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 tutorial walks through how to set up a simple shooting mechanic in the HSR Horizon Editor. In this tutorial, you create a simple gun, bullet, and target and then build the interactivity between them through TypeScript. After completing these steps, you can extend the tutorial with sound, visual effects, and other elements of a complete game.

- This tutorial extends from the tutorial in the Quick Start Guide. For more information, see Quick Start Guide doc.

Learning Objectives

- Build entities in the Horizon Editor (design by component)
    - Add and configure components to entities
    - Attach scripts to entities

- Organize entities in the Hierarchy panel
- Creating scripts
- Adding dependencies
- Native events
- GetComponent
- Previewing and testing

Differences from HUR:

Many of the implementation differences in these objectives are minor when compared to HUR. Key differences:

- **Design by component:** Allows you to create entities as a set of components, each of which has a configurable set of properties. In this manner, entity composition is much more flexible.
- **Eventing:** This tutorial touches on eventing. In HSR, eventing has some nuanced differences from the HUR implementation.

Entities to Create

- Group of entities that compose the gun
    - Script
- Projectile
    - Script
- Target
    - Script
- Scoreboard
    - Script

Known Issues and Limitations

As of 2025-05-19:

- **Important**: By default, previewing in the Horizon Editor previews a build in mobile-first desktop (Desktop XS mode). The TypeScript APIs do not yet support adding on-screen buttons, which means that you cannot add Aim or Fire buttons to the screen. This is a known issue.
    - **Workaround**: You can preview in Desktop VR, although the game mechanics are a little less fluid. These steps are described in this document.
- The Grab Handle component is oriented backward. This is a known issue.
    - **Workaround**: Rotation the grab handle component's Local Rotation to point the model toward the negative Z axis.

To Begin

The first step is to create your project in your preferred location.

1. Load the Horizon Editor.
2. In the Projects window, click **+Create > Add new project**.
3. Enter a name for your project, such as `Simple Shooting Tutorial`.
4. Under Project location, click **Select folder**. Navigate your local environment to select the parent folder for your project.
5. Click **Create**.
6. A new project folder with your project name is added to the folder, and the starting set of assets and folders are added below it.

For more information on project structures, see Quick Start Guide for HSR doc.

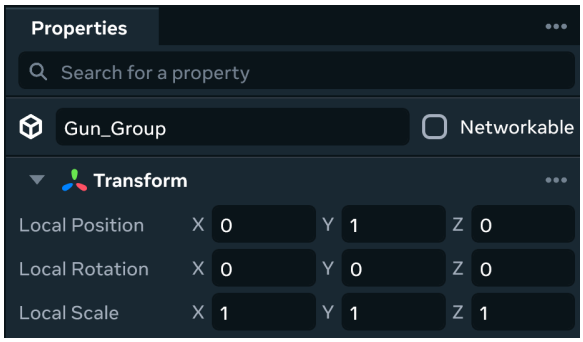
Create Gun

The first step in this tutorial is to create a gun that is:

- Grabbable
- Receives input
- Spawns projectiles

This gun is composed of multiple entities, which are gathered together as a group. Attached to one of the entities is a script that governs the gun's behaviors.

1. At the top of the Hierarchy panel in your project, select the `space.hstf` template tab. This **space template** is the top-level template file for defining the world or space.
2. You should see the `StartingWorld` node and several entities listed beneath it. For more information on these entities, see Quick Start Guide for HSR doc.
3. Right-click `StartingWorld` and select **Add > Entity**.
4. Rename this entity: `Gun_Group`. This entity serves as the parent or root node of the sub-entities that compose the gun entity.
5. By default, the entity is positioned at `(0,0,0)` which is below the surface of the default plane in your world. Reposition the entity to: `(0,1,0)`.



Design by component

At this point, you've created the parent entity for the gun, which has a single component: Transform.

In the next sections, you create sub-entities of the `Gun_Group` entity. The entities have multiple components, each of which provides access to capabilities.

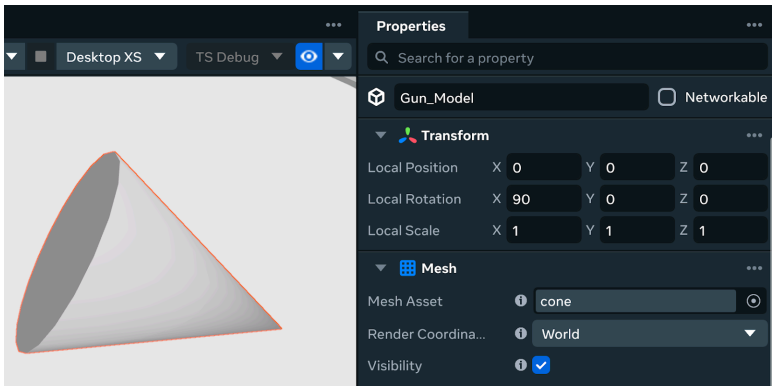
Tip: In effect, each component is a reference to a Component class defined in TypeScript. The TypeScript of the class enables the functionality, which you can configure through the exposed properties in the Properties panel. In this manner, you can assemble complex entities as sets of multiple components and entities, which are built on top of TypeScript scripts. This approach ensures great flexibility in how you build things.

Create gun model

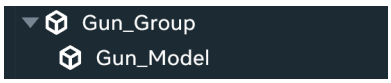
The entity you created is not visible in the world, since it does not yet have any geometry associated with it. The next step is to create the gun model. In this case, you can use a primitive to represent the gun.

1. In the Hierarchy panel, select the `Gun_Group` entity.
2. From the Horizon Editor menu, select **Create > Shapes > Cone**.

3. Rename the entity: `Gun_Model`.
4. Reposition the entity: `(0,1,0)`.
5. Change its rotation: `(90,0,0)`.
    a. **Note**: This rotation points the model toward the negative Z axis. Currently, grab handles result in entities pointing backward, and this rotation is to compensate for that issue. If grab handles are operating correctly, you can skip or fix this rotation.



