DRAFT

Note: This content represents the current state of the HSR platform, its applications, and workflows. The platform is under continuing development, and this content may require periodic updating. This content should not be interpreted as official product documentation.

This guide provides an overview of how eventing works in HSR and some basic examples of native, local, and networked events.

Overview

Event definition

An **event** is a message from one script component to another, often containing some set of data as a payload. When you create an event, you create the following:

- The event definition defines the name of the event, its scope, and the event payload.

- An event listener is an object in a script that defines what happens when the script receives the event. Each script that responds to the event must include a listener.
- An event message can be triggered in a script as needed and delivered to all event listeners within scope.

A simple example:

- A player enters a Trigger Zone.
- The script attached to the Trigger Zone sends the `incrementScore` event to the `Scoreboard` entity.
- The event listener in the `Scoreboard` entity for the `incrementScore` event retrieves the current value from the Text component in the `Scoreboard` entity, adds 1 to the value, and writes the updated value back to the Text component.

The event definition is stored in the `Scoreboard` script. The event listener is defined in the script.

The event definition is imported from the `Scoreboard` script into the Trigger Zone script. The event message is defined in the script.

Events and networking

For more information, see Overview of Ownership TBD.

Event Types

The following are the basic types of events in HSR.

Event type	Description	Usage Notes
Native	These events are sent from the native runtime engine to the scripting layer. Events like `OnWorldUpdateEvent` provide low-level functionality to which script methods can subscribe. Native events are always sent at the entity level and then propagated to the component level.	Native events are triggered as part of normal script and platform execution. They can be subscribed to by creators.

Local	Custom local events can be sent to other entities on the local client. They do not cross to other clients on the network. ● `LocalEvent` events are useful for immediate user feedback related to user actions, such as playing audio or triggering animations. ● By nature of executing entirely on the local client, these events are reliably synchronous.	**Note**: Any custom event message that doesn't require networking should use local events to eliminate latency.
Network	Custom network events can be sent to specified entities or broadcast to all entities across the network.	Use networked events for anything that requires cross-client communication, such as dealing damage to another player.

Basic Tips

- An event is sent from the client that owns the entity from where it is called.
    - By default, most entities are owned by the server.
- Local events are for sending messages to other entities on the same client. Local events never cross the network. Local events are very low-latency.
    - Use local events wherever possible.
    - Use local events for sending messages between entities that are specific to the player.
    - **Tip**: If your event message is to update properties of a component or entity, you may be able to circumvent sending the local event by making direct modifications to properties on other local entities and their components.
- Network events are for sending messages to entities and components on other clients on the network. Network events can be slow.
    - Network events are sent reliably by default, which means an acknowledgment is returned. However, if there are network issues, this can double the delay.
    - Use network events for changes to state that affect the global environment.

Event execution

Non-deterministic order of execution

While order of execution of events within a script can be somewhat managed, there is no guarantee for the order of execution of events between scripts. If two entities with attached scripts are created at world startup, you cannot guarantee the order in which their `OnEntityStart` events are executed.

Similarly, you cannot determine the order of receipt of events.

Within a single script, however, you can expect events to execute in a more predictable order on the client and the server.

Default native event execution

The following script applies some minor modifications to the default script template to report to the console when some native events have executed.

Tip: Code added to the default template is denoted with `// added`.

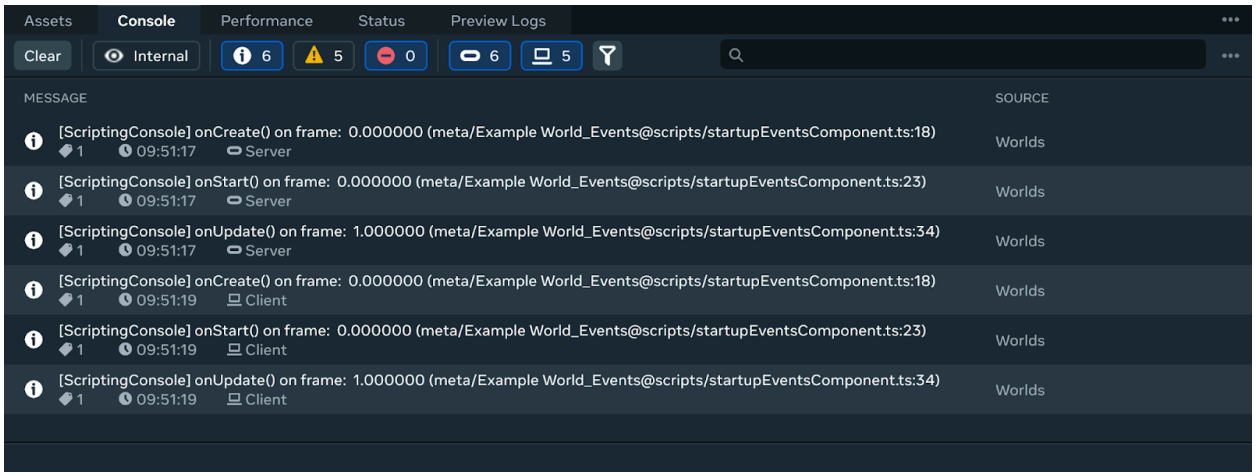
TypeScript

```
import {component, Component, OnEntityStartEvent, subscribe,
OnWorldUpdateEvent, OnWorldUpdateEventPayload, OnEntityDestroyEvent} from
'meta/platform_api@index';

import { OnEntityCreateEvent } from 'meta/platform_api@index';



@component()

export class startupEventsComponent extends Component {

  // Called upon the creation of the Component and before OnUpdateEvent



  // added

  private frameCounter: number = 0;
```

```
private timeCounter: number = 0;

private booOnUpdate: boolean = false;


// added

@subscribe(OnEntityCreateEvent)

onCreate() {

  console.log('onCreate() on frame: ', this.frameCounter);

}


@subscribe(OnEntityStartEvent)

onStart() {

  console.log('onStart() on frame: ', this.frameCounter);

}


// Called once per frame

@subscribe(OnWorldUpdateEvent)

onUpdate(params: OnWorldUpdateEventPayload) {

  // added

  this.frameCounter++;

  this.timeCounter += params.deltaTime;

  if (this.booOnUpdate == false) {

    this.booOnUpdate = true;

    console.log('onUpdate() on frame: ', this.frameCounter);
```

```
    }
  }

  // added
  @subscribe(OnEntityDestroyEvent)
  onDestroy() {
    // following won't appear in logs
    console.log('onDestroy() on frame: ', this.frameCounter);
  }
}
```

