Table of Contents

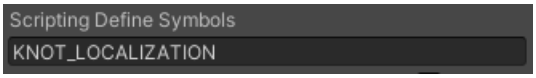
KNOT Localization is a lightweight, scalable and extensible texts & assets localization system for Unity.

- Uses traditional key-value pair collection as a source data structure along with convenient user-friendly interface to access localizable data in one place without having to manually synchronize keys and switch between localization source files.
- For game designers, it keeps editor workflow as simple as possible: Add Key > Set localized Value > Add Key Reference > Done!
- For programmers, it has modular architecture with the possibility to implement custom Texts & Assets data sources, runtime data loading & unloading logic (including remote), editable as property & runtime accessible metadata and even the whole localization manager without modifying the codebase.
- Compatible with all platforms.
- Full Editor Undo / Redo support.
- Well organized and commented source code without custom dependencies.

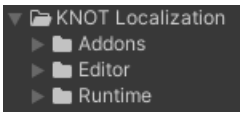
1. Installation

1.1 Download and import package.

- KnotProjectSettings asset will be created under your root **Assets** folder.
- **KNOT_LOCALIZATION** preprocessor will be added to your Scripting Define Symbols.



The root **KNOT Localization** folder structure should look like this:



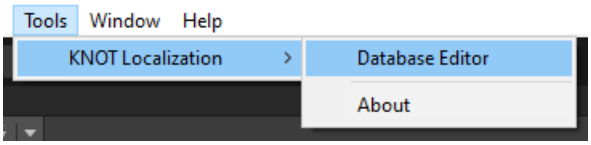
- **Editor** & **Runtime** folders contains core scripts and resources.
- **Addons** contains preinstalled addons like **Localized Components**, **Demo**, **Tests** etc.

You are able to move **KnotProjectSettings** asset and **KNOT Localization** folder to any subfolder in your project.

1.2 Before continue working with KNOT Localization it is recommended to run the Tests to make sure that core functionality is working as expected in a context of your project and Unity version you are using.

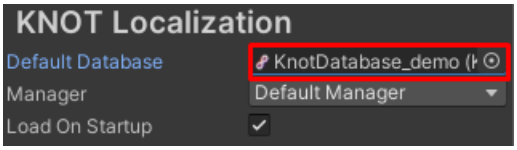
2. Creating Database

2.1 Open **Tools/KNOT Localization/Database Editor** window and create new Database asset as suggested.



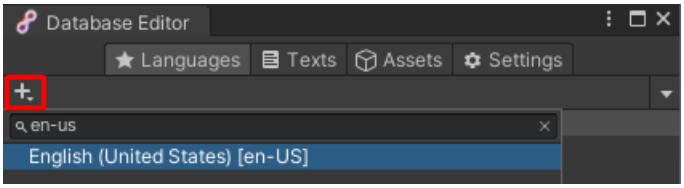
> **NOTE**
>
> You can also create new **Database** using Project window context menu - **Create/KNOT Localization/Database**

2.2 Go to **Edit/Project Settings.../KNOT Localization** and ensure that your newly created **Database** is assigned to **Default Database**.

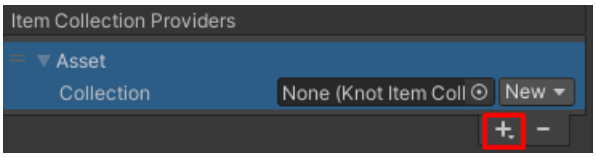


3. Adding Text Collection

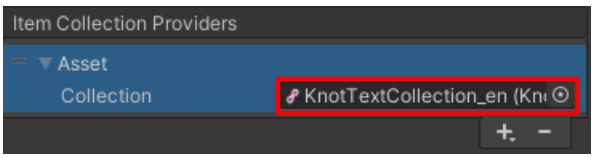
3.1 Switch back to **Database Editor** window and open **Languages** tab. Click **+** to select Culture Name and create new Language.



