to JsonUtility at any time:

```
EventBridge.Configuration.SetJsonService(new UnityJsonService());
```

CUSTOM LOGGER

Example:

Namespace: `IEVO.WebPontis.Logging`

By default, **WebPontis** uses **WebPontisLogger**, which logs messages to the Unity Console using **Debug.Log**, **Debug.LogWarning**, and **Debug.LogError**.

While this is convenient during development, there are cases where a **custom logger** is preferred.

When to use a custom logger

You may want to replace the default logger if:

- You need to, cloud, or external logging system
- You want to **filter or format logs differently**
- You're building a tool or framework and want to **integrate with your own logging pipeline**
- You need to **display logs inside the UI** of your Unity application (e.g., a debug overlay or console)

How to replace it

Implement your own **ILogger**:

```
public interface ILogger
{
    void Debug(string message);
    void Info(string message);
    void Warning(string message);
    void Error(string message);
    void SetLogLevel(LogLevel level);
}
```

Example custom logger:

```

public class CustomLogger : ILogger
{
    private LogLevel _level = LogLevel.Debug;

    public void SetLogLevel(LogLevel level) => _level = level;

    public void Debug(string message) => Log(LogLevel.Debug,
message);
    public void Info(string message) => Log(LogLevel.Info,
message);
    public void Warning(string message) => Log(LogLevel.Warning,
message);
    public void Error(string message) => Log(LogLevel.Error,
message);

    private void Log(LogLevel level, string message)
    {
        if (_level > level || _level == LogLevel.None)
            return;

        switch (level)
        {
            case LogLevel.Debug:
            case LogLevel.Info:
                UnityEngine.Debug.Log($"Custom log {message}");
                break;
            case LogLevel.Warning:
                UnityEngine.Debug.LogWarning($"Custom log
{message}");
                break;
            case LogLevel.Error:
                UnityEngine.Debug.LogError($"Custom log
{message}");
                break;
        }
    }
}

```

Apply it at runtime:

```
DebugLogger.SetLogger(new CustomLogger());
```

Reset to default

```
DebugLogger.SetLogger(new WebPontisLogger());
```

CUSTOM INTEROP SERVICE

Advanced Use Case – Cross-platform or native integrations.

WebPontis is designed with flexibility in mind. While it works out of the box in **WebGL/WebGPU** via JavaScript interop, the entire communication layer is abstracted through the **IInteropService** interface.

You can inject your own implementation of this interface to route events through **any external environment** — such as:


- Native plugins (C/C++ integration)
- Desktop applications
- WebSocket communication
- Custom simulation tools

How it works

Internally, **WebPontis** uses one of two built-in implementations:

Implementation	Used in
WebInteropService	Automatically used in WebGL builds
LoopbackInteropService	Used in the Unity Editor or non-WebGL platforms

These implementations are selected automatically based on the build platform.

 **Note:** In Editor mode, **LoopbackInteropService** simulates browser communication inside Unity, which allows testing without building WebGL.

Replacing the Interop Layer

You can override the interop service at runtime:

```
EventBridge.Configuration.SetInteropService(new  
MyCustomInteropService());
```

Your custom class must implement:

```
public interface IInteropService
{
    void DispatchEvent(string type, int dataType, byte[] data, int
dataLength);
    void AddEventListener(string type, Action<string, int, IntPtr,
int> listener);
    void RemoveEventListener(string type, Action<string, int,
IntPtr, int> listener);
    void SetTargetOrigin(string origin);
    string GetTargetOrigin();
    void SetOrigins(string[] origins);
    string[] GetOrigins();
    void SetChannels(byte channels);
    byte GetChannels();
}
```

This allows full control over how data is sent and received between Unity and external systems.

Example Use Cases

WebSocket-based interop

You can implement an **IInteropService** that:

- Sends serialized events via WebSocket
- Receives binary payloads from a server or browser extension
- Dynamically controls connected clients

This is ideal for **local development**, **editor tools**, or **multi-instance testing**.

Native plugin bridge

In native or embedded environments, you might want to:

- Bridge Unity with native C/C++ code
- Use **DllImport** or **COM, IPC, shared memory**, etc.
- Inject messages directly into **EventBridge** from a host app

If you override the interop service, you are fully responsible for encoding, routing, and life cycle of all events.

