RDT

Runtime Debug Toolkit User Guide

Inspect runtime objects, edit supported values, and execute debug actions from a focused in-game interface.

Version 1.0

Requires Unity 6.0 or newer. Currently validated with Unity 6000.5.4f1 and URP.

Last updated: July 2026

Contents

1. Overview

Runtime Debug Toolkit is an in-game debugging interface for Unity. It gives developers and testers a direct view of the current scene, selected Components, editable runtime values, and project-defined debug actions without building a separate debug screen for every system.

What you can do

- Browse GameObjects and Components while the game is running.
- Inspect and edit supported public fields and properties.
- Explore nested data without displaying an entire object graph at once.
- Expose safe project commands as searchable debug actions.
- Use the included UI as-is or extend it with project-specific types and templates.

Designed for development workflows

The toolkit is compact, draggable, and hidden until requested. It can be restricted to development builds, which is the recommended configuration for production projects.

No input package is mandatory. Projects can open the toolkit through their own code, the Legacy Input Manager integration, or the optional Input System integration. Setup is covered in section 2 and activation choices are explained in section 4.

Render pipeline compatibility

The runtime toolkit and UI are render-pipeline independent. The included demo scene is configured for the Universal Render Pipeline (URP).

No external service

The toolkit does not require an account, network connection, subscription, telemetry, or runtime data collection.

2. Quick Start

1 - Add the configured prefab

Drag RuntimeDebugToolkit/Runtime/Prefabs/RuntimeDebugToolkitUI.prefab into the scene. The prefab already contains the UI Document and all UI resources required by the toolkit.

> Recommended setup
>
> Use the configured prefab instead of building the object manually. This avoids missing UI references and provides the intended default styling.

2 - Check the Inspector status

Select the prefab instance. In Setup status , UI Document, Panel Settings, Visual Tree, and UI Resources should all be valid. If a status is missing, use the configured prefab again or restore the indicated reference.

3 - Choose how the toolkit opens

Use one of the activation methods below. Section 4 contains the complete setup and code examples.

Method	Best fit	Default behavior
Direct API	The project already has its own input or debug menu.	Call Show(), Hide(), or Toggle().
Legacy Input	The Legacy Input Manager is enabled.	Optional component using F1 by default.
Input System	com.unity.inputsystem is installed.	Optional component using Keyboard/F1 by default.

4 - Enter Play Mode

Trigger the selected activation method. The window opens on the left side of the display and can be moved by dragging its header.

Development builds only

The configured prefab enables Development builds only. Keep it enabled unless the toolkit is intentionally part of the release build. Build behavior is explained in section 10.

3. Inspecting Runtime Objects

GameObjects

The first panel lists root GameObjects from the active scene and DontDestroyOnLoad. Select an object to view its Components. Use Open and Close to browse children.

- The search field filters root GameObject names.
- Children remain grouped under their parent instead of being flattened into the search results.
- Use Refresh after creating, destroying, or moving objects when an immediate update is needed.

Components

Select a Component to display its inspectable values. Missing Script entries are ignored. The Component search filters the currently displayed Component type names.

Variables

The toolkit inspects public instance fields and non-indexed properties. Simple values are displayed when they are readable, writable, and supported. Readable complex objects can be opened to inspect editable children.

- The search field filters top-level variable names on the selected Component.
- Nested objects are loaded only when opened, keeping large Components readable.
- Repeated references are not expanded again, preventing circular hierarchies.
- If the selected GameObject or Component is destroyed, the interface returns to a valid selection state.

The complete built-in type list and conversion behavior are in section 5. To add a project-specific type, see section 7.

Editing values

Most inputs commit after editing is complete. While a field is focused, live synchronization does not overwrite the value being entered. Game logic can still change the value again after it is committed.