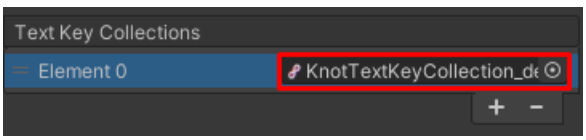
3.2 Select newly created Language and add Asset Collection Provider to **Item Collection Providers**.



3.3 Create and assign new Text Collection asset to **Asset Collection Provider**.

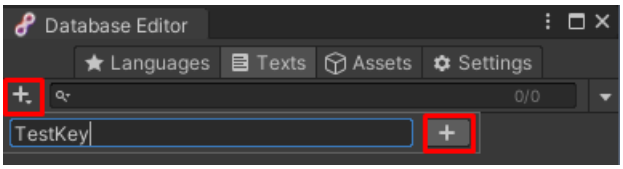


3.4 Open **Settings** tab, create and assign new Text Key Collection asset.

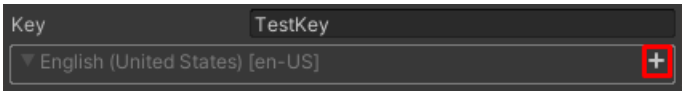


4. Adding Text Keys

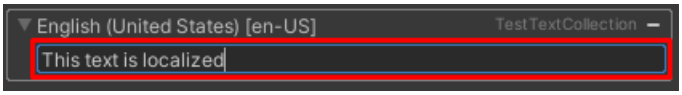
4.1 Open **Texts** tab and create new **Key**.



4.2 Select newly created Key, click **+** to add new value to the corresponding **Language**



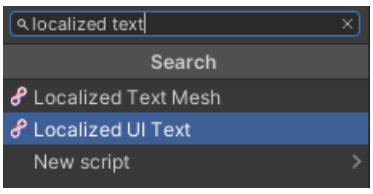
4.3 Enter localized text.



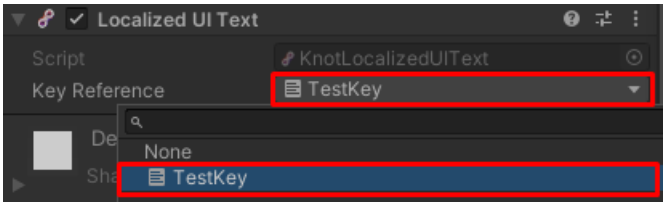
5. Testing Components

5.1 Make sure that your project has **Unity UI** package installed in Package Manager.

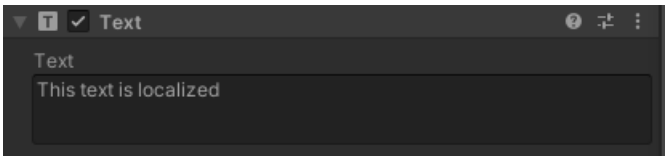
5.2 Create new Game Object with Unity's native **Text** component and attach **Localized UI Text** component.



5.3 Assign previously created Key to Key Reference field.



5.4 Hit **Play**. Localized text will be assigned to **Text** component.


Text
Text
This text is localized

Changing Language

```
KnotLanguageData targetLanguage = KnotLocalization.Manager.Languages.FirstOrDefault(d => d.SystemLanguage ==
SystemLanguage.English);
if (targetLanguage != null)
    KnotLocalization.Manager.LoadLanguage(targetLanguage);
```

Subscribing to Language Loaded callback

```
KnotLocalization.Manager.StateChanged += state =>
{
    if (state == KnotManagerState.LanguageLoaded)
    {
        //Selected or startup language is loaded
    }
};
```

Getting Text Value

```
string myLocalizedText = KnotLocalization.GetText("myKey");
```

or

```
string myLocalizedText = KnotLocalization.Manager.GetTextValue("myKey").Value;
```

or

```
KnotTextKeyReference myKeyRef = new KnotTextKeyReference("myKey");
string myLocalizedText = myKeyRef.Value;
```

