

GRIMQ DOCUMENTATION – 1.3

GrimQ is primarily a query library for Legend of Grimrock entities. The syntax is very similar to LINQ and reminds of SQL, for those who already know those languages.

Besides query capability, GrimQ offers additional domain-specific utility functions which can be useful to needs different from just querying and thus offer an advantage also to those who are not interested in the querying part (which is the most of the library anyway). If that's you, jump to the Grimrock Specific Functions chapter to skip the querying part.

GrimQ Documentation – 1.3	1
Introduction to queries.....	3
Important Cautions!	3
Setup	3
Importing Data Sources	4
Automatic Import (from)	4
Importing Arrays	4
Importing Arbitrary tables (Dictionaries).....	4
Importing Iterators, Empty importer and Grimrock Specific Initializers	5
Performing the query	6
Projection Methods – select, selectMany	6
Filtering Methods – where, whereIndex, take, skip	7
Set Methods – union, except, intersection, distinct	8
Combination Methods – concat, zip	8
Terminating methods – Getting a Result	9
Picking Methods – first, Last, random, min, max.....	9
Inspection Methods – any, all, count, contains	9
Aggregation Methods – sum, average	10
Visitor Methods – foreach, map, xmap.....	10
Conversion methods - toArray, toDictionary, toIterator.....	11
Grimrock Specific Functions.....	13
Enumerations.....	13
Query initializer functions	13
Predefined predicates.....	14
Utility functions	15
String functions	16
Auto objects	16
Configuration	18
Jkos framework integration	18

Known Issues 19

Common query examples..... 19

Credits, License and Disclaimers 20

INTRODUCTION TO QUERIES

A GrimQ query is composed by three parts:

- Importing a data source into GrimQ
- Querying the GrimQ collection
- Terminating the query, by returning a result or by performing an action

For example:

```
grimq.from(allEntities(1))  
  :where(function(v) return #v.name > 5; end)  
  :select(function(v) return v.name; end)  
  :foreach(print)
```

Is composed by those three parts:

- `grimq.from` – imports the entities returned by “allEntities” into the GrimQ structure
- `:where` and `:select` – perform the query by pruning data and by transforming it
- `:foreach(print)` – ends the query performing an action (calling `print` for every element in the collection). We will use this a lot in our examples at first – for the moment, trust that all that *foreach(print)* does is printing each element of the collection.

IMPORTANT CAUTIONS!

Legend of Grimrock does not support arbitrary data in save games. In particular only strings, booleans, numbers and tables of these types (including tables of tables of tables of integers for example) can be stored in a saved game.

Thus the following precautions should be taken at the moment:

- Always declare the `grimq` object as local (or do not declare it at all)
- If a collection result must be stored:
 - always go to one of the `toArray/toDictionary` methods (`toIterator` returns an iterator, which also is not compatible with saved games).
 - ensure you only have strings, numbers and booleans in your final collection

This might seem limiting and difficult to enforce, but really if you consume the result of your queries right away, you don't risk any problem.

Testing regularly with saved games is recommended.

SETUP

To setup GrimQ for use, simply copy the source of `grimq.lua` file and paste it into a scripting entity named “grimq”.

At the top of that entity is a variable called **MAXLEVEL** – update it during setup and everytime you change the number of levels in the dungeon (until Almost Human releases the next version :)).

IMPORTING DATA SOURCES

A lot of data sources can be imported into GrimQ:

- Automatically with auto-detection of the best method to use through the *from* method
- Arrays (tables with consecutive integer keys): through the *fromArray* method or through the *fromArrayInstance* method
- Dictionary (tables with key-value pairs): through the *fromDictionary* method
- Iterators: through the *fromIterator* method
- Nothing: through the *fromNothing* method
- Grimrock specific imports like *fromChampions*, *fromPartyInventory*, etc.

AUTOMATIC IMPORT (FROM)

If using the *from* method, an automatic inspection of the argument leads to the method using the right initializer for most cases.

Where an ambiguity may arise is when telling between Arrays and Dictionaries: the *from* method checks if an item with key 1 exists in the given table, if it exists, it imports it as an Array, otherwise it imports it as a Dictionary.

Even if you use *from* to import from a data source, read the following paragraphs, as you need to know how the import is done under the hood.

IMPORTING ARRAYS

To import arrays (apart from the *from* method) two methods exist: *fromArray* and *fromArrayInstance*.

The first one creates a copy of the data-source, the second uses the data-source itself for the first query execution. In general, the only difference is that if you use *fromArrayInstance* and you change the original data-source before performing the first query method, the result will change. On a performance point-of-view, *fromArrayInstance* is faster.

The *from* method automatically uses *fromArrayInstance*.

Example:

```
local array = { "ciao", "hello", "au revoir" }
grimq.from(array)
    :foreach(print)
```

Prints

```
ciao
hello
au revoir
```

IMPORTING ARBITRARY TABLES (DICTIONARIES)

To import an arbitrary table – which we call a dictionary - use the *fromDictionary* method, or the *from* method. The resulting collection will be of subtables containing two items: *key* – the item which was the key in the original table, and *value* – the item which was the value in the original table.

Note that the if your dictionary uses the number 1 as a key, you won't be able to use the *from* method to import it. If your dictionary uses number, it's recommended that you explicitly use *fromDictionary* instead.

IMPORTING ITERATORS, EMPTY IMPORTER AND GRIMROCK SPECIFIC INITIALIZERS

You can use the *fromIterator* method to import from an iterator. The iterator must be one returning a single item.

If you have an array of iterators, you can use *fromIteratorsArray* to import them all in a single GrimQ collection.

You can also use the *fromNothing* method to start with an empty GrimQ structure (or *from(nil)*).

Finally there are many import methods which are domain specific to Grimrock, like *fromChampions*, *fromPartyInventory*, etc. For these, reference the domain specific method reference.

PERFORMING THE QUERY

After a data-source has been imported, it's time to perform a query on the data, that is transform the collection into another collection closer to the result we want. We can use:

- Projection methods: *select*, *selectMany*
- Filtering methods: *where*, *whereIndex*, *take*, *skip*
- Set methods: *union*, *except*, *intersection*, *distinct*
- Combination methods: *concat*, *zip*

Note for advanced nitpickers: it might seem strange that *select* isn't a filtering method as filtering is also called "selection"; blame it on SQL, LINQ and other query languages which in common practice changed the meaning of *select* methods. This syntax leads to query which are in a format more familiar among those used to query languages, so I followed this common practice.

PROJECTION METHODS – SELECT, SELECTMANY

The *select* method allows to take the elements in the collection and transform them into something else.

Select takes a parameter, a *selector* function, which takes the element and should return the "transformed" element.

For example:

```
grimq.from(allEntities(1))
      :select(function(v) return v.id; end)
      :