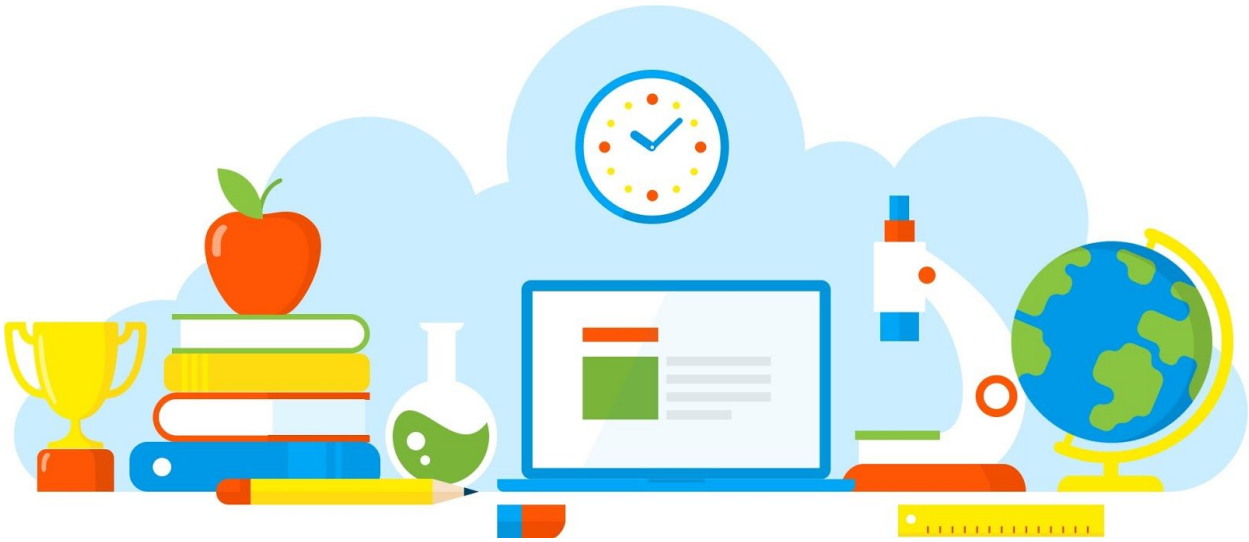


Unium

A Unity library for automated testing and development tools

http://github.com/gwaredd/unium

v1.0

Introduction

Unium is an experimental library for the purposes of facilitating automated testing and tool development for your Unity games.

The main idea is twofold. Firstly, embed a web server into your project to provide an interface onto the game. Secondly, implement a query language that to some degree takes care of the tedious bits.

The advantages of a web server is that HTTP provides a technology agnostic protocol that places no restrictions on whatever tools and frameworks you wish to use. It also means it will work whether the game is running in editor, on device or on some headless server in the clouds.

Hopefully the query language and automatic reflection reduces the amount of manual serialisation code that often makes these kind of systems a pain to work with.

Whilst the inspiration for the project was drawn from many sources, conversations and experiences, it was in some small part influenced by Selenium and Appium and therefore adopts this precedence for awful naming conventions :)

Feedback

I would love hear any feedback, comments, bugs or feature requests so please feel free to create an issue or pull request on github or mail me at gwaredd@hotmail.com.

https://github.com/gwaredd/unium

Getting started

An interactive tutorial can be found in the github project, just clone the repro and run the scene. This should open your browser automatically and demonstrate the key features in a live game.

It also includes some simple implementations to get you going.

If you have installed Unium from the Unity Asset Store then please ensure the tutorial folder is copied into your StreamingAssets folder.

To add Unium to your own projects ...

- Copy the Unium directory (Assets/Unium) into the Assets folder of your project
- Create an empty object in your scene and attach the `UniumComponent`
- Run the scene and check it is working by going to http://localhost:8342/about

NB: For builds on device, the ***Development Build*** flag must be enabled. Unium is disabled by default for release, although this behaviour is configurable.