In the Hierarchy panel, click `Gun_Model` and drop it over `Gun_Group` to make it a sub-entity (**child**) of the main node.



Replace Physics

Click the `Gun_Model` node. You can see that it has a Physics component attached to it.

This Physics component applies to only the model entity; other entities in the `Gun_Group` do not have a Physics component or have their own Physics component, either of which may cause conflicts with the overall `Gun_Group` entity.

The fix is to remove the Physics component from the `Gun_Model` entity and then create one on the `Gun_Group` entity.

Note: Physics components can inherit Colliders components on a child entity. In this case, the physics runs on the parent `Gun_Group` entity, but the child `Gun_Model` entity still hosts its own collision mesh.

1. Select the `Gun_Model` entity. In the Properties panel, click the context menu for the Physics component and select **Remove component**.
2. Select the `Gun_Group` entity.

3. Click **Add a component**.
4. Search for: `Physics`.
5. Add the Physics component.
6. You can leave the default values.

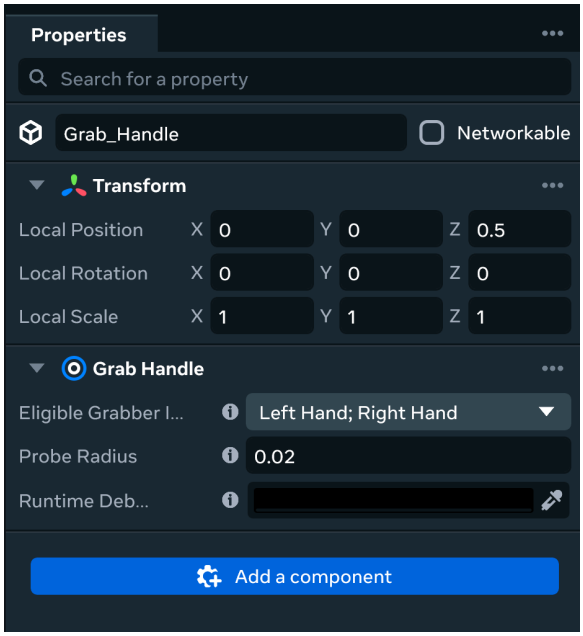
Make gun grabbable

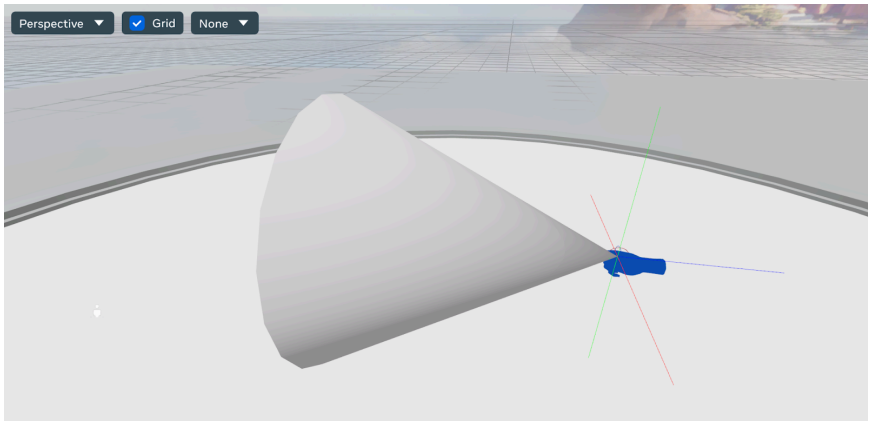
You can't use a handgun unless you can grab it.

1. Select the `Gun_Group` entity.
2. Click **Add a component**.
3. Search for: `Grabbable`.
4. Add the component.

The above step makes the entire group eligible to be grabbed. The next step is to identify where on the model a player's avatar can grab it. In general, this is called a **grab point**. In practice, you add a Grab Handle component.

1. In the Hierarchy panel, right-click the `Gun_Group` entity and select **Add > Entity**.
2. Rename the entity: `Grab_Handle`.
3. In the Properties panel, click **Add a component**.
4. Search for: `Grab Handle`.
5. Add the component.
6. For the Transform values, set its Local Position to: `(0,0,0.5)`.
    a. **Note**: This step positions the entity relative to the positioning of the parent entity.
7. Set the rotation to: `(0,0,0)`.
    a. **Note**: This step rotates the handle to align with the negative Z-axis of the parent entity, as a workaround to the alignment issue for the Grab Handle component. This is a known issue.





Integrate grab handle with other entities

At this point, the grab handle enables the `Gun_Group` entity to be grabbed at the Grab Handle location. However, the grab handle does not move along with the player after it is grabbed, which should be fixed.

1. Click the `Gun_Group` entity.
2. For its Physics Body component, set its Type value to `Dynamic Collider`. This setting allows the collider for the Physics component to move as the `Gun_Group` entity (and all child entities) move.
3. Set its Use Gravity property to `false`. This step prevents the gun from falling on the ground and makes it more accessible.

4. Set Reparent While Held property to `true`, which reparents the entity in the world to the player when it is held.

Checkpoint

At this point the gun should be grabbable by the player. When the player moves, the gun should move with the player.