CUSTOM MESSAGE FORMAT

Example: [05_CustomMessageFormat](#)

Advanced Use Case – Customize how message events are structured in JavaScript.

By default, **WebPontis** sends outbound events (from Unity to JavaScript) as either:

- **CustomEvent** when using the **Events** channel (**window.dispatchEvent**)
- A plain object when using the **Messages** or **MessagesParent** channels (**window.postMessage**).

Default message format (postMessage)

When using **window.postMessage(...)**, Unity sends this structure by default:

```
{
  "type": "your:event:type",
  "data": { ... }
}
```

This format is widely supported and easy to parse — especially in event buses, iframe messaging, and structured APIs.

Why customize it?

You may want to:

- Match an existing message protocol (e.g. **detail, payload, action**)
- Add metadata like timestamp, sessionId, source, version
- Use shorthand keys (**t, d**) to reduce size
- Wrap messages into a top-level envelope for unified parsing

How to customize message structure

1. Create a new **.jslib** file in your Unity project (e.g., **WebPontisCustomMessageFormat.jslib**).
2. Define a method that overrides the message formatters:

```
mergeInto(LibraryManager.library, {

  InitCustomMessageFormat: function() {
    if (!Module.WebPontis || !Module.WebPontis.EventBridge)
    return;

    Module.WebPontis.EventBridge.OutboundMessageFormatter =
function(type, data) {
  return {
    eventName: type,
    payload: data
  };
};

    Module.WebPontis.EventBridge.InboundMessageFormatter =
function(event) {
  var raw = event.data;
  return {
    type: raw && raw.eventName,
    data: raw && raw.payload
  };
};

    Module.WebPontis.Logger.info('[EventBridge] Custom message
formatters registered.');
```


3. Create C# class **WebPontisCustomMessageFormat.cs**

```
using System.Runtime.InteropServices;
using UnityEngine;

public class WebPontisMessageCustomFormat : MonoBehaviour
{
    [DllImport("__Internal")]
    private static extern void InitCustomMessageFormat();

    private void Awake()
    {
        InitCustomMessageFormat();
    }
}
```

4. Call the JS function **InitCustomMessageFormat()** from C# using **[DllImport("__Internal")]**.
5. Place this MonoBehaviour in your startup scene to ensure the formatters are applied before any events are sent or received.

 **Note:** This only affects **Messages** and **MessagesParent** channels (**postMessage**-based). Does **not** affect **CustomEvent** dispatch (used by the **Events** channel).

DEBUGGING TOOLS

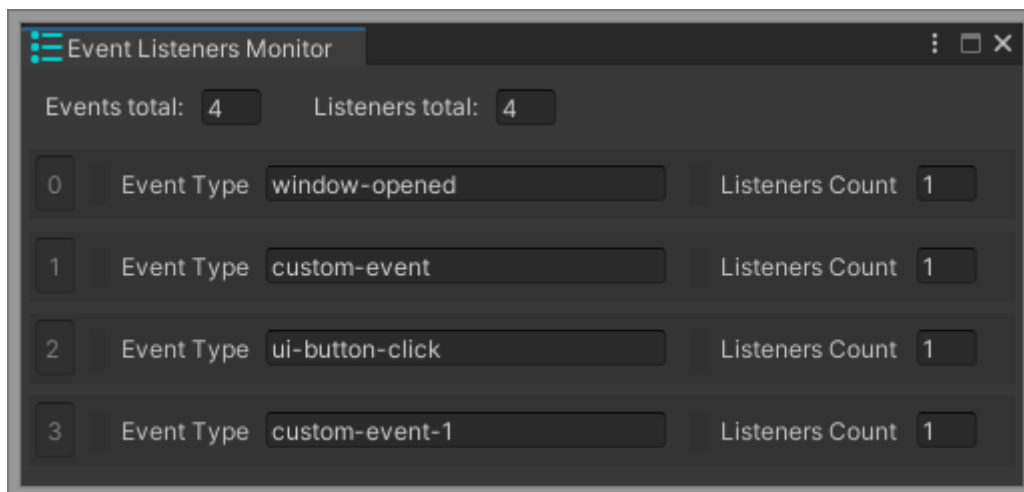
Menu: Tools > WebPontis

WebPontis includes several powerful debugging tools to help you monitor, simulate, redirect and trace events during development.