When the above script is executed, you should see the following entries in the Console tab:



Notes:

- All of the script's native events execute on the Server client before executing on the visitor's client (player).
- Order of execution is: `onCreate()`, `onStart()` and `onUpdate()` with the latter executing for the first time on Frame 1.
    - The `onDestroy()` event does not appear in the console in this case, since the parent entity is destroyed when the whole preview is brought down.

Owner-NonOwner event execution

Note: Entities that are part of the world template (`space.hstf`) are owned by the server by default. You can write Owner-subscribed functions for entities that are known to be part of the world template.

As needed, when you define an event listener, you can define the client(s) where it is triggered. In the following example, the event listener for `OnWorldUpdateEvent` is configured to execute on the owner of the entity only:

```
TypeScript
 @subcribe(OnWorldUpdateEvent, {execution: Execution.Owner})

 onUpdateEverywhere(params: OnUpdateParams) { ... }

}
```

You can pass into the event handlers the `execution` parameter, which supports the following modes through the `Execution` enum.

Note: Using the `Everywhere` option to a subscription does not increase calls across the network and should not affect performance.

```
JavaScript
/**
 * The possible options for where to execute subscribe
 */
declare enum Execution {
  /**
   * Subscribe will only happen on the client owning the entity
   */
  Owner = 1,
  /**
   * Subscribe will only happen on clients that do not own the entity
   */
  NonOwner = 2,
  /**
   * Subscribes on all clients
   */
  Everywhere = 3,
}
```

Client-Server event execution

Important: Client-server distinctions are not directly supported by the networking model. In the future, there may be multiple servers in a session. While the networking model favors owners and non-owners (proxies), client-server models may be more familiar, and the following solution may work in single-server environments.

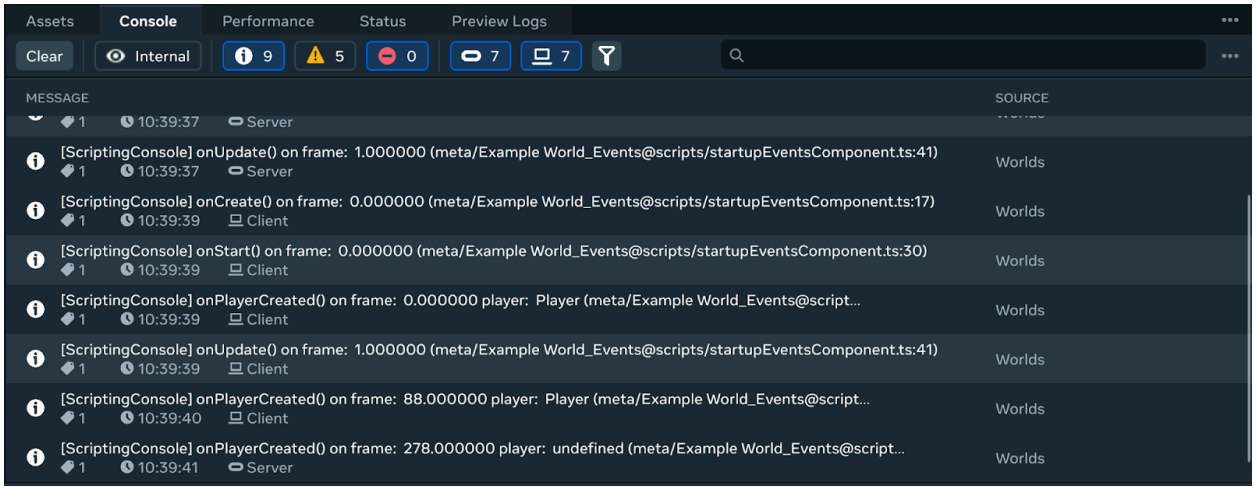
You may have noticed in the previous section that there is no option to execute an event on the server only or on all visitors only. In some cases, this may be relevant.

In HSR, you can manage execution of your events on the client or server through event listeners. To the above script, you can add the following subscriber to the `OnPlayerCreatedEvent()`. Note the added imports.

TypeScript

```
// added

import { OnPlayerCreatedEvent, OnPlayerDestroyedEvent, PlayerService } from
'meta/platform_api@index';

...



// added

@subscribe(OnPlayerCreatedEvent)

onPlayerCreated() {

  const playerService = PlayerService.get();

  const player = playerService.getLocalPlayer();

  console.log('onPlayerCreated() on frame: ', this.frameCounter, 'player: ',
player?.name);

}
```

When the modified script is executed, the following appears in the Console tab:



Notes:

- The `onPlayerCreated()` method executes after the script-based native events have executed at least once.
- The event first executes only the client and returns a valid value for `player.name`.
- The event executes nearly 200 frames later on the server, which returns an `undefined` value for `player.name`.