GQL

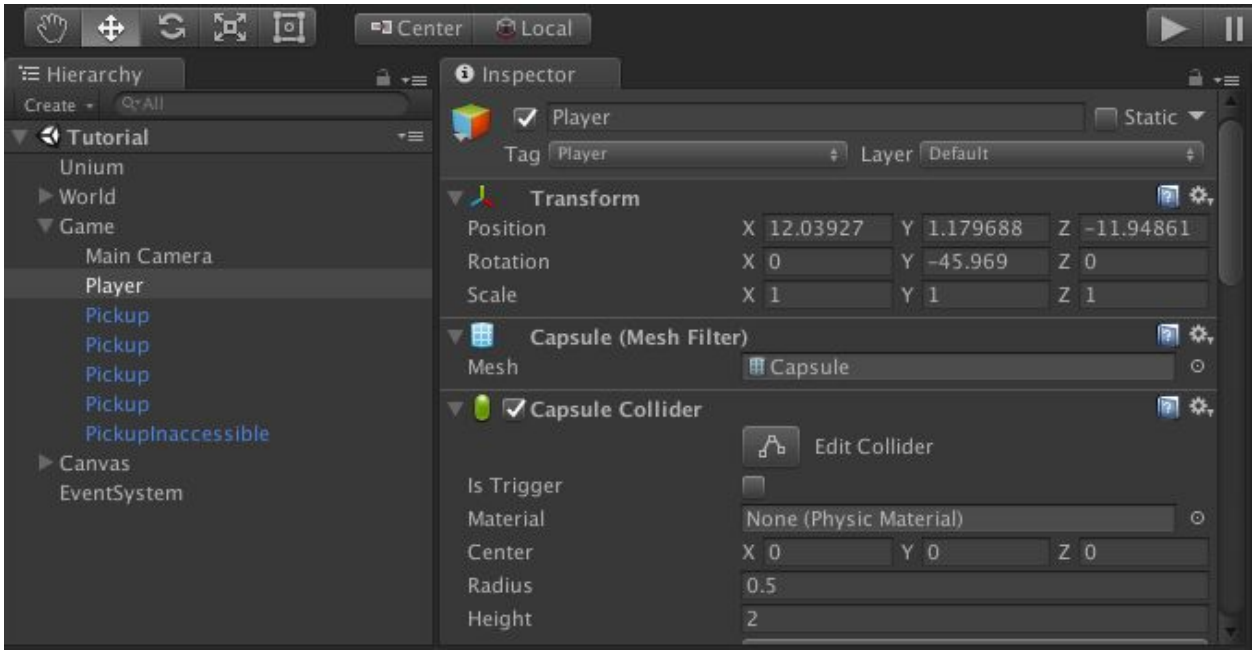
GQL (Game Query Language) is an XPath-like query language that lets you retrieve data from the game, set variables and invoke functions.

It is available on a special route called `/q` and operates on a custom tree data structure. The most important node in this tree is the scene.

The query selects all the objects in the tree that match that path and returns an array of the results (in JSON format). This may be an empty array if no objects match or there is an error executing the query.

GQL may also return HTTP errors if the query fails to parse or certain exceptions are thrown.

Examples below are from the tutorial level which has the following scene graph.



Getting Values

Queries select objects in the tree that match the path and reflect back their values.

Selecting object by name

Objects can be referenced by their path. For example, to get the player game object.

```
/q/scene/Game/Player
```

Accessing Properties

Properties on objects can be selected with dot notation.

For example, to get the position of the player

```
/q/scene/Game/Player.Transform.position
```

Or to just get the x position of the player

```
/q/scene/Game/Player.Transform.position.x
```

Multiple Matches

The path can match multiple objects.

To get all child objects of Game called Pickup

```
/q/scene/Game/Pickup
```

Wildcards

Wildcards can be used to match partial names.

To get all the child objects of Game with name that starts with P

```
/q/scene/Game/P*
```

Where Clauses

Square brackets `[]` can be used to filter objects based on an expression.

Variables within the brackets refer to properties of the current node.

The object itself can be referred to with `$` (mainly for arrays of primitive types, for example integers that pass some filter).

If the expression evaluates to an integer number it will be treated as an index into a list, otherwise it will filter the list based on the “truth” of the expression.

Find all the pickups and return the 3rd one from the list (0 based index naturally)

```
/q/scene/Game/Pickup[2]
```

Or as a filter, find the position of the pickups whose x value is greater than or equal to 0.

```
/q/scene/Game/Pickup.Transform.position[x>=0]
```

Or the filter applied to the Pickup, returning the the RotateSpeed value.

```
/q/scene/Game/Pickup[Transform.position.x>=0].Pickup.RotateSpeed
```

Finding Objects

You can use `//` to recursively search for child nodes that match the path that follows.

Find all the pickups in the scene (find by name)

```
/q/scene//Pickup
```

Or, find all the game objects with the Pickup tag

```
/q/scene//[tag='Pickup']
```

Find Objects with Component

By combining the recursive search with a wildcard you can find objects with specific components.