EventListenersMonitor

Menu: Tools > WebPontis > Event Listeners Monitor

Displays all currently registered event types and how many listeners are active per event.



Useful for:

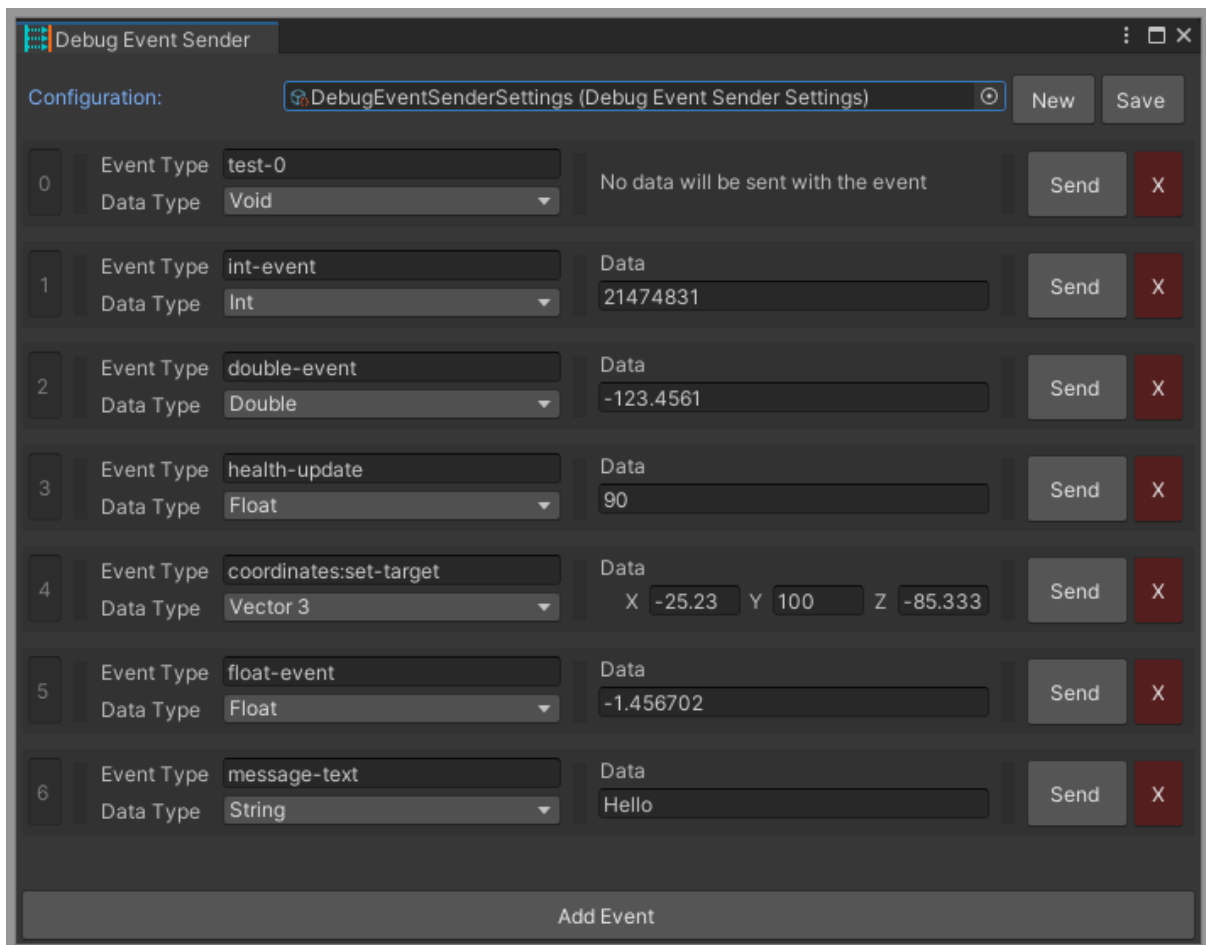
- Tracking listener leaks
- Understanding what Unity is currently "listening" for
- Verifying component registration (e.g., **EventBridgeListener**, code listeners)

DebugEventSender

Menu: Tools > WebPontis > Debug Event Sender

Allows you to manually send events to Unity as if they were coming from JavaScript. You can define the event type, select a payload type (**string**, **bool**, **int**, **float**, **Vector3**, or custom object), and trigger dispatch directly from the Editor.