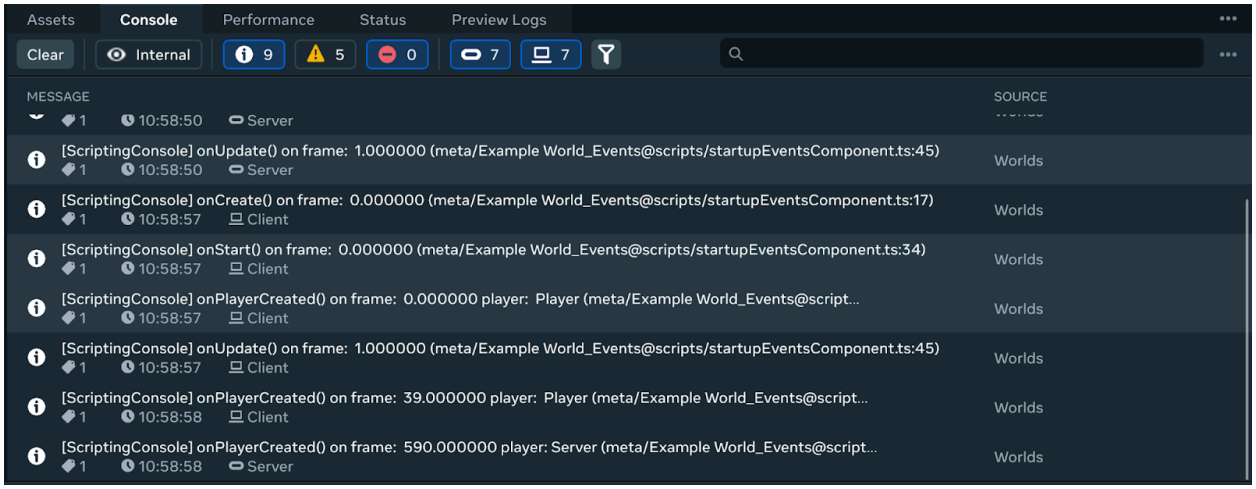
To prevent double-execution of player-related events, you can segment the `OnPlayerCreated()` method based on whether the local player object is `undefined`, and therefore corresponds to the player on the Server client.

TypeScript

```
// added

@subscribe(OnPlayerCreatedEvent)

onPlayerCreated() {

  const playerService = PlayerService.get();

  const player = playerService.getLocalPlayer();

  if (player == undefined) {

    console.log('onPlayerCreated() on frame: ', this.frameCounter, 'player:
Server');

  } else {
```

```
    console.log('onPlayerCreated() on frame: ', this.frameCounter, 'player: ',
player.name);

  }

}
```

Console:



Ideally, the check to see if the local player is the Server is turned into a common function (`isServer()`), which is executed only once per-script execution.

TypeScript

```
  // added

  @subscribe(OnPlayerCreatedEvent, {execution: Execution.Everywhere})

  onPlayerCreated() {

    const playerService = PlayerService.get();

    const player = playerService.getLocalPlayer();

    if (player) {

      if (this.isServerPlayer(player)) {
```

```
        this.subscribe(event1, this.serverEvent1Handler())
        this.subscribe(event2, this.serverEvent2Handler())
      } else {
        console.log('onPlayerCreated() on frame: ', this.frameCounter, 'player:
', player.debugId);
        this.subscribe(event1, this.clientEvent1Handler())
        this.subscribe(event2, this.clientEvent2Handler())
      }
    }
  }
...
  // added
  private isServerPlayer(player: Maybe<Entity>): boolean {
    if (player == undefined) {
      return true;
    } else {
      return false;
    }
  }
private serverEvent1Handler(): void {
  // do server stuff
}
private serverEvent2Handler(): void {
  // do server stuff
```

```
}

private clientEvent1Handler(): void {

  // do client stuff

}

private clientEvent2Handler(): void {

  // do client stuff

}
```

The check to see if the local player is a server is completed only one time per script execution. As a result of that check, the subscribers for server and client can be easily segmented.

Transfer of ownership and subscriptions

At runtime, it's possible to transfer the ownership of an entity, which impacts event execution based on the type of subscription.

Subscription type	TypeScript	When ownership is transferred

Automatic	TypeScript
	`@subscribe(onLittleEvent,{execution: Execution.Everywhere})`
	`littleEvent {`
	`  // do stuff`
	`}`
		These subscriptions depend on the ownership property of the entity for their execution context. When ownership changes, these subscriptions are reset and then resubscribed based on their `Execution` parameter.

Manual		These subscriptions are unaffected and continue to be active subscriptions on the client that set them up.

	TypeScript `this.subscribe(onLittleEvent, this.onLittleEventHandler())`	As needed, the ownership APIs can be used to manually unsubscribe or resubscribe as needed.

Event Examples

This section contains basic examples of events of each type:

- Local
- Network

Native events

Native events are created and generated by the platform at runtime. You do not have to define native events.

To respond to a native event, your script must include the following:

- An import statement
- An event subscriber
- An event handler

Some native events are automatically added to the script template.

OnStartEvent

After all of the entities of the world have been created (see `OnEntityCreateEvent`) below, the `OnStartEvent` is triggered for each script.