Find all objects in the scene with a Pickup component and return the RotateSpeed

```
/q/scene//*.Pickup.RotateSpeed
```

Setting Values

You can use GQL to change values by simply adding `=<some value>` to the end of the query. For example,

Change the rotate speed of all the pickups

```
/q/scene//Pickup.Pickup.RotateSpeed=5
```

Change the name of all the pickups

```
/q/scene//Pickup.name=Fish
```

Set the position of the player to the origin

```
/q/scene//Player.Transform.position={'x':0,'y':0,'z':0}
```

Currently only primitive types and Vector3 can be set.

The query will return the new values. If it cannot convert the string passed to the underlying value type it will be ignored.

Invoking Functions

If the selected object is a function you can invoke it by appending parenthesis `()` to the query.

Invoke `SomeFunction` on a component

```
/q/scene/Some/Object.Component.SomeFunction()
```

Invoke `SomeFunction` on a component with arguments

```
/q/scene/Some/Object.Component.SomeFunction(a,b,c)
```

Find the Player and call `MoveTo` with the origin as a target (Vector3).

```
/q/scene//Player.Player.MoveTo({'x':0,'y':0,'z':0})
```

Currently function arguments can only be primitive types or Vector3.

If the argument strings cannot be converted to the underlying value types then the function invocation will be ignored.

The query will return an array of the function return values (or nulls if this is a void type).

Expressions Reference

Note the following for expressions in GQL where clauses

- The standard C++ rules apply regarding operator precedence
- The following are equivalent
  - `&, &&` and `and`
  - `|, ||` and `or`
  - `=` and `==`
  - `!` and `not`
- You can use `true, false` and `null`

You can use the following functions, they map to the C# System.Math equivalents.

- abs
- cos
- sin
- tan
- sqrt
- ceil
- floor
- log
- round
- sign

Web Sockets

The RESTful interface is convenient but there are a couple of things it is not suitable for. Specifically, watching variables change over time and getting notifications about in-game events. This is where web sockets can help.

The tutorial comes with some simple example implementations that may be helpful reference.

Protocol

Unium communicates over websockets by passing JSON messages back and forth.

Requests sent to the game has an id and a query ...

```
{ id: "<message id>", q: "</query>" }
```

The id field is optional, the query is the same as the RESTful interface.

A reply is of the form:

```
{ id: "<message id>", data: ...}
```

Or

```
{ id: "<message id>", error: "..."}
```

Where the message id, if present, is the same as original request. This allows you to identify and “demux” the incoming message streams in the case where you have multiple messages in-flight.

For example

```
{ id: "about", q:”/about” }
```

May return something like ...

```
{ id: "about", data: {'Unium':'1.0',...} }
```

Repeaters

Watching variables is done with repeaters (horrible name, sorry). Repeaters repeat the query at a given period to “sample” the results. The default sample period is 1 second.

For example, get the current frame rate every second

```
{ id: "fps", q:"/q/stats.FPS", repeat:{} }
```

Sample the frame rate 4 times a second

```
{ id: "fps", q:"/q/stats.FPS", repeat:{ freq: 0.25 } }
```

Sample the frame rate every frame

```
{ id: "fps", q:"/q/stats.FPS", repeat:{ freq: 0 } }
```

Repeat takes these optional parameters:

```
skip    : <skip the first n number of samples>,
samples : <number of samples to collect>,
freq    : <numbers of seconds to wait between sampling>
```

The initial message will return before sending any sample data

```
{ id: "<message id>", info: "repeating" }
```

If the samples parameter is set it will return this message when finished

```
{ id: "<message id>", info: "finished" }
```

Using the stop socket command will return

```
{ id: "<message id>", info: "stopped" }
```

Binding Events

Unium can “bind” to any C# event with a delegate type of Action<object> using the `/bind` route.

Any events that fire will have the same message id as the bound event.

For example, to get the debug output ...

```
{ id: "debug", q:"/bind/events.debug" }
```

If an event is successfully bound, it will return

```
{ id: "<message id>", info:"bound" }
```

Using the unbind socket command will return

```
{ id: "<message id>", info:"unbound" }
```

Socket Commands

Additionally, sockets have their own interface for controlling behaviour. You can use the following as you would any other route.

`/socket.stop(<id>)`

Stop a repeater with the given message id, e.g. /socket.stop(fps)

`/socket.unbind(<id>)`

Unbind a previously bound event with the given message id, e.g. /socket.unbind(debug)

`/socket.ping()`

Return pong

`/socket.repeaterCount`

Return the number of active repeaters

Web Server

The `UniumComponent` is responsible for managing the web server and dispatching incoming requests to the appropriate route handler.