4. Activation API

Direct API

Direct control is the most flexible option because it uses the project's existing input flow and adds no input dependency.

```
using RuntimeDebugToolkit.UI;
using UnityEngine;

public sealed class DebugMenuController : MonoBehaviour
{
    [SerializeField] private RuntimeDebugToolkitUI toolkit;

    public void ToggleDebugMenu()
    {
        toolkit.Toggle();
    }
}
```

RuntimeDebugToolkitUI also provides Show(), Hide(), and IsVisible.

Legacy Input Manager

Add RuntimeDebugToolkitLegacyInputTrigger to the toolkit GameObject, assign the RuntimeDebugToolkitUI reference, and choose a key. The component is available only when the Legacy Input Manager backend is enabled.

Input System

When com.unity.inputsystem is installed, add RuntimeDebugToolkitInputSystemTrigger and assign the toolkit reference. Its embedded input action uses Keyboard/F1 by default and can be edited in the Inspector.

> Optional dependency
>
> Runtime Debug Toolkit does not install the Input System package. The integration becomes available automatically when the project already uses that package.

5. Supported Variable Types

Field	Supported types	User-facing behavior
Text	string	Delayed single-line text field.
Integral	sbyte, byte, short, ushort, int, uint, long, ulong	Uses the matching signed, unsigned, or wide integer field. Narrow values are clamped.
Decimal	float, double, decimal	Uses the matching decimal input.
Boolean	bool	Toggle field.
Enum	Any enum	Dropdown populated from the enum names.
Vector2	Vector2, Vector2Int	X and Y fields. Vector2Int discards decimal input.
Vector3	Vector3, Vector3Int	X, Y, and Z fields. Vector3Int discards decimal input.
Color	Color, Color32	RGBA sliders and numeric channels. Editing uses 8-bit channel precision.
Quaternion	Quaternion	Raw X, Y, Z, and W components. Values are not converted to Euler angles or normalized automatically.

Read-only and unsupported values

A simple read-only value is not shown as an editable field. A readable complex object can still expose editable child members. Unsupported simple types are skipped until a custom field mapping is added as described in section 7.

6. Registering Debug Actions

Actions provide controlled project commands such as healing a character, changing an environment setting, spawning a test object, or resetting a scenario. Action names must be unique.

Parameterless action

```
private const string ResetAction = "Debug / Reset Match";

private void OnEnable()
{
    RuntimeDebugManager.RegisterAction(ResetAction, ResetMatch);
}

private void OnDisable()
{
    RuntimeDebugManager.UnregisterAction(ResetAction);
}

private void ResetMatch()
{
    // Project-specific reset logic.
}
```

Actions with parameters

```
private void OnEnable()
{
    RuntimeDebugManager.RegisterAction<float>("Debug / Heal", Heal);
    RuntimeDebugManager.RegisterAction<Color>("Debug / Set Tint", SetTint);
    RuntimeDebugManager.RegisterAction<Vector3, bool>("Debug / Teleport", Teleport);
}
```

The API supports zero to three parameters. Named callback methods provide the clearest parameter labels in the popup.

Action lifecycle

Register actions when their owning system becomes available and unregister them when it is disabled or destroyed. This prevents duplicate names and actions that point to unavailable objects.

Running an action

Parameterless actions run directly. Parameterized actions open a popup. If a parameter type has no usable field, Run is disabled and the unsupported type is shown. See section 5 for built-in types and section 7 for custom support.

7. Extending the Toolkit

A custom variable field is a complete mapping between a CLR type, a field category, a binder, and a UXML template. All four parts are required before the value can appear and be edited in the runtime interface.

Adding a custom variable field

The following example adds an editor for UnityEngine.Bounds. Use the same sequence for another type, replacing the controls and conversion logic as needed.

1. Add a field category

Add a unique value to VariableFieldType. The enum identifies the Inspector template and binder mapping; it does not identify the CLR type by itself.

```
public enum VariableFieldType
{
    // Built-in values...
    Quaternion = 8,
    Bounds = 9
}
```

Keep custom numeric values stable. Because this enum is package source code, retain the custom entry when merging a future package update.