You can also create and save **custom configurations** to reuse common test events.



Useful for:

- Verifying **EventBridgeListener** setup
- Quickly testing how Unity reacts to specific events
- Replaying known test cases
- Organizing a set of reusable test messages

 **Note:** Can be used together with **WebPontis Forwarder** to simulate and inspect events between Unity Editor and the browser — even without building a WebGL app.

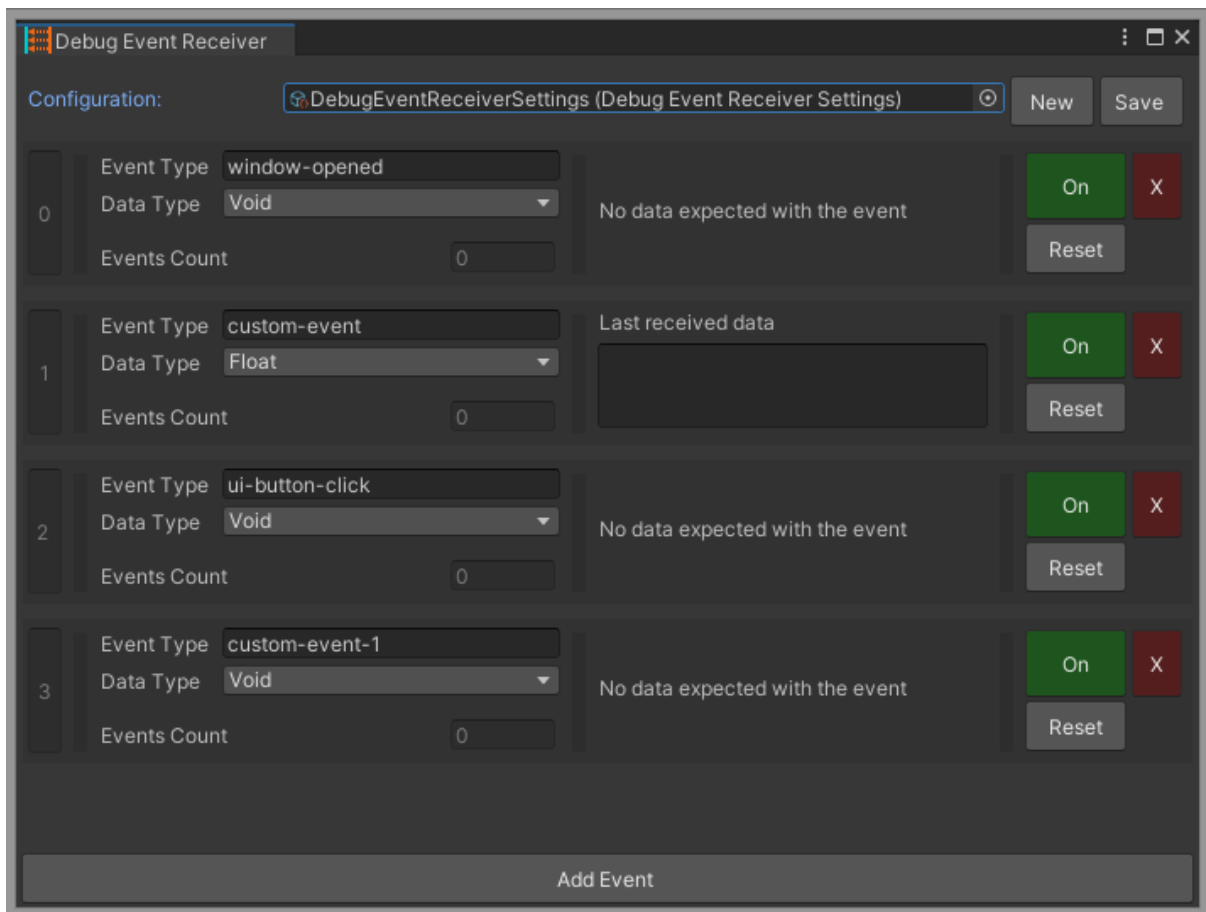
DebugEventReceiver

Menu: Tools > WebPontis > Debug Event Receiver

Displays all incoming events received by Unity in real time.