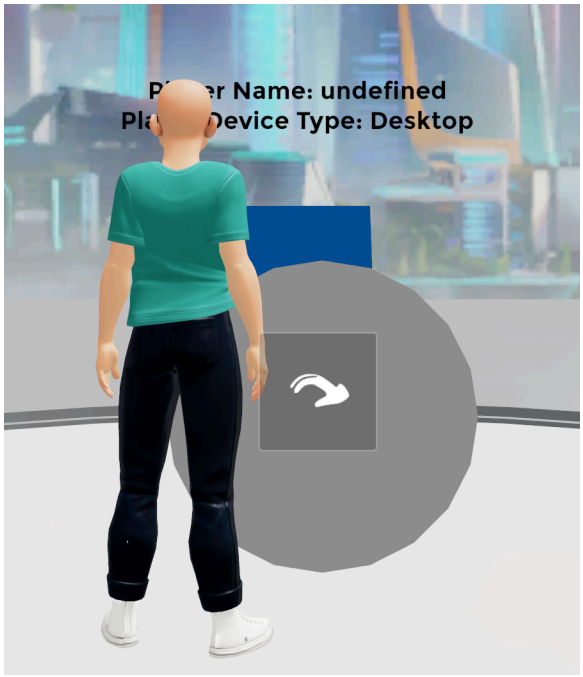
To test:

1. In the Horizon Editor toolbar, Verify that you see `Desktop XS` as the selected preview option. This option means that the rendered preview emulates the desktop & mobile experience. More on this later.
2. Click the **Play button**.

The Preview opens in a new window.

Use `WASD` to move your avatar over to the gun.

Click the **Pickup icon** on the screen:



The gun should appear in your player's hand.

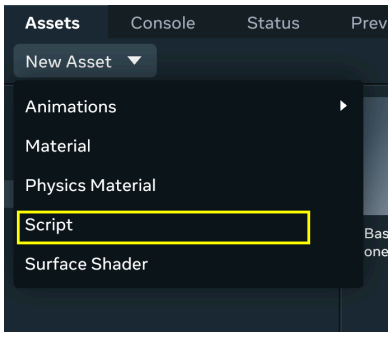
Attach gun script

To make the gun usable by a player, the gun requires scripting for:

- Events that fire when the player grabs and releases the gun
- Capture and action on player input events
- Bullet spawning

In the first pass at this script, you create placeholders for these events, including bullet spawning. When they are triggered, you deliver a console message. Using this simple approach, you can verify your script logic and event functionality before adding in the operative code for these events.

1. To create a new script, in the Horizon Editor, open the **Assets tab**.
2. The Assets tab is opened, displaying your project folder.
3. To store your scripts in a new folder in your project folder:
    a. Right-click the project title node in the Assets tab and select **Open in Explorer**.
    b. In the Explorer window, create a new folder, such as `scripts`.
    c. Return to the Horizon Editor.
4. To create the script, select **Add New > Script** from the Assets tab drop-down.



5. Navigate to your `scripts` folder, if needed.
6. For the script name, enter: `BaseGunComponent`.
    a. The suffix `.ts` is appended to the filename on disk.
    b. **Tip**: It's recommended that you add `Component` to the end of the name.
7. Click **+ Create Script**.
8. The script is saved.
    a. Note: You cannot open a script directly from the Horizon Editor.
9. Right-click your saved script and select **Show in File Explorer**.
10. In the Explorer window, locate the `BaseGunComponent.ts` file. Notice that there is a parallel file: `BaseGunComponent.ts.assetmeta`.
    a. When your script file is created, Horizon Editor passes it through the Asset Processor, which generates this asset metadata file.

b. This **asset metadata** enables the script to be integrated and used in the platform. Do not delete this file. If you do, you must refresh your project by pressing `F5` in the Horizon Editor.

11. Open `BaseGunComponent.ts` in the editor of your choice.
    a. **Tip**: You can double-click a script in the Assets tab to open it in your configured editor. To configure the location of your IDE, select **File > Scripting Settings**.

Build event subscriptions

In the first pass at this script, you create the event subscriptions for a set of platform events. In the local function that is executed for each event, you write a message to the console, which provides a testing platform for your event logic.

Here is the base template for the `BaseGunComponent.ts` file:

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



@component()

export class BaseGunComponent extends Component {

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

  @subscribe(OnEntityStartEvent)

  onStart() {

    console.log('onStart');

  }



  // Called once per frame

  @subscribe(OnWorldUpdateEvent)

  onUpdate(params: OnWorldUpdateEventPayload) {

  }
```

```
}
```

In the above, you have event subscriptions for two platform events:

- `OnEntityStartEvent`, which fires when the script is loaded and triggers the local method: `onStart()`
- `OnWorldUpdateEvent`, which fires every frame during runtime and triggers the local method: `onUpdate()`

You can create similar event subscriptions for other events. Please insert the following code after the `OnStart()` method:

JavaScript

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


@subscribe(OnPlayerGrabEvent)
onGrab(params: PlayerGrabEventPayload) {
  console.log('onGrab');
}


@subscribe(OnPlayerReleaseEvent)
onRelease(params: PlayerGrabEventPayload) {
  console.log('onRelease');
}
```

```
  @subscribe(OnPlayerInputEvent)
  onInput(params: PlayerInputEventPayload) {
    console.log('onInput');
  }
```

You may see error messages related to these events. They need to be added among your imports:

JavaScript

```
import {component,
  Component,
  OnEntityStartEvent,
  subscribe,
  OnWorldUpdateEvent,
  OnWorldUpdateEventPayload,
  OnPlayerInputEvent,
  PlayerInputEventPayload,
  property,
  OnEntityCreateEvent,
  } from 'meta/platform_api@index';
import { OnPlayerGrabEvent,
  OnPlayerReleaseEvent,
  PlayerGrabEventPayload
 } from 'meta/player_grabbing@index';
```

Note that the event subscriptions include:

- Event itself
- **Note**: Events to which you subscribe (`@subscribe`) must also include a Payload object.

Tip: In some IDEs, when you enter references to objects that are new to your script, the corresponding imports may be automatically added for you.

Bullet spawning event placeholder

When a bullet is fired, the script needs to respawn to the firing mechanism and to spawn a bullet with a vector applied to it. Later, you create a template for the bullet asset. For now, you must add a script property to the template that you later build and populate.