- Order of execution is not guaranteed.
- The `OnEntityStartEvent` is included in the default template.

The following example is from the default template. In this case, the `OnEntityStartEvent` subscription is imported, a listener is inserted, and a message (`onStart`) is written to the console.

TypeScript

```
import {component, Component, OnEntityStartEvent, subscribe,
OnWorldUpdateEvent, OnWorldUpdateEventPayload} from 'meta/platform_api@index';



@component()

export class DefaultScriptTemplateComponent extends Component {

  // Called upon the creation of the Component and before OnUpdateEvent

  @subscribe(OnEntityStartEvent)

  onStart() {

    console.log('onStart');

  }
```

OnEntityCreateEvent

The `OnEntityCreateEvent` event is executed after all of the components of an entity have been created.

- You may reference other components in the entity in this method.
- It is not safe to reference the `OnEntityCreateEvent` handlers in other entities of the world. Wait and use the `OnStartEvent` event handler.
- **For HUR creators**: `OnEntityCreateEvent` is similar to the `preStart()` method.

The following example adds an `OnEntityCreateEvent` subscriber to the script. In this case, a local flag `booCreated` is changed as part of the inline event handler.

Note the `// added OnEntityCreateEvent` indicators.

TypeScript

```
import {component, Component, OnEntityStartEvent, subscribe,
OnWorldUpdateEvent, OnWorldUpdateEventPayload} from 'meta/platform_api@index';
```

```
// added OnEntityCreateEvent
import { OnEntityCreateEvent } from 'meta/platform_api@index';

@component()
export class DefaultScriptTemplateComponent extends Component {

  // added OnEntityCreateEvent
  private booCreated: boolean = false;

  // Called upon the creation of the Component and before OnUpdateEvent
  @subscribe(OnEntityStartEvent)
  onStart() {
    console.log('onStart');
  }

  // added OnEntityCreateEvent
  @subscribe(OnEntityCreateEvent)
  onCreate() {
    console.log('onCreate');
    this.booCreated = true;
  }
```

OnDestroyedEvent

You can import and subscribe to the `OnDestroyedEvent` local event, which is triggered when the local entity is destroyed.

Note: It is possible the networked entity has already been deleted by the time this event is emitted, so it is not safe to rely on the entity's state or data at this time.

TypeScript

```
// added OnEntityDestroyEvent

import { OnEntityDestroyEvent, OnEntityDestroyParams } from
'meta/platform_api@index';



@component()

export class DefaultScriptTemplateComponent extends Component {



  // added OnEntityDestroyEvent

  @subscribe(OnEntityDestroyEvent)

  onDestroy() {

    console.log('onDestroy');

  }
```

OnGlobalEntityDestroyEvent

When the local entity is destroyed, this local event is broadcast locally to all entities.

Note: It is possible the networked entity has already been deleted by the time this event is emitted, so it is not safe to rely on the entity's state or data at this time.

In the following example, the `OnGlobalEntityDestroyEvent` and its related `OnGlobalEntityDestroyedParams` parameter object are imported. Since this event is broadcast to all local entities, the entity that has been destroyed must be provided as part of the event message.

TypeScript

```
// added OnGlobalEntityDestroyEvent

import { OnGlobalEntityDestroyEvent, OnGlobalEntityDestroyedParams } from
'meta/platform_api@index';



@component()

export class DefaultScriptTemplateComponent extends Component {



    // added OnGlobalEntityDestroyEvent

  @subscribe(OnGlobalEntityDestroyEvent)

  onDestroyGlobal(params: OnGlobalEntityDestroyedParams) {

    console.log('onDestroyGlobal with params: ' + params.entity);

  }
```

OnWorldUpdateEvent

The `OnWorldUpdateEvent` is emitted from the runtime every frame. Its payload includes a single parameter `deltaTime`, which measures the time in milliseconds that have elapsed since the previous frame.

Note: Avoid executing much code on every frame, which can significantly impact performance. In particular, if your code makes bridge calls, you can see a significant performance degradation. In very few cases do you need to execute code each frame.

- The `OnWorldUpdateEvent` is included in the default script template.

The following example creates two counters based on the `OnWorldUpdateEvent`.

- `frameCount`: counts number of frames
- `secondsCount`: counts number of elapsed seconds