Subscribing to Text Updated callback

```
void OnEnable()
{
    KnotLocalization.RegisterTextUpdatedCallback("myKey", TextUpdated);
}

void OnDisable()
{
    KnotLocalization.UnRegisterTextUpdatedCallback("myKey", TextUpdated);
}

void TextUpdated(string text)
{
    //Do something with localized text assigned to myKey
}
```

or

```
KnotTextKeyReference myKeyRef = new KnotTextKeyReference("myKey");
myKeyRef.ValueUpdated += text =>
{
    //Do something with localized text assigned to myKey
};
```

Accessing Metadata

Database

```
MyCustomMetadata myMetadata = KnotLocalization.Manager.Database.Settings.Metadata.OfType<MyCustomMetadata>().FirstOrDefault();
```

Selected Language

```
MyCustomMetadata myMetadata = KnotLocalization.Manager.SelectedLanguage.Metadata.OfType<MyCustomMetadata>().FirstOrDefault();
```

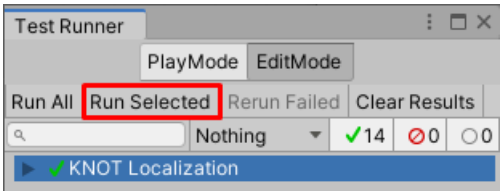
Text Key

```
MyCustomMetadata myMetadata = KnotLocalization.Manager.GetTextValue("MyKey").Metadata.OfType<MyCustomMetadata>
().FirstOrDefault();
```

Creating simple Database from scratch at runtime

```
void Awake()
{
    KnotDatabase myDatabase = ScriptableObject.CreateInstance<KnotDatabase>();

    KnotLanguageData myLanguage = new KnotLanguageData(SystemLanguage.English);
    KnotTextCollection myTextCollection = ScriptableObject.CreateInstance<KnotTextCollection>();
    myTextCollection.Add(new KnotTextData("myKey", "myText"));

    myLanguage.CollectionProviders.Add(new KnotAssetCollectionProvider(myTextCollection));
    myDatabase.Languages.Add(myLanguage);

    KnotLocalization.Manager.SetDatabase(myDatabase);
    KnotLocalization.Manager.LoadLanguage(myLanguage);

    KnotLocalization.Manager.StateChanged += OnStateChanged;
}

void OnStateChanged(KnotManagerState state)
{
    if (state == KnotManagerState.LanguageLoaded)
    {
        Debug.Log(KnotLocalization.GetText("myKey")); //myText
    }
}
```

1. Make sure that you have **Test Framework** package installed in Package Manager.
2. Go to **Window/General/Test Runner**, select **KNOT Localization** and hit **Run Selected**.



Open **KnotLocalization_demo** scene located under **KNOT Localization/Addons/Demo** folder and hit **Play**.

Project Settings asset is a project wide settings that keeps reference to default Database and Manager. It can be accessed in **Edit/Project Settings.../KNOT Localization**.

On application startup (After Loading Assemblies), if **Default Database** is assigned and **Load on Startup** is enabled, KnotLocalization will pass **Default Database** to **Manager** and forces to load startup language.

Reset User Settings clears current user related data like Current Active Database, expanded items in Database Editor and so on. This will not affect any project settings or assets data.

Database asset keeps the reference to Language Data collection, Text & Asset Key Collections, Text & Asset Controllers and Language Selector.

> **NOTE**
>
> You can create **Database** asset using Project window context menu - **Create/KNOT Localization/Database**

Language Data is defined by Culture Info and stores references to all language-related data in Collection Providers list.

Runtime Item Collection Provider implementation defines how Item Collections assets should be loaded & unloaded during runtime by Manager.