Bullet template as a property reference:

For the event to spawn a bullet, you add the following `@property()` declaration and the reference to the property itself:

```
JavaScript
export class BaseGunComponent extends Component {

  @property()

  private bulletTemplate: Maybe<TemplateAsset> = null;
```

The declaration for the `bulletTemplate` property contains the `Maybe<Template>` assignment, which is required for non-primitive properties. Without this reference, the TypeScript compiler may fail to serialize the reference and fail to compile the script.

Note: When declaring non-primitive properties and variables, always wrap the declaration in the `Maybe<T>` construction. This construction prevents the property from being marked as undefined at runtime, which causes an error.

The `Maybe` and `Template` components must be added to your imports. Note that `Maybe` must be imported as a type:

```
JavaScript
import { TemplateAsset } from 'meta/platform_api@index';

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

SpawnBullet method:

You can now add the spawn bullet event method:

```
JavaScript

  private spawnBullet(template: TemplateAsset, gunTransform:
TransformComponent) {

    console.log('spawnBullet');

  };
```

Later, this method is called from the `OnPlayerInputEvent` method.

Checkpoint:

Your script should look like the following:

```
JavaScript

import {component,

  Component,

  OnEntityStartEvent,

  subscribe,

  OnWorldUpdateEvent,

  OnWorldUpdateEventPayload,

  OnPlayerInputEvent,

  PlayerInputEventPayload,

  TemplateAsset,

  TransformComponent,

  property,

  OnEntityCreateEvent,
```

```
  } from 'meta/platform_api@index';
import { OnPlayerGrabEvent,
  OnPlayerReleaseEvent,
  PlayerGrabEventPayload
 } from 'meta/player_grabbing@index';
import type { Maybe, IEntity } from 'meta/platform_api@index'


@component()
export class BaseGunComponent extends Component {
  @property()
  private bulletTemplate: Maybe<TemplateAsset> = null;


  @subscribe(OnEntityCreateEvent)
  onCreate() {
    console.log('onCreate');
  }


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

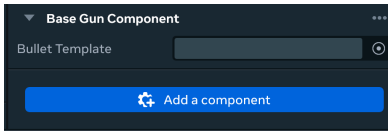
```
  @subscribe(OnPlayerGrabEvent)
  onGrab(params: PlayerGrabEventPayload) {
    console.log('onGrab');
  }

  @subscribe(OnPlayerReleaseEvent)
  onRelease(params: PlayerGrabEventPayload) {
    console.log('onRelease');
  }

  @subscribe(OnPlayerInputEvent)
  onInput(params: PlayerInputEventPayload) {
    console.log('onInput');
  }

  private spawnBullet(template: TemplateAsset, gunTransform:
TransformComponent) {
    console.log('spawnBullet');
  };

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