This section covers the underlying system should you need to work with it.

Component Settings

Port

Default 8342

For nostalgic reasons, Unium runs on port 8342. If this is not convenient, you can change this in the settings.

Run in background

Default true

If set, will sets the Application.runInBackground flag. Normally Unity will pause the game when going into the background, which can be inconvenient when working with external tools.

Autostart

Default true

Start the server automatically when the object is enabled. If this is not set you will have to explicitly call `StartServer()` on the component.

Static Files

Default null

The game will serve static files from the root of the `StreamingAssets` folder. This is convenient when you want to include a nice debug UI with a build. You can provide a subfolder here.

Enable Debug

Default false

Log requests as they are made.

Routes

Routes map incoming requests to associated handling routines by their URL (or partial URL). Whilst GQL provides a good deal of functionality, if you want to add something bespoke you can register your own routes.

The default routes along with the other configuration can be found in the `Unium.cs` file.

Add a route

```
RoutesHTTP.Add( "/about", HandlerUtils.HandlerAbout );
```

Maps any URL starting with /about to the HandlerAbout function.

Note that HTTP and WebSockets have their own separate routing configuration, although they can share handler functions.

Route Handler

The route handler is a function that takes two parameters, a RequestAdapter and a string. The adapter is an object that represents the incoming request.

The path is any remaining part of the URL after the registered path. If you need the full path, you can get this from the adapter.

```
public static void HandlerAbout( RequestAdapter req, string path )
{
    req.Respond( JsonReflector.Reflect( new
    {
        Unium = Unium.Version.ToString( 2 ),
        Unity = Application.unityVersion,
        ...
    }
```

```
}
```

Note that a handler ***must*** call one of the “response” functions on the request. These are:

- **Respond** - respond to the request with some data
- **Reject** - respond with an HTTP error code
- **Redirect** - redirect the caller to another route (does nothing for a WebSocket request)
- **Defer** - defer the request (see below)

The handler can also throw an error, which will cause it to respond with 400 Bad Request.

Deferred Requests

Sometimes a handler may not be able to respond immediately to a request. In this case they can create a ‘context’ and defer the request.

The context is attached to the request and it will be put into a queue to be “resubmitted” at a future point.

Note that in future this may be replaced with a coroutine.

Webserver Key Facts

Singleton

It only makes sense to have one web server. Any additional `UniumComponents` will automatically destroy themselves.

DontDestroyOnLoad

The object the component is attached to will be set to DontDestroyOnLoad, allowing it to persist through scene loading.

Multi-Threaded

For performance, the server is multithreaded. Incoming requests are processed on their own worker thread from the .NET ThreadPool.

This allows the server to process as much of the request as possible outside of the main game thread. However, the handler code is by default run on the game thread (otherwise you wouldn't be able to call much of the Unity API).

Be aware though, any route flagged as "immediate" will be dispatched on the worker thread. This makes sense for certain actions that do not require the Unity API, so if you choose to make use of this, make sure the handler is thread safe.

REST and Websockets

The web server supports both RESTful HTTP requests and WebSockets.

Miscellaneous

Performance

You can improve the runtime performance of the library by changing the .NET API level.

From the menu

```
File -> Build Settings -> Player Settings …
```

Set the API level

```
Other Settings -> (Configuration) -> Api Compatibility Level
```

To

```
.NET 2.0
```

This will enable regular expression compilation that is not available in the .NET 2.0 subset.

Note that in the future I plan to replace the regular expressions with a faster more bespoke parsing, so this will not be necessary.

Release builds

By default, Unium is only enabled for development builds or in the editor. It is completely compiled out for release builds. To change this behaviour you can define `UNIUM_ENABLE` or `UNIUM_DISABLE` as preprocessor directives accordingly.

From the menu

```
File -> Build Settings -> Player Settings …
```

Then

```
(Configuration) -> Scripting Define Symbols
```

GQL and DontDestroyOnLoad game objects

Unity removes objects set as DontDestroyOnLoad from the scene graph and places them in a special separate scene. This means you will not be able to query them with GQL in the normal way.

If you want to access them through GQL, add the `UniumRegister` component to the object. This will add them to the root of the GQL search tree. By default it will use the given name, although you can override this by setting the “Optional Name” property on the component.

Note these are accessed directly from `/q` rather than `/q/scene`, e.g.

```
http://localhost:8342/q/myobject
```

Default routes

The default configuration can be found in the `Unium.cs` file in the root of the directory.