No coding required — just open the window and watch events as they arrive.

You can also define **receiver configurations** to filter or group event types, improving focus during debugging.



Useful for:

- Verifying **EventBridgeDispatcher** setup
- Inspecting event types, data types, and payloads
- Debugging Unity event handlers
- Validating deserialization logic

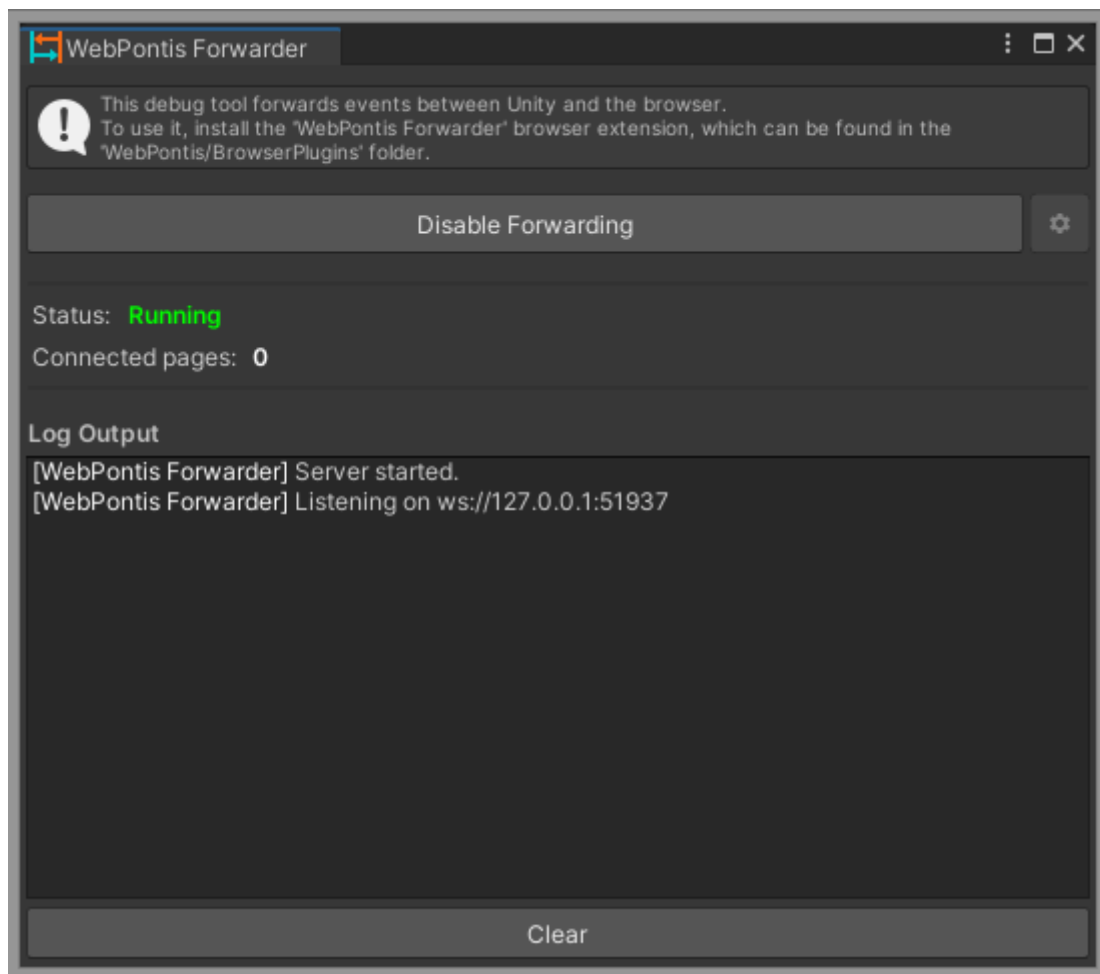
Note: Works seamlessly with **WebPontis Forwarder**, allowing you to monitor real messages redirected from a live browser.

WebPontis Forwarder

Menu: Tools > WebPontis > WebPontis Forwarder

WebPontis Forwarder is a **powerful development utility** that allows you to **bridge live browser messages directly into the Unity Editor**, without needing to build or run a WebGL version of your app.