```
};
```

Attach script:

You can now attach the script to your `Gun_Group` entity.

1. Select the `Gun_Group` entity.
2. In the Properties panel, click **Add a component**.
3. Search for `Base Gun`.
4. Add the component.



Save your project file. Your project assets should also be refreshed.

Tip: You can manually refresh by pressing `F5` at any time.

Checkpoint

You can now check the logic of your `BaseGunComponent.ts`. When you preview your world, the following actions should generate log messages in the Console:

- Startup: `OnStart()`
- Player grabs gun: `OnPlayerGrabEvent()`
- Player drops gun: `OnPlayerReleaseEvent()`
    - **Note**: In Desktop XS preview, press the on-screen icon to drop.
- Any player movement: `OnPlayerInputEvent()`
- The spawn bullet event is not wired up yet.

Create Bullet Template

Since your weapon fires bullets, you must provide instances of the bullet, when the trigger button is pressed. The mechanism for developing and deploying these instances of a bullet is a template.

A **template** defines an entity, package, dependency, or even a space for use in the HSR platform. In your project, any `.hstf` file is a template.

- The `space.hstf` file is called a **space template**, as it contains some extra metadata to define the top-level of your world.

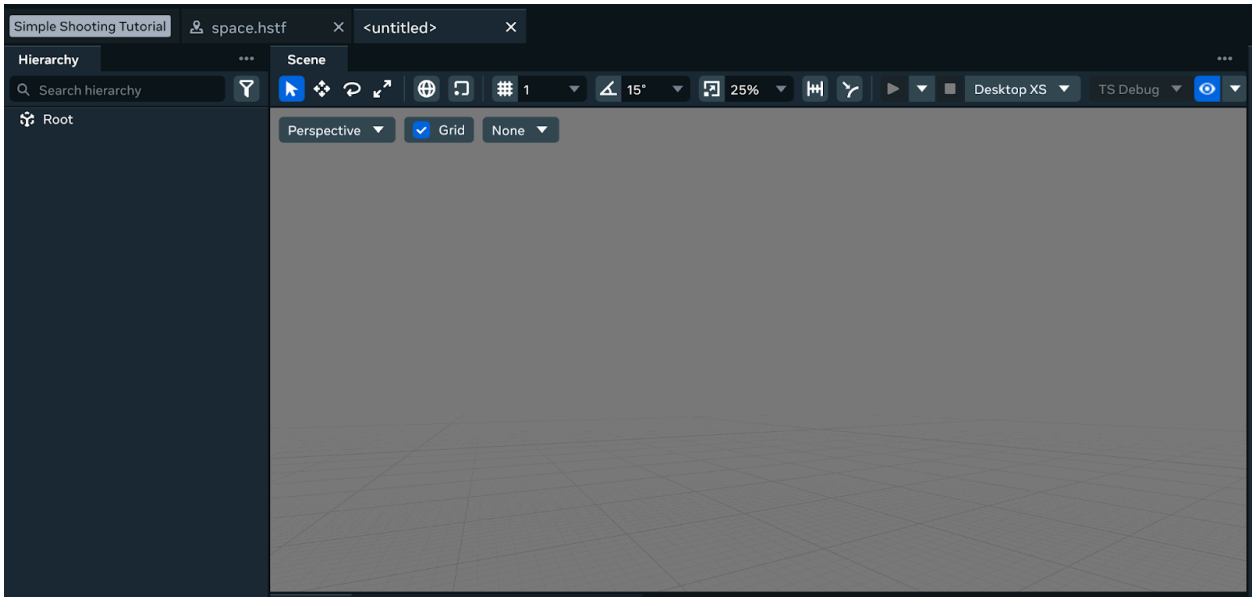
You can create additional templates for reusable assets. Each template may contain one or more entities, which allows you to iterate quickly over these entities and to deploy updates to all instances through a single source.

Create folder

Through Windows Explorer, you may wish to create a folder in your project's folder for these reusable components, such as `templates` or `prefabs`.

Create template file

1. Return to the Horizon Editor.
2. Press `F5` to refresh your project assets. The Horizon Editor is now aware of the new folder you created.
3. Select **File menu > New Template**.
4. Save the template to the new folder in your project. Name the template file: `bullet.hstf`.



Your template file includes:

- A `Root` node, under which you create the entities of the template.
- A gray-world background.

Add bullet primitive

To your new template, you add a sphere, which represents the bullet model.

1. From the menu system, select **Create menu > Shapes > Sphere**.
2. A sphere primitive is added to the template. It contains multiple components, including components for transformation, mesh, collision, and physics.
3. Rename primitive entity to: `Bullet_Model`.
4. To shrink it closer to bullet size, set Local Scale to: `(0.2, 0.2, 0.2)`.
5. Since the model is part of the larger group of assets under `Root`, you must manage positioning through the `Root` node.
    a. For the `Root` node, click **Add a component**.
    b. Search for: `Transform` and select it.
    c. **Note**: Instantiating a template **includes** the `Root` node. When the template is spawned, the `Root` node is used for tracking transformation information. For templates that are being spawned, the `Root` node must include a Transform component.
6. For the `Root` entity, please configure the following:
    a. Add a Collider Sphere component.

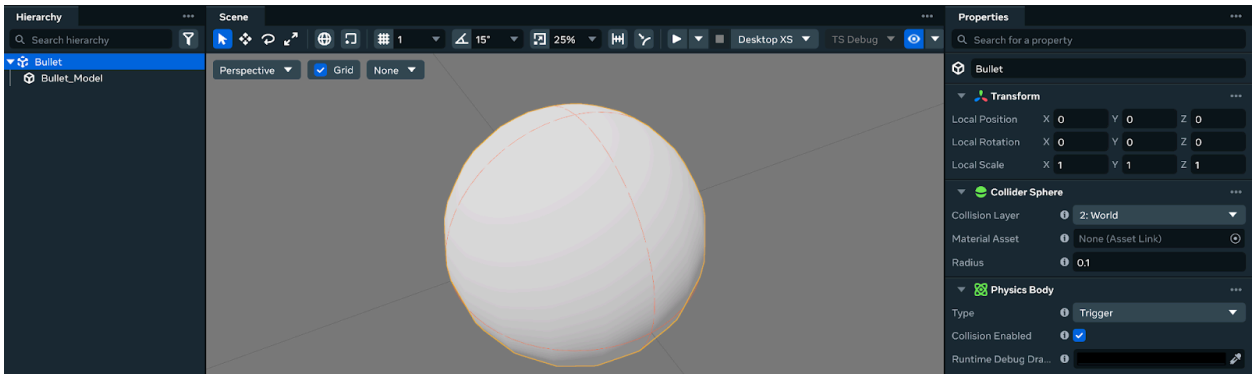
i. For the Collider Sphere component, set Radius to `0.1`.

b. Add a Physics Body component.

i. For the Physics Body component:

1. Set Type to `Trigger`.
2. Set Use Gravity to `false`.

7. Rename the `Root` node to `Bullet`. When the template is instantiated, `Bullet` is the name of the instance.
8. Save `bullet.hstf`.



Attach bullet script

You now create a script to power the bullet.

1. Open the **Assets tab**.
2. In your project folder, open the folder where you create scripts.
3. Select **Add New > Script**.
4. Name this script: `BulletComponent`.
5. Double-click the script in your external editor.

onUpdate() event:

Your script must track the `onUpdate()` world event, which is already included in the script file as a placeholder. For now, you can add a console message. Note that this console message executes every frame, so it should only remain during development:

```javascript
// Called once per frame

@subscribe(OnWorldUpdateEvent)
```

```
onUpdate(params: OnWorldUpdateEventPayload) {
  console.log("onUpdate");
}
```

This method is developed later.

moveSpeed property:

Add to your script a property called, `moveSpeed`. This property captures the speed at which the bullet moves.

Script properties: As a script property, `moveSpeed` is made externally available through the Properties panel, allowing non-engineers to change its value. In this manner, designers can adjust gameplay settings without having to touch code.

Add `moveSpeed` just below the class declaration:

```
JavaScript
export class BulletComponent extends Component {
  @property()
  private moveSpeed: number = 1;
```

Transform component variable:

Add a local variable to store a reference to the bullet entity's Transform component.

Later, this reference is used to programmatically update the properties inside the Transform component.

```
JavaScript
@component()
export class BulletComponent extends Component {
  @property()
```

```
  private moveSpeed: number = 1;


  private transform: Maybe<TransformComponent> = null;
```

The `transform` reference is of type `TransformComponent`. As a non-primitive type, it must be bracketed by a `Maybe<T>` qualifier, which prevents runtime errors for possible undefined values.

In the `OnCreate()` method, assign this reference to the local Transform component.