2. Create the binder

Derive from VariableFieldBinder. Query the controls declared by the UXML, copy the runtime value into them with SetValueWithoutNotify, and register every value-change callback in OnBind. Remove the same callbacks in OnUnbind.

```
using RuntimeDebugToolkit.Core;
using RuntimeDebugToolkit.UI.Binders;
using UnityEngine;
using UnityEngine.UIElements;

public sealed class BoundsVariableBinder : VariableFieldBinder
{
    private readonly Vector3Field _center;
    private readonly Vector3Field _size;

    public BoundsVariableBinder(DebuggedVariable variable, VisualElement element)
        : base(variable, element)
    {
        _center = element.Q<Vector3Field>("bounds-center");
        _size = element.Q<Vector3Field>("bounds-size");
    }

    public override void OnRefreshDisplay()
    {
        var value = (Bounds)_variable.GetValue();
        _center.SetValueWithoutNotify(value.center);
        _size.SetValueWithoutNotify(value.size);
    }

    protected override void OnBind()
    {
        _center.RegisterValueChangedCallback(OnCenterChanged);
        _size.RegisterValueChangedCallback(OnSizeChanged);
    }

    protected override void OnUnbind()
    {
        _center.UnregisterValueChangedCallback(OnCenterChanged);
        _size.UnregisterValueChangedCallback(OnSizeChanged);
    }

    private void OnCenterChanged(ChangeEvent<Vector3> evt)
        => _variable.SetValue(new Bounds(evt.newValue, _size.value));

    private void OnSizeChanged(ChangeEvent<Vector3> evt)
        => _variable.SetValue(new Bounds(_center.value, evt.newValue));
}
```

3. Create the UXML template

Create one UXML asset for the field. The toolkit clones this asset for every matching variable. The two metadata labels must be named variable-name and variable-type; the toolkit fills them automatically. Every control queried by the binder must also keep the exact name used in C#.

```
<ui:UXML xmlns:ui="UnityEngine.UIElements" editor-extension-mode="False">
    <!-- Binder contract: keep both metadata labels and both Vector3Field names. -->
    <Style src="../Styles/RuntimeDebugToolkitLists.uss"/>
    <Style src="../Styles/RuntimeDebugToolkitFields.uss"/>
    <Style src="../Styles/RuntimeDebugToolkitControls.uss"/>

    <ui:VisualElement class="rdt-variable-item">
        <ui:VisualElement class="rdt-variable-details">
            <ui:Label name="variable-name" text="Bounds" class="rdt-item-title"/>
            <ui:Label name="variable-type" text="Bounds" class="rdt-item-subtitle"/>
        </ui:VisualElement>
        <ui:VisualElement class="rdt-variable-control">
            <ui:Vector3Field name="bounds-center" label="Center" class="rdt-input"/>
            <ui:Vector3Field name="bounds-size" label="Size" class="rdt-input"/>
        </ui:VisualElement>
    </ui:VisualElement>
</ui:UXML>
```

The stylesheet paths above match a template stored beside the built-in VariableFields templates. If the custom UXML is stored elsewhere, update each relative path in UI Builder. The classes are recommended for visual consistency, while the named elements form the functional contract.

4. Assign the template in the Inspector

Select the GameObject containing RuntimeDebugToolkitUI. Under UI resources > Variable field templates, select + Add field template, set Field type to Bounds, and assign the new UXML asset. A binder registration does not create or assign this VisualTreeAsset automatically.

5. Register the CLR type and binder

Register the mapping once per Play session. BeforeSceneLoad runs after the toolkit resets its static registries and remains reliable when Domain Reload is disabled.

```
using UnityEngine;

public static class BoundsFieldRegistration
{
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    private static void Register()
    {
        VariableTypeMapper.RegisterBinder(
            VariableFieldType.Bounds,
            (variable, element) => new BoundsVariableBinder(variable, element),
            typeof(Bounds));
    }
}
```

Registering later from Awake or another runtime startup point is also supported: the active interface refreshes when the mapping changes. Do not rely only on a static constructor, because it is not rerun between Play sessions when Domain Reload is disabled.