It communicates over WebSocket (localhost) and can **forward events in both directions** (Editor ↔ Browser).



How to use it

1. **Open Unity Editor.**
2. Go to **Tools > WebPontis > WebPontis Forwarder.**
3. In the Forwarder window, click **"Enable Forwarding"**.
4. Open your **browser** and navigate to your WebGL build or testing HTML page
5. Click the **WebPontis Forwarder extension icon**. You should see the detected context appear in the list.
6. In the target context section click **"Activate"**

Supported flows

Direction	Description
Browser → Unity	Test JS events directly in the Editor
Unity → Browser	Forward Unity events to browser , no WebGL build needed
Manual Simulation	Works perfectly with DebugEventSender and Receiver

Useful for:

- Testing real browser scripts and HTML integration without building
- Using browser developer tools (console, scripts) live against Unity
- Developing JS-heavy flows while staying in Unity Editor
- Debugging multi-iframe or multi-origin scenarios

Browser Integration

To enable browser ↔ Unity communication, you **need to install** the **WebPontis Forwarder browser extension**.

The extension is included with the plugin and supports:

- **Chromium-based browsers** (Chrome, Edge, etc.)
- **Firefox**

Plugin Location

You can find the extension source files in your project under:

WebPontis / BrowserPlugins / WebPontisForwarder /

Inside this folder, you'll find separate versions for each browser:

- **Chromium/** - for Chrome, Edge, etc.
- **Firefox/** - for Firefox

Installation Instructions

For **Chromium-based** browsers (Chrome, Edge):

1. Open **chrome://extensions**
2. Enable **Developer mode**
3. Click **Load unpacked**
4. Select the folder:
WebPontis/BrowserPlugins/WebPontisForwarder/Chromium/
5. Reload your Unity WebGL page

For **Firefox**:

1. Open **about:debugging**
2. Click **This Firefox**
3. Click **Load Temporary Add-on**

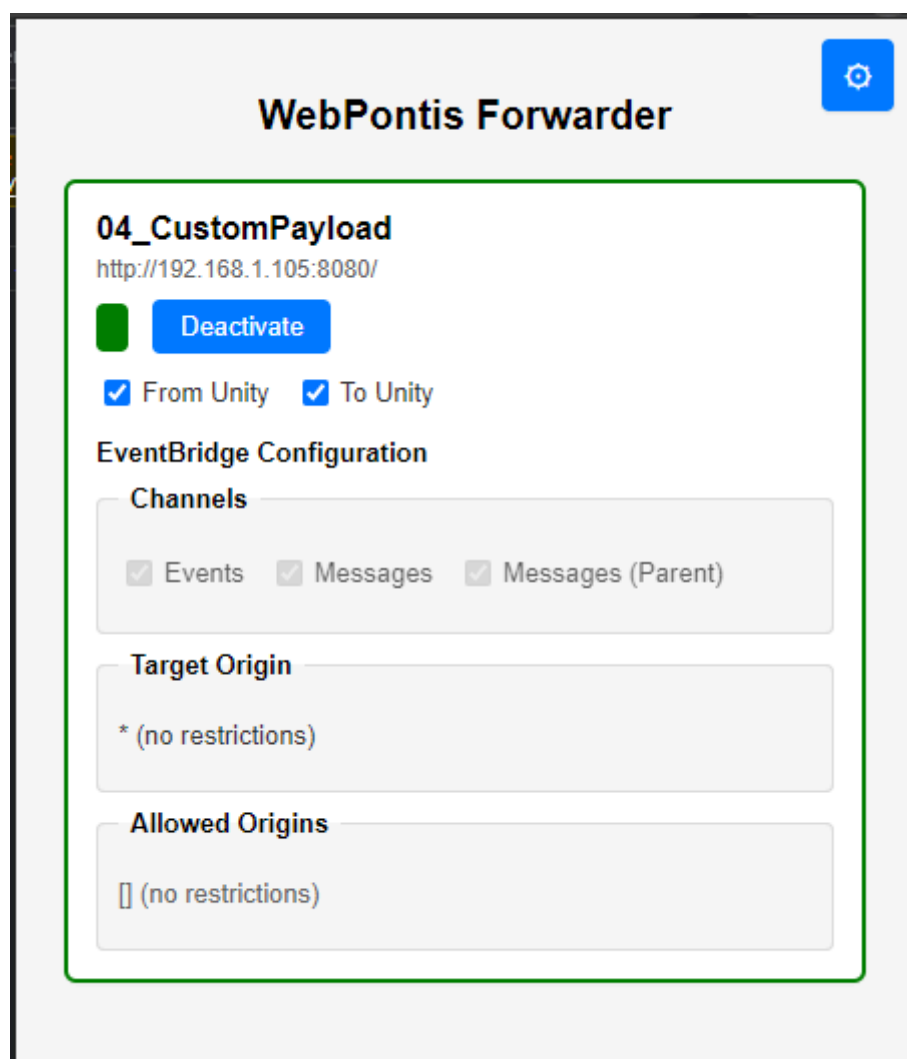
4. Select the file:
WebPontis/BrowserPlugins/WebPontisForwarder/Firefox/manifest.json
5. Reload your Unity WebGL page