```
JavaScript
@subscribe(OnEntityCreateEvent)

onCreate() {

  console.log('onCreate');

  this.transform = this.entity.getComponent(TransformComponent);

}
```

When the Bullet template is instantiated, its transform component is cached to the `transform` variable. Later, `OnUpdate()` is modified to reposition the bullet based on changes in time and the `moveSpeed` property value.

Move the bullet

Create an empty function called `move`, which takes as inputs:

- The `transform` variable
- `moveSpeed` variable
- `deltaTime` variable (more on this later)

JavaScript

```
private move(transform: TransformComponent, deltaTime: number, moveSpeed:
number) {

  console.log('move');

};
```

The `move` function is called from `OnUpdate()`, as long as the value for `transform` is not null:

JavaScript

```
// Called once per frame

@subscribe(OnWorldUpdateEvent);

onUpdate(payload: OnWorldUpdateEventPayload) {

  // console.log("OnUpdate");

  if (!this.transform) { return } else {

    this.move(this.transform, payload.deltaTime, this.moveSpeed);

  };

};
```

Since it's possible for `GetComponent(TransformComponent)` to return `null`, you must check for a null value before you pass it as a parameter to the `move` function. Within the `move` function, you do not need to check for a valid value.

- Note that the console log for this item is now commented out, as the `move` function registers a console log message, which executes each frame.

Add code to the `move` function to change the position of the `this.transform` component, which causes movement of the entity:

```
JavaScript

private move(transform: TransformComponent, deltaTime: number, moveSpeed:
number) {

  console.log('move');

  transform.worldPosition =
transform.worldPosition.add(transform.worldForward.mul(moveSpeed * deltaTime)

  );

};
```

The `transform.worldPosition` adds to `worldForward` position the `moveSpeed` value multiplied by `deltaTime`, which represents the amount of time between the current and most previous frame.

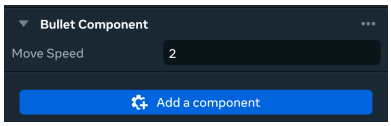
Note: Do not use `localPosition` information for this transformation.

Tip: The `deltaTime` value that is part of the `OnUpdate()` payload is maintained by the system and can be used for tracking time and therefore movement in the world at runtime.

Attach script to Bullet node

1. In the Hierarchy panel, select the `Root` node.
2. In the Properties panel, click **Add a component**.
3. Search for `BulletComponent`.
4. Add the script.

You should see the `Move Speed` (display name) parameter exposed in the Properties panel.



Checkpoint: Select `bullet.hstf` tab. Preview this template. You should see the bullet drifting across the screen.

Add bullet reference

To make the gun functional, the remaining tasks are:

- Create a reference to the bullet by the gun
- Create code to spawn the bullet
- Spawn a bullet when the gun is grabbed and the trigger is pulled

To create a reference:

1. In the Hierarchy panel, select the `Gun_Group` node.
2. In the Assets tab, open the folder containing your `bullet.hstf` template.
3. Drag and drop the template from the Assets tab to the `Bullet Template` property under the `Base Gun Component` script component. See below.

The `BaseGunComponent` script now has a reference to the bullet template to spawn.

Spawning

In the Properties panel, open the context menu for the `Base Gun Component` and select **Open script**.

In the script, add a new variable for storing the Transform component of the base gun component. In the `OnCreate()` method, populate this variable, which looks like the following:

```
JavaScript
private transform: Maybe<TransformComponent> = null;


@subscribe(OnEntityCreateEvent)

onCreate() {
```

```
  console.log('onCreate');

  this.transform = this.entity.getComponent(TransformComponent);

};
```

To access the entity's Transform component, you must create an explicit reference to it.

spawnBullet function

You can now update the spawnBullet function, which creates an instance of the bullet template:

```
JavaScript
private spawnBullet(template: TemplateAsset, gunTransform: TransformComponent)
{

  console.log('spawnBullet');

  WorldService.get().spawnTemplate({templateAsset: template, position:
gunTransform.worldPosition, rotation: gunTransform.worldRotation, networkMode:
NetworkMode.LocalOnly})

  .then((entity: IEntity) => {

    console.log('spawnBullet success');

  })

  .catch((e) => {

    console.error("spawnBullet failed to spawn: " + e)

  })

 };
```

Notes:

- The `spawnTemplate` method takes inputs of:
  - Bullet template
  - Gun's position

    - Gun's rotation
    - Network mode of `LocalOnly`, which means that the bullet is spawned only on the local client. For this simple demonstration, it does not need to be shared across the network.
- The `then/catch` structures are used to capture success/failure outcomes from the `spawnTemplate` method.
    - **Note**: These are important structures to use for asynchronous requests to the server, which allow you to capture success and failure and to respond accordingly. These should be used for all async operations.

Firing the gun

The gun can only be fired when a player is holding it. Create a Boolean variable to store whether the gun is held or not as part of the class declaration:

```
JavaScript
@component()

export class BaseGunComponent extends Component {

  @property()

  private bulletTemplate: Maybe<TemplateAsset> = null;



  private transform: Maybe<TransformComponent> = null;

  private isHeld: Boolean = false;
```

In the `OnGrab()` and `OnRelease()` methods, add statements to assign values to `isHeld`:

```
JavaScript
@subscribe(OnPlayerGrabEvent)

onGrab(params: PlayerGrabEventPayload) {

  console.log('onGrab');

  this.isHeld = true;

};
```

```
@subscribe(OnPlayerReleaseEvent)

onRelease(params: PlayerGrabEventPayload) {

  console.log('onRelease');

  this.isHeld = false;

};
```

Now that you can determine if the gun is held, you can determine if it's time to fire it. Firing requires that:

- The gun is held
- BulletTemplate component exists
- Transform component exists
- The trigger has been pushed

If the above checks are `true`, then fire the bullet. This code is added to the `OnInput()` method:

JavaScript

```
@subscribe(OnPlayerInputEvent)

onInput(params: PlayerInputEventPayload) {

  console.log('onInput');

  if (!this.bulletTemplate) {

    console.error("onInput failed to spawn: no bullet template")

    return;

  }

  if (!this.transform) {

    console.error("Missing transform component.")

    return;
```

```
  };


//    if (this.isHeld && params.rTriggerButton) {
if (this.isHeld && params.xButton) {
  this.spawnBullet(this.bulletTemplate, this.transform);
  };
};
```

Since the properties can be null, it's important to check as early as possible to see if they are null and to return from the method if so.

Note: The `xButton` parameter is the key binding for the X button on the VR controller. The corresponding keyboard mapping is `T`.

Full script:

Your full `BaseGunComponent.ts` script should look like the following:

JavaScript

```
import {component,
  Component,
  OnEntityStartEvent,
  subscribe,
  OnWorldUpdateEvent,
  OnWorldUpdateEventPayload,
  OnPlayerInputEvent,
  PlayerInputEventPayload,
```

```
  TemplateAsset,
  TransformComponent,
  property,
  OnEntityCreateEvent,
  WorldService,
  NetworkMode,
  } from 'meta/platform_api@index';
import { OnPlayerGrabEvent,
  OnPlayerReleaseEvent,
  PlayerGrabEventPayload
 } from 'meta/player_grabbing@index';
import type { Maybe, IEntity } from 'meta/platform_api@index'
import { SoundComponent } from 'meta/audio@index';

@component()
export class BaseGunComponent extends Component {
  @property()
  private bulletTemplate: Maybe<TemplateAsset> = null;

  private transform: Maybe<TransformComponent> = null;
  private isHeld: Boolean = false;

  private sound: Maybe<SoundComponent> = null;
```

```
@subscribe(OnEntityCreateEvent)
onCreate() {
  console.log('onCreate');
  this.transform = this.entity.getComponent(TransformComponent);
}

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

@subscribe(OnPlayerGrabEvent)
onGrab(params: PlayerGrabEventPayload) {
  console.log('onGrab');
  this.isHeld = true;
}

@subscribe(OnPlayerReleaseEvent)
onRelease(params: PlayerGrabEventPayload) {
  console.log('onRelease');
  this.isHeld = false;
```

```
  }


  @subscribe(OnPlayerInputEvent)
  onInput(params: PlayerInputEventPayload) {
    console.log('onInput');
    if (!this.bulletTemplate) {
      console.error("onInput failed to spawn: no bullet template")
      return;
    }
    if (!this.transform) {
      console.error("Missing transform component.")
      return;
    }


//    if (this.isHeld && params.rTriggerButton) {
  if (this.isHeld && params.xButton) {
    this.spawnBullet(this.bulletTemplate, this.transform);
    }
  }


  private spawnBullet(template: TemplateAsset, gunTransform:
TransformComponent) {
    console.log('spawnBullet');
```

```
    WorldService.get().spawnTemplate({templateAsset: template, position:
gunTransform.worldPosition, rotation: gunTransform.worldRotation, networkMode:
NetworkMode.LocalOnly})
    .then((entity: IEntity) => {
      this.sound = entity.getComponent(SoundComponent);
      if (this.sound) {
        this.sound.play();
      }
      console.log('spawnBullet success');
    })
    .catch((e) => {
      console.error("spawnBullet failed to spawn: " + e)
    })

  }

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

Note that the `OnWorldUpdateEvent` has been commented out. No need to run code every frame if it is unused.

Checkpoint

Note: Since you cannot add a button for the firing function to the default Desktop XS previewing method, you must choose one of the other methods in which to preview this world.

Preview in Desktop VR:

- Use `WASD` to move your avatar. You can use the mouse or `Q` and `R` to rotate.
- Move to the pointed end of the cone. When you are near the grab handle, press and hold `2`. The gun is grabbed.
- To fire the gun, press `0`.

Preview in headset:

Use the Meta Quest Developer Hub application to connect your headset to your local desktop. Then, you can preview your local project in your headset. For more information, see Preview Local HSR Project doc.

When you preview the world in Desktop VR or Quest Link, you should be able to pick up the gun and fire a bullet.

Create Target

Now that you have a gun that fires, you need something to hit. This section walks through the process of creating a target that takes damage from impacts from bullets.

Note: This example uses custom events to demonstrate messaging from one component to another. In this case, since the target of the message is a single known component, you should use the `GetComponent()` method instead.

Event entities:

The following are created as part of this exercise:

- TargetComponent
- Define the parameter class
- Create the event
- Subscribe to the event
- Send the event

Create target object

To create the target, you can use a Plane primitive.

1. In the menu, select **Create > Shapes > Plane**.
2. Rename this entity: `Target`.
3. Add a Collider component.
    a. Verify that collisions are enabled.
4. Add a Physics Body component.
    a. For the Physics Body, set its Type value to `Dynamic`.
    b. Note: The bullet entity notifies itself of collisions using a trigger. Static objects do not trigger triggers, so this Physics Body must be set to `Dynamic`, even though it doesn't move.
    c. Set its Use Gravity property to `false`.
5. You can experiment by applying a texture.
    a. Create a flat image 512 x 512.
    b. Fill it with the color of your preference.
    c. Save this file into a `textures` directory in your project folder.
    d. In the Editor, press `F5` to refresh.
    e. For your Target entity, under its Mesh component, click Light Map Texture. Select the texture file that you created.

Create event and parameter

Now that you have created the target entity, you can create the event that instructs the target to take damage. An **event** is a message emitted from one entity and delivered to one or more entities or components that are subscribing to the event. An event is composed of the following elements:

- The event definition
- The payload that the event delivers to the subscriber

Create event file

Since this event can be used in theory by any other script in the world, it should be created in a separate file, which can be imported into any script that wishes to use it.

1. Through the Assets tab, create a new script.
2. Navigate to the scripts folder inside your project folder.
3. Name the script: `Events`.
4. Double-click the script to open it in your preferred text editor.
5. Replace its contents with the following:

```JavaScript
import { serializable } from 'meta/platform_api@index';

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


@serializable()

export class DamageParams {

  readonly value: number = 0;

};


export const COMBAT_EVENTS = {

  takeDamageLocal: new LocalEvent("TakeDamageLocal", DamageParams),

};
```

The above contains three declarations:

- `COMBAT_EVENTS` as a set of events.
- In this case, it holds a single event: `takeDamageLocal`, which is used to apply damage to the event subscriber.
- `DamageParams` defines the payload that is delivered to the event subscriber with the event.