Persistent Item Collection Provider implementation is used by **Database Editor** to view and edit Item Collections values with Undo / Redo support.

For example, Resource Collection Provider implements asynchronous Item Collection loading & unloading. Storing Item Collections inside a Resources folder ensures that only Item Collections associated with Selected Language are loaded into memory at runtime.

You can define your own Item Collection Provider by creating a class that implements IKnotRuntimeItemCollectionProvider and IKnotPersistentItemCollectionProvider (optional). After that you are able to add it to Language Data in **Database Editor**.

NOTE

Custom class should have no constructor or at least one public constructor without arguments.

WARNING

If you decide to remove, change the name, namespace or assembly of custom class you will get serialization error and possibly lose data. As a temporary solution, add [**MovedFrom**] attribute to your class before making those changes.

Example Usage

Issue ID

Item Collection asset with IKnotItemCollection implementation stores Text & Asset Item Data collection. For example, Text Collection can be assigned to Asset Collection Provider and edited in **Database Editor**.

In runtime, it is passed to Controller by Manager with combination of Key Collection.

> **NOTE**
>
> You can create **Item Collection** asset using Project window context menu - **Create/KNOT Localization/**

Key Collection asset used to store Text & Asset keys with Metadata collection.

In runtime, it is passed by Manager to Controller with language-related Item Collection keys.

> **NOTE**
>
> You can create **Key Collection** asset using Project window context menu - **Create/KNOT Localization/Key Collection**

Metadata implementations stores custom data inside Metadata Container and can be edited as property in **Database Editor**. Metadata can be assigned to Database, Language Data and each key in Key Collection.

Metadata Container separates Runtime and Editor-only metadata to ensure that Editor-only metadata will not be included in build.

Built-in Metadata types

- Fallback Language. Can be added to Database or Language. Used by Manager to additionally load fallback Language Item Collections if some of the keys stored in Key Collection are not presented in Item Collections.
- Asset Type Restriction. Editor-only metadata that is used to restrict asset type for particular Asset Key.
- Comment.

Interfaces

- Implementing IKnotTextFormatterMetadata and IKnotAssetFormatterMetadata allows to format key value in runtime before accessing it.
- Implementing IKnotCultureSpecificMetadata allows to get Culture Info from selected language. Culture Info is provided before formatting values.

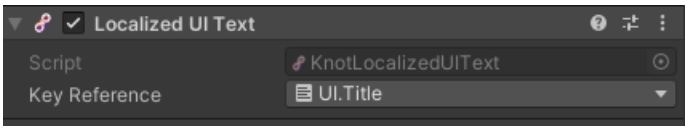
You can define your own custom Metadata class by implementing IKnotMetadata. After that you are able to add it in **Database Editor**.

> **NOTE**
>
> Custom class should have no constructor or at least one public constructor without arguments.

> **WARNING**
>
> If you decide to remove, change the name, namespace or assembly of custom class you will get serialization error and possibly lose data. As a temporary solution, add [**MovedFrom**] attribute to your class before making those changes.
>
> Example Usage
>
> Issue ID

Key Reference provides access to Text & Asset values along with Metadata at runtime.

For example, storing Text Key Reference as a field allows to pick Text Key from **Default Database** assigned in Project Settings.



Controller implementations is used to build Key-Value dictionary for selected language at runtime.

For example, Text Controller builds Texts values from provided Key Collection and Text Collections, allows to add or override custom text values and subscribe for value update callbacks that is invoked whenever the language or the value itself are changed.

You can define your own custom Controller class by implementing IKnotController interface or deriving from KnotController class. After that you are able to select it in **Database Editor** Settings.

NOTE

Custom class should have no constructor or at least one public constructor without arguments.

WARNING

If you decide to remove, change the name, namespace or assembly of custom class you will get serialization error and possibly lose data. As a temporary solution, add [**MovedFrom**] attribute to your class before making those changes.

Example Usage

Issue ID