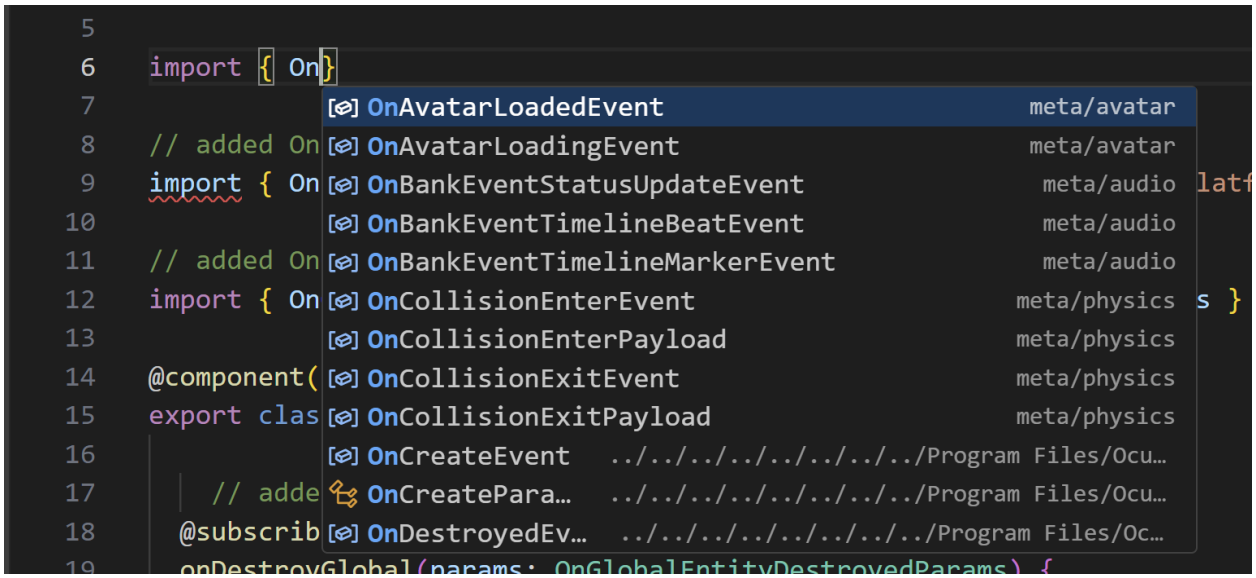
TypeScript

```typescript
@component()
export class DefaultScriptTemplateComponent extends Component {
  private frameCount: number = 0;
  private secondsCount: number = 0;
  private totalSeconds: number = 0;

    // Called once per frame
  @subscribe(OnWorldUpdateEvent)
  onUpdate(params: OnWorldUpdateEventPayload) {

    // frame counter reported every 100 frames
    this.frameCount++;
    if (this.frameCount % 100 == 0) {
      console.log('onUpdate: ' + this.frameCount + ' frames, ');
      if (this.frameCount > 100000) {
        // reset counter
        this.frameCount = 0;
      }
    }

    // seconds counter reported every 10 seconds
    this.totalSeconds += params.deltaTime
    this.secondsCount += params.deltaTime
```

```
    if (this.secondsCount > 10) {
      console.log('onUpdate: ' + this.totalSeconds + ' seconds, ');
      // reset counter
      this.secondsCount= this.secondsCount - 10;
    }
  }
```

Exploring other native and non-native events

If you're using Visual Studio Code, you may be able to use the following hack to explore the available events. In one of your code files, enter: `import { On` and you may see something like the following:



You can add these events and their payload objects to your script. Then, right-click the individual entries in the import statement and select **Definition…**.

You can see API reference documentation for the selected import.

Local events

Local events are not sent over the network. They can only be sent from a client to other entities on the client.

Local events can have a payload message or not. In the following example, `myLocalEvent` is defined to require the `myLocalPayload` event payload, which is a single string/value pair (`{value1: “My String”}`.

- `myLocalEvent` is invoked in multiple ways, using different payloads for each invocation.

The `myEmptyLocalEvent` event contains no payload in its definition.

TypeScript

```
import {component, Component, OnEntityStartEvent, subscribe,
OnWorldUpdateEvent, OnWorldUpdateEventPayload} from 'meta/platform_api@index';


import { property, LocalEvent, OnEntityCreateEvent, EventService} from
'meta/platform_api@index';


const scriptName = "ExampleLocalEventsComponent.ts";


class MyPayload {

  @property()

  value_1: string = '';

}


export const MyLocalEvent = new LocalEvent(
```