Out of the box of the box it comes with some common routes that provide some functionality that you may find useful. Note that there is a separate configuration for RESTful and Websocket routes.

All routes return JSON data unless otherwise specified.

HTTP Routes

`/about`

Get some general information about the game instance.

`/q`

GQL end point.

`/utils/debug`

Returns the last 100 lines of the debug log output.

`/utils/screenshot`

Take a screenshot then get redirected to the file. Returns an image.

`/utils/scene/<name>`

Load a scene by name (e.g. /utils/scene/tutorial). If no name given then list all scenes.

`/file/streaming/<filename>`

Load a file from the `StreamingAssets` folder. If the path is a directory then list the contents.

`/file/persistent/<filename>`

Load a file from the persistent assets folder. If the path is a directory then list the contents.

Websocket Routes

`/about`

Get some general information about the game instance.

`/q`

GQL end point.

`/utils/scene/<name>`

Load a scene by name (e.g. /utils/scene/tutorial). If no name given then list all scenes.

`/bind`

This uses GQL to select event objects from the search root and add callbacks. This lets you watch game events. For example, to register to receive debug out from the game the then /bind/events.debug

`/socket`

Special end point for socket control.

GQL Nodes

Default top-level nodes of the GQL search tree.

`/q/scene`

The Unity scene graph.

`/q/stats`

Basic stats about the current game instance.

`/q/events.debug`

Fires an event for every debug log message.

`/q/events.sceneLoaded`

Fires an event when a scene is loaded.

Roadmap

The library is largely experimental. If it proves useful I will be improving it over time. Below are a list of things I may or may not be adding in no particular order. If you have any strong feelings feel free to put in a feature request as an issue on github.

Callouts and cloud testing

Whilst connecting from a tool to a game instance works for most use cases, there are situations where it would be convenient for the game to connect to a tool or service.

For example, if you have a bank of test machines and you want the game to register that it is ready for testing. Or if the network infrastructure prevents you from connecting to the port (for example, blocked by a firewall).

Ideally we want to be able to use the cloud hosted devices (e.g. AWS Device Farm) to farm out automated tests on demand.

Passing complex objects through GQL

Currently you can only pass primitives or Vector3's for setting values or invoking functions. Whilst you can work around this restriction, serialising more complex value types should be supported.

Named objects and GQL parameters

Currently there is no good way of using the results of one query in another. Saving named results and implementing "GQL parameters" may provide a mechanism for this.

Recording and playback

Theoretically, it should be possible to record and playback player input to automatically generate test scripts. This would speed up authoring automated tests significantly.

Automatic game learning

Theoretically, GQL can infer state, so there is an idea to use neural networks or some kind of search based algorithm (e.g. MCTS or RHEA) to learn how to play any game

automatically. This means you get automated playthroughs for free (with a bit of processor time set aside for learning).

In theory this could automatically find bugs based on the difference between expected learned q-values and actuals … with a bit of machine learning added in.

Probably.

Extras

The “extras” folder contains a few supporting projects. Some of these are work in progress or placeholders. However, the following two directories may be of interest.

Examples

Simple examples demonstrating possible use cases.

A La Carte

A web application that lets you quickly knock together simple debug menus.

Additional Components

UniumSimulate

The UniumSimulate component uses the Unity EventSystem to simulate UI events. This can be handy for tests that require actions such as “click on these x,y screen coordinates”.

This is not a perfect emulation of input events. MouseDown handlers will not be invoked and it will only hit test against the UI canvas. Future versions may add this functionality.

To use, add the `UniumSimulate` component to the same game object that has the `UniumComponent`. This creates a new endpoint `/q/simulate` with the test functions, which will return the path to the game object hit or null.

`/q/simulate.click(x,y)`

Emulate clicking on the screen at the x,y position. This will invoke the IPointerClickHandler on the first underlying game object.

For example: `/q/simulate.click(20,600)`

`/q/simulate.drag(fromX,fromY,toX,toY,time)`

Emulate dragging an object from a given x,y position to another x,y position over a given time. This invokes the underlying IBeginDragHandler, IDragHandler and IEndDragHandler accordingly.

For example: `/q/simulate.drag(0,0,100,100,0.5)` to drag from 0,0 to 100,100 over 0.5 seconds.

Coordinates are in screen space. There are also normalised versions of the functions that take a position in the range [0,1] and converts them to screen coordinates.

`/q/simulate.clickNormalised`

`/q/simulate.dragNormalised`

License

The code is free to use and modify as you wish without restrictions. However, it would be nice to hear about projects using it and your experiences with it, so please get in touch.

GL & HF

Gw