Recommended practice: It's possible to submit the event payload as a generic as part of the event definition. However, for network events, this type of construction causes the event to fail silently. You should use the above construction in which the event payload is submitted as the second parameter.

Parameter class

The parameter class defines the message that is sent with the event. When you define a `LocalEvent`, the `<T>` must resolve to the default constructor for messages for the event.

Event parameters are defined as classes with specific requirements:

- The class must be serializable, which means that `serializable` must be imported into the file and applied to the class definition.
- All fields that are part of the class must be marked as `readonly`.
    - **Note**: This read-only attribute does not apply to the messages that you send. When you create the message, you can define a new instance of the parameter class, assigning values to these attributes that are marked `readonly` in the parameter class definition.

In the above definition, you can see that the event is defined as a `LocalEvent` using `DamageParams` as its parameter class. This class delivers a single piece of data: a parameter called `value`.

Create Target script

Now, you build the mechanisms for receiving and processing the event.

1. Create a new script in your script folder.
2. Call this script: `TargetComponent`.
3. Add this script to your Target entity.
4. Open the script in your external editor.

Replace the contents of the script with the following:

JavaScript

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



@component()

export class TargetComponent extends Component {

  @property()

  private maxHealth: number = 0;

  private currentHealth: number = 0;



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

```
  @subscribe(OnEntityStartEvent)
  onStart() {
    console.log('onStart: Target Component');
    this.resetHealth()
  };

  private resetHealth() {
    console.log("Resetting health.");
    this.currentHealth = this.maxHealth;
  };

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

Create event subscription

Save the file. To the above file, you must add the event subscription for the event that you previously created. An **event subscription** is a listener for an event message. When the event is received, the listener's code is executed and can reference the payload message of the event.

You create an event subscription for a custom event like you do for a native event. In this case, you import the event from the `Events` file into your `TargetComponent` script. You also need to import the `DamageParams` type separately from the same file.

```
JavaScript

import { COMBAT_EVENTS } from "./Events";

import type { DamageParams } from "./Events";
```

Note that the path at the end of the above is relative to the current directory for the `TargetComponent` script.

Your `TargetComponent` script can now be set up with a subscriber to this event. Please add the following code:

```
JavaScript
@subscribe(COMBAT_EVENTS.takeDamageLocal)

private takeDamage(damage: DamageParams) {

  // prevent negative damage (i.e. healing)

  if (damage.value <= 0) {

    console.log("Attempting to deal negative damage.")

    return;

  };


  console.log("Taking damage: " + damage.value.toString())

  // prevent taking more damage than we have health

  this.currentHealth -= Math.min(this.currentHealth, damage.value);


};
```

Notes:

- The `@subscribe` statement identifies the event to which to subscribe.
- The `takeDamage` function takes an input called `damage` of `DamageParams`, which matches the type of the event payload.

- Note that the values of the event payload are referenced in the function as: `damage.value`, as `value` is the specific property in the payload containing the amount of damage.

Send event

The `takeDamageLocal` event must be set from the `BulletComponent` script when a collision is detected. Open `BulletComponent.ts`.

Since its Physics Body Type of the entity is set to `Trigger`, you must import the trigger event definitions. Add the following imports:

```
JavaScript
import { OnTriggerEnterEvent, OnTriggerEnterPayload } from
'meta/physics@index';

import { COMBAT_EVENTS } from './Events';
```

When the trigger of the Physics Body is entered, the event must be sent, before the bullet entity is destroyed:

```
JavaScript
@subscribe(OnTriggerEnterEvent)

onTriggerEnter(payload: OnTriggerEnterPayload) {

  console.log("Bullet collision!")

  this.sendLocalEvent(COMBAT_EVENTS.takeDamageLocal, {value: 5},
payload.actorEntity)

  this.entity.destroy();

};
```

Checkpoint

TBD Work in progress.

When you preview your world now, you should be able to:

- Pick up the gun
- Aim and fire the gun
- Make a bullet impact the target
- See `Bullet collision!` in the console.

GetComponent

The event system that you previously created is a reasonable approach to transmitting messages between components. Eventing works well when the targets are unknown and/or are multiple.

However, your event is a message delivered through the runtime and potentially across clients, which can introduce latency issues, collisions, and potentially other issues.

In this case, the target of your event message is a specific component. Since the component is known, you can:

- Create a reference to the entity
- Create a reference to the entity's component
- Modify the properties of the component

Replace the code in the `OnTriggerEnterEvent` with the following:

JavaScript

```
@subscribe(OnTriggerEnterEvent)

onTriggerEnter(payload: OnTriggerEnterPayload) {

  console.log("Bullet collision!")

  // this.sendLocalEvent(COMBAT_EVENTS.takeDamageLocal, {value: 5},
payload.actorEntity)

  const targetComponent = payload.actorEntity?.getComponent(TargetComponent);

  if (targetComponent) {

    targetComponent.onTakeDamage({value: this.damage})

  };
```

```
    this.entity.destroy();
  };
```

You may need to add `TargetComponent` as an import.

Conclusion

You now have a working shooting mechanic. You have built:

- A working gun, including a script to spawn a bullet based on a trigger press.
- A bullet that responds to collisions and features a configurable speed.
- A target that can take damage based on bullet collisions.
- A simple event to manage updating a local health property based on an event generated from a bullet collision.

You have learned how to:

- Build simple entities using primitives
- Design entities using components
- Reference the components in the current entity and in other entities in the world
- Subscribe to native events
- Define, send, and subscribe to custom events
- Send messages to the console
- Preview your world in the Horizon Editor

Well done!