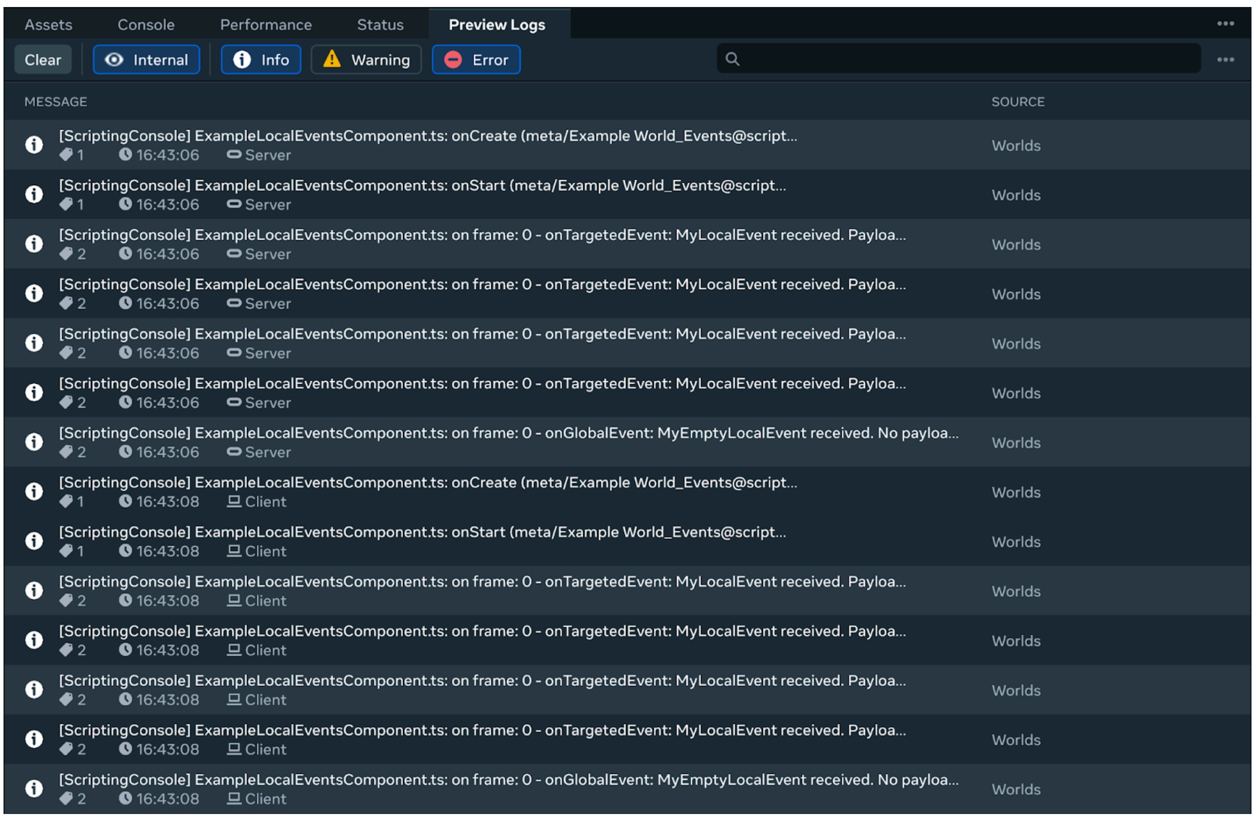
```
  'myLocalEvent',
  MyPayload,
);

// If you do not provide a payload type, the event will default to using an empty payload.
export const MyEmptyLocalEvent = new LocalEvent(
  'myTargetedLocalEvent'
);

@component()
export class ExampleLocalEventsComponent extends Component {

  private frameCount: number = 0;

  @subscribe(OnEntityCreateEvent)
  onCreate() {
    console.log(scriptName + ': onCreate');
    // Manually subscribe.
    this.subscribe(MyEmptyLocalEvent, this.onGlobalEvent);
    this.subscribe(MyLocalEvent, this.onTargetedEvent);

  }
```

```
// Called upon the creation of the Component and before OnUpdateEvent
@subscribe(OnEntityStartEvent)
onStart() {
  console.log(scriptName + ': onStart');

  // Send local event to all entities (globally)
  EventService.sendLocally(MyLocalEvent, {value_1: "myPayload0-inline"});

  // Send local event to all entities (globally)
  let myPayload1 = new MyPayload();
  myPayload1.value_1 = "myPayload1";
  this.sendLocalEvent(MyLocalEvent, myPayload1);

  // Send local event to a specific entity
  let myPayload2 = new MyPayload();
  myPayload2.value_1 = "myPayload2";
  EventService.sendLocally(
        MyLocalEvent,
        myPayload2,
        {eventTarget: this.entity}
      );

  // Send local event to a specific entity
```

```
    let myPayload3 = new MyPayload();

    myPayload3.value_1 = "myPayload3";

    this.sendLocalEvent(MyLocalEvent, myPayload3, this.entity);


    // You can send empty payloads

    EventService.sendLocally(MyEmptyLocalEvent, {});


  }



  @subscribe(MyEmptyLocalEvent)

  onGlobalEvent() {

    console.log(scriptName + ': on frame: ' + this.frameCount + ' -
onGlobalEvent: MyEmptyLocalEvent received. No payload.');

  }


  @subscribe(MyLocalEvent)

  onTargetedEvent(payload: MyPayload) {

        console.log(scriptName + ': on frame: ' + this.frameCount + ' -
onTargetedEvent: MyLocalEvent received. Payload: ' + payload.value_1);

  }


  // Called once per frame

  @subscribe(OnWorldUpdateEvent)
```

```
  onUpdate(params: OnWorldUpdateEventPayload) {
    this.frameCount++;
  }
}
```

In the Console output, the event handlers log their messages in the same order as the events are invoked in the script (as expected):



Network events

Network events are broadcast across the network from one client to one or more clients.

Tip: Use network events for changes to global state or changes to an entity's state that have global implications.

The following example includes several variations on network events, including:

- Network event with payload:
  - Payload specified inline
  - Payload specified as separate object
- Network event with empty payload
- Sending events:
  - To everyone
  - To entity owner
  - To a specified entity on all clients
  - To a specific component on all clients

TypeScript

```
import {component, Component, OnEntityStartEvent, subscribe,
OnWorldUpdateEvent, OnWorldUpdateEventPayload, serializable} from
'meta/platform_api@index';


import {property, EventService, NetworkEvent} from 'meta/platform_api@index';


const scriptName = "ExampleNetworkEventsComponent.ts";


@serializable()

export class MyPayload{

 @property()

 readonly value_1: string = '';

}


@serializable()

export class MyNetworkEventPayload2 {

  public readonly myNum: number;

  constructor(someNum: number = 0) {
```