> Validation checklist
>
> - The CLR type is included in supportedTypes.
> - The enum value has a UXML template in the RuntimeDebugToolkitUI Inspector.
> - UXML names exactly match every Q<T>() query in the binder.
> - OnRefreshDisplay does not emit change events.
> - Every callback registered in OnBind is removed in OnUnbind.

Creating a custom parser

A parser is only needed when a value must be converted from text through DebuggedVariable.SetRawValue(). A binder that writes an already typed value with SetValue(), such as the Bounds binder above, does not require a parser.

Implement IDebuggedVariableParser. SupportedTypes declares the exact CLR types handled by the parser, and TryParse returns the converted object without throwing for ordinary invalid input.

```
using System;
using RuntimeDebugToolkit.Parsers;

public sealed class GuidDebugParser : IDebuggedVariableParser
{
    public Type[] SupportedTypes => new[] { typeof(Guid) };

    public bool TryParse(string value, Type variableType, out object parsedValue)
    {
        if (variableType == typeof(Guid) && Guid.TryParse(value, out var guid))
        {
            parsedValue = guid;
            return true;
        }

        parsedValue = null;
        return false;
    }
}
```

Registering and using the parser

```
using UnityEngine;

public static class DebugParserRegistration
{
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    private static void Register()
    {
        RuntimeDebugManager.RegisterVariableParsers(new GuidDebugParser());
    }
}
```

A new parser replaces the previous parser for the same supported type. DebuggedVariable.SetRawValue() looks up the parser registered for the variable's declared type, calls TryParse, then assigns the converted value.

Binder or parser?

Use a binder to define how a type is displayed and edited in UI Toolkit. Use a parser to convert text into that type. A custom feature may use either one or both, depending on whether its UI controls produce typed values or raw text.

For visual-only changes to existing fields, keep the current binder and follow section 8 instead of registering a new field category.

8. Customizing UXML and USS

The interface can be restyled or rearranged with UI Builder. Start from a duplicate of the configured prefab and its UI assets so package updates do not overwrite project-specific presentation.

UXML contracts

Named elements queried by C# or a binder must keep their names. Every included UXML file contains a short contract comment identifying these elements. Visual hierarchy and classes can be adjusted as long as the required elements still exist.

USS organization

Style sheet	Purpose
Window	Window shell, header, tabs, sizing, and visibility.
Controls	Buttons, toolbars, search fields, and common inputs.
Lists and Panels	Rows, titles, counters, and section sizing.
Fields	Built-in variable editors and nested object layout.
GameObject Tree	Parent, child, selection, and expansion states.
Actions and Popups	Execution feedback and modal editors.
ScrollViews	List scrolling and scrollbar presentation.

Safe styling rules

- Keep the rdt- prefix for toolkit classes to avoid collisions with the project.
- Keep semantic is-* state classes used by the runtime code.
- Keep stylesheet references relative so the toolkit can live under a different Assets path.
- After upgrading Unity, verify the scoped .unity-* selectors because UI Toolkit may change its generated control hierarchy.

To create a new functional value editor rather than only changing presentation, follow section 7.

9. Demo Scene

Open RuntimeDebugToolkit/Demo/Scenes/RuntimeDebugToolkitDemo.unity to try the complete workflow. The scene uses Unity primitives and package materials, so no external art asset is required.

> Render pipeline
>
> The included demo scene and materials are configured for URP. The runtime toolkit itself does not depend on URP rendering APIs.

Suggested walkthrough

- Open the toolkit from the bottom-right demo HUD.
- Select the primary inspection drone and edit health, movement, color, enum, vector, quaternion, and nested profile values.
- Run the heal, damage, tint, teleport, spawn, clear, and reset actions.
- Spawn support drones, refresh the hierarchy, and inspect objects created at runtime.
- Inspect the environment Component and change its presentation settings.

The Demo folder is optional in a production project. Do not add the demo scene to Build Settings unless it is intentionally part of the application.