Usage

Once the extension is installed and your Unity Editor is running the WebPontis Forwarder server, you can use the **Browser Plugin popup** to control **which html page or iframes** should redirect events **to** and **from** Unity.

Each detected context (e.g. a tab, iframe, or standalone HTML page) will appear as a separate section in the popup.

You can activate or deactivate message forwarding for each one **individually**. This allows selective message redirection, even across multiple environments or testing pages.




Browser Plugin Settings

In most cases, **you don't need to change these settings** — the plugin works out of the box with the default port and message format.

However, you might want to adjust them if:

- You're using a **custom message format**
- You're running the Unity WebPontis Forwarder on a **non-default port**

Click the  **gear icon** in the popup to open the **WebPontis Forwarder Settings** panel.


From there, you can:

- Change the **connection port**
- Define a **custom outbound message format** (Unity → JS)
- Define a **custom inbound message format** (JS → Unity)

These settings are especially useful when working with non-standard message structures or embedding Unity into complex browser environments.

Port Settings

- **Port:**
Defines which port the browser extension should use to connect to the Unity Editor WebSocket server.

 **Note:** This **must match** the port set in the Unity Editor (Tools > Web Pontis > WebPontis Forwarder).
- **Save:** Applies the new port and restarts connection.
- **Reset to Default:** Restores default port (**51937**).

WebPontis Forwarder Settings

Port Settings

This port must match the one set in Unity's WebPontis Forwarder.
Changing it requires reconnecting both sides.

Port:

Save

Reset to Default

Message Formatters

Outbound Formatter

Defines how the message will be sent using `postMessage`.
Use `$type` and `$data` as placeholders.
Example: `{ "eventName": "$type", "payload": "$data" }`

Template (JSON):

```
{
  "type": "$type",
  "data": "$data"
}
```

Inbound Formatter

Maps fields from `event.data` to extract **type** and **data**.
Supports dot notation for nested fields.
Example: `{ "container": { "eventName": "...", "payload": {...} } }`
→ Type Path: `container.eventName`
→ Data Path: `container.payload`

Type Path:

Data Path:

Apply Formatters

Reset to Default

Message Formatters

You can control how messages are **sent from Unity to JS** (Outbound) and **received from JS into Unity** (Inbound) via `postMessage`.

Outbound Formatter

Defines the message structure Unity will send to the browser.

- **Template (JSON):**

A customizable JSON object template where:

- **\$type** is replaced with the event name
- **\$data** is replaced with the serialized payload

Example:

```
{
  "eventName": "$type",
  "payload": "$data"
}
```

Inbound Formatter

Maps the structure of incoming browser messages to extract the event **type** and **data**.

- **Type Path:**

Path to the field inside `event.data` that holds the event name.

Supports **dot notation** for nested fields (e.g., `container.eventName`)

- **Data Path:**

Path to the field holding the payload data.

Also supports nested fields.

Example:

```
{
  "container": {
    "eventName": "chat:send",
    "payload": { "message": "Hello" }
  }
}
```

Controls

- **Apply Formatters:** Saves and activates the current format templates and paths.
- **Reset to Default:** Restores standard behavior:

CONTACTS

Web site: <https://ievo.games/>

E-mail: contact@ievo.games (subject: WebPontis)