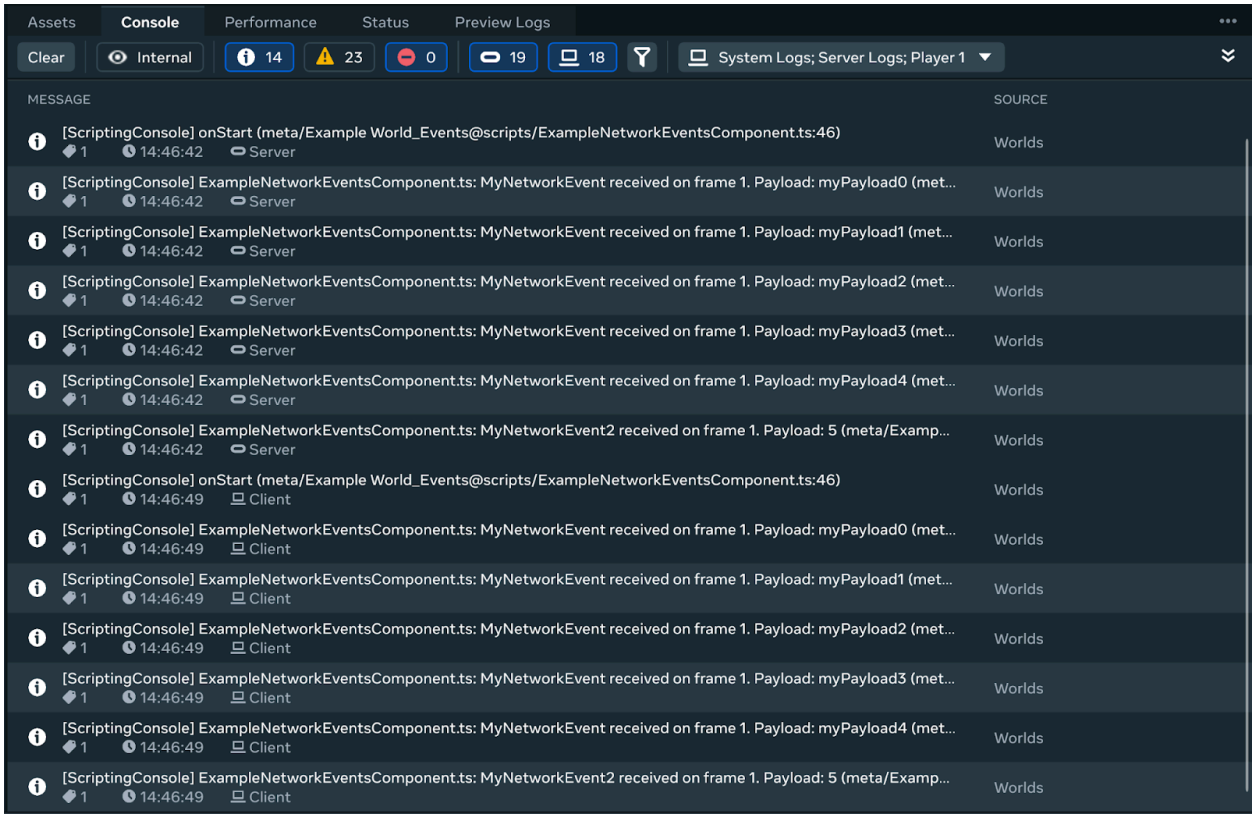
```
    this.myNum = someNum;
  }
}



export const MyNetworkEvent = new NetworkEvent(
 'MyEvent',
 MyPayload,
);

export const MyNetworkEvent2 = new NetworkEvent(
 'MyEvent2',
 MyNetworkEventPayload2,
);



// If you do not provide a payload type, the event will default to using an
empty payload.
export const MyEmptyNetworkEvent = new NetworkEvent(
 'MyEmptyEvent',
);


@component()
export class ExampleNetworkEventsComponent extends Component {
```

```
  private frameCount: number = 0;


  // Called upon the creation of the Component and before OnUpdateEvent

  @subscribe(OnEntityStartEvent)

  onStart() {

    console.log('onStart');


    // Only the owner can send events to non-owning clients

   if (this.entity.isOwned()) {


    // You can broadcast the event to a specific component on all clients

    EventService.sendToEveryone(MyNetworkEvent, {value_1: "myPayload0"}, this);


     // You can broadcast the event to a specific entity on all clients

     EventService.sendToEveryone(MyNetworkEvent, {value_1: "myPayload1"},
this.entity);


   }


   // Non-owning clients can send events to the owning entity

   this.sendEventToOwner(MyNetworkEvent, {value_1: "myPayload3"});
```

```
   // Non-owning clients can send events to specified components on the owning
entity

   EventService.sendToOwner(MyNetworkEvent, {value_1: "myPayload4"}, this);


   // submitting single-value payload as part of payload definition

   const myNewPayload2 = new MyNetworkEventPayload2(5);

   EventService.sendToEveryone(MyNetworkEvent2, myNewPayload2, this.entity);


  }


 @subscribe(MyNetworkEvent)

 handleProxyComponentTarget(msg: MyPayload) {

   console.log(scriptName + ': MyNetworkEvent received on frame ' +
this.frameCount + '. Payload: ' + msg.value_1);

 }


  @subscribe(MyNetworkEvent2)

 handleEmptyNetworkEvent2(msg: MyNetworkEventPayload2) {

   console.log(scriptName + ': MyNetworkEvent2 received on frame ' +
this.frameCount + '. Payload: ' + msg.myNum);

 }


 @subscribe(MyEmptyNetworkEvent)

 handleEmptyNetworkEvent() {
```

```
   console.log(scriptName + ': MyEmptyNetworkEvent received on frame ' +
this.frameCount + '. No payload.');

 }




  // Called once per frame

  @subscribe(OnWorldUpdateEvent)

  onUpdate(params: OnWorldUpdateEventPayload) {

    this.frameCount++;

  }

}
```

Console when previewed:


Assets
Console
Performance
Status
Preview Logs
Clear
Internal
14
23
0
19
18
System Logs; Server Logs; Player 1
MESSAGE
SOURCE
[ScriptingConsole] onStart (meta/Example World_Events@scripts/ExampleNetworkEventsComponent.ts:46)
1 14:46:42 Server
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent received on frame 1. Payload: myPayload0 (met...
1 14:46:42 Server
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent received on frame 1. Payload: myPayload1 (met...
1 14:46:42 Server
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent received on frame 1. Payload: myPayload2 (met...
1 14:46:42 Server
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent received on frame 1. Payload: myPayload3 (met...
1 14:46:42 Server
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent received on frame 1. Payload: myPayload4 (met...
1 14:46:42 Server
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent2 received on frame 1. Payload: 5 (meta/Examp...
1 14:46:42 Server
Worlds
[ScriptingConsole] onStart (meta/Example World_Events@scripts/ExampleNetworkEventsComponent.ts:46)
1 14:46:49 Client
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent received on frame 1. Payload: myPayload0 (met...
1 14:46:49 Client
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent received on frame 1. Payload: myPayload1 (met...
1 14:46:49 Client
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent received on frame 1. Payload: myPayload2 (met...
1 14:46:49 Client
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent received on frame 1. Payload: myPayload3 (met...
1 14:46:49 Client
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent received on frame 1. Payload: myPayload4 (met...
1 14:46:49 Client
Worlds
[ScriptingConsole] ExampleNetworkEventsComponent.ts: MyNetworkEvent2 received on frame 1. Payload: 5 (meta/Examp...
1 14:46:49 Client
Worlds

Methods for sending events

Depending on whether you are sending events to be received at the entity or component level, the following outcomes may occur.

Tip: Specify target entities when sending events whenever you can, which reduces network traffic.

Level	Methods	Outcome	Notes
Component	`sendEventToEveryone`	Sends `NetworkEvent` to corresponding component on every client, including executing client.	Can only be called from the client that owns the entity.
	`sendEventToOwner`	Sends `NetworkEvent` to the owning client of this component, including executing client.	
	`sendLocalEvent`	Sends `LocalEvent` to all listeners on the executing client. Can optionally specify target entities.	
Entity	`sendEventToEveryone`	Sends `NetworkEvent` to all listeners across all components of the target entity.	
	`sendEventToOwner`	Sends `NetworkEvent` to all listeners across all components of the target entity.	

Service (`EventService.`)	`sendGlobally`	Sends a `NetworkEvent` to all listeners on all clients.	The `sendGlobally` method is an indiscriminate broadcast to all clients. Any listener on any client can potentially receive the event, which greatly increases network traffic, leading to performance issues if used incorrectly. Use this method only when necessary.
Service (`EventService.`)	`sendLocally`	Sends a `LocalEvent` to listeners on the local client. By default, the event is sent to all entities. Optionally, you can specify to send to a specific entity.	The `sendLocally` method is useful for sending to a specific or all listeners on the local client.

For networked entities:

Networked entities can communicate with other unrelated networked entities by retrieving a reference to the entity or component that you want to send an event to and call `sendEventToOwner` on it. In the handler for the sent event, call `this.sendEventToEveryone` to send a different event to the entity or component on all other clients.

[META_INTERNAL] DOC NOTES

In the following example, ComponentA is sending and receiving events while ComponentB is just receiving events (and both are attached to different entities).

ComponentA.ts:

JavaScript

```
import {component, Component, OnEntityStartEvent, Execution, type Maybe, type
IEntity} from 'meta/platform_api@index';

import {myNetworkEvent, myLocalEvent} from './ExampleCustomEvents';

import {ComponentB} from './ComponentB';


@component()

export class ComponentA extends Component {


  @property()

  public otherEntity: Maybe<IEntity>; // an entity that has ComponentB


  @subscribe(OnEntityStartEvent, {execution: Execution.Everywhere})

  public onStart() {

    if (this.isOwned()) {

      // send an event to this component instance on all clients

      this.sendEventToEveryone(myNetworkEvent, {});

      // send an event to the owning client of otherEntity, which will
propagate to all
```

```
      // attached components including ComponentB
      otherEntity?.sendEventToOwner(myNetworkEvent, {});
      // you could also get a reference directly to ComponentB and send the
event
      const otherComponent = otherEntity?.getComponent(ComponentB);
      if (otherComponent) {
        otherComponent.sendEventToOwner(myNetworkEvent, {});
      }
    }
    // send a local event directly to otherEntity
    if (otherEntity) {
      this.sendLocalEvent(myLocalEvent, {}, otherEntity);
    }
    // send a local event as a broadcast to all local entities
    this.sendLocalEvent(myLocalEvent, {});
  }


  @subscribe(myNetworkEvent, {execution: Execution.Everywhere})
  private myNetworkEventHandler() {
    console.log("ComponentA received MyNetworkEvent!");
  }
}
```

ComponentB.ts:

```javascript
import {component, Component, Execution} from 'meta/platform_api@index';

import {myNetworkEvent, myLocalEvent} from './ExampleCustomEvents';


@component()

export class ComponentB extends Component {


  @subscribe(myNetworkEvent, {execution: Execution.Everywhere})

  private myNetworkEventHandler() {

    console.log("ComponentB received myNetworkEvent!");

  }


  @subscribe(myLocalEvent, {execution: Execution.Everywhere})

  private myLocalEventHandler() {

    console.log("ComponentB received myLocalEvent!");

  }

}
```

Reference:
https://www.internalfb.com/wiki/Horizon/Onboarding/Scripting/Scripting_Model/Events/#subscribing-to-events

ChrisB Feature Request:
https://fb.workplace.com/groups/hsrstudiofeedback/permalink/522253050909345/

Local events

```
import {LocalEvent} from './events/EventType';
```

Network events