10. Build and Runtime Considerations

Release availability

The configured prefab enables Development builds only. In a non-development build, the toolkit removes itself at startup. Disable this option only when release access is intentional.

Fast Enter Play Mode

The toolkit supports entering Play Mode with Domain Reload disabled. At the beginning of each session it clears registered actions and custom parsers, restores the built-in parsers, and restores the built-in variable mappings.

Project-specific actions, parsers, and binders must therefore register once per Play session. Use OnEnable/Awake for instance-owned registrations or RuntimeInitializeOnLoadMethod with BeforeSceneLoad for static registrations. Examples are provided in sections 6 and 7.

Why nested content is loaded on demand

Large runtime objects can expose many nested members. The toolkit waits until Open is selected before creating child fields, keeping the initial interface responsive and avoiding work for data the user never views.

Runtime value updates

Displayed values synchronize while the window is visible. Refresh pauses for the field currently being edited so live game values do not interrupt user input.

Reflection and code stripping

The toolkit discovers public members through reflection. Test the final platform, scripting backend, and managed stripping level. If an expected member is absent in an IL2CPP build, preserve that member or its owning type through the project's link.xml or Preserve configuration, then rebuild.

Practical performance guidance

- Avoid expensive work or side effects inside properties intended for inspection.
- Close the toolkit when it is not needed during profiling.
- Use the manual refresh controls after large hierarchy changes.

Release safety

Do not register actions that expose credentials, personal information, destructive operations, or privileged production functionality. Keep the toolkit development-only unless release access is a deliberate product feature.

11. Troubleshooting

Symptom	Resolution
The toolkit does not appear	Confirm the prefab is active and use an activation method from section 4. In a release build, check Development builds only as described in section 10.
The Inspector reports missing setup	Use the configured prefab or restore the UI Document, Panel Settings, Visual Tree, and UI resource references indicated by Setup status.
F1 does nothing	Confirm the appropriate trigger is attached and assigned. Verify the project's selected input backend; see section 4.
A variable is missing	Confirm it is a public instance field or non-indexed property. Simple values also require a setter and a supported field; see sections 5 and 7.
A complex object has no editable children	Its children may be read-only, unsupported, null, cyclic, destroyed, or unavailable through a getter.
An action is unavailable	At least one parameter type has no usable field mapping or template. See sections 5 and 7.
A value returns after editing	Confirm the input was committed and check whether game logic or the property's setter changes it again.
A custom registration disappears on the next Play	Register it once per session with Awake, OnEnable, or BeforeSceneLoad. Do not rely only on a static constructor; see sections 7 and 10.
A custom parser is not called	Register it before use and call DebuggedVariable.SetRawValue() for that type. See section 7.
Styling changed after a Unity upgrade	Inspect the scoped .unity-* selectors with UI Toolkit Debugger; see section 8.
A reflected value is missing in IL2CPP	Review the stripping guidance in section 10 and preserve the expected member or type.

12. Public API Summary

Type	Purpose
RuntimeDebugToolkitUI	Shows, hides, toggles, and reports the visibility of the runtime interface.
RuntimeDebugManager	Registers parsers and actions, unregisters actions, clears runtime registries, invokes actions, and returns the current action list.
DebuggedClass	Represents an inspected object and exposes its available public variables.
DebuggedVariable	Represents a field, property, or action parameter and provides typed or text-based value access.
DebuggedAction	Represents a named callback and its configurable parameters.
VariableTypeMapper	Associates CLR types with UI field categories and binder factories.
VariableFieldBinder	Base class for synchronizing a DebuggedVariable with a UI Toolkit template.
IDebuggedVariableParser	Converts text into a supported runtime value type.

Public API documentation is available through IDE code completion. Usage workflows are described in sections 4, 6, and 7.

Support requests

Include the Unity version, target platform, render pipeline, scripting backend, relevant Console output, and clear reproduction steps when contacting support.

Runtime Debug Toolkit

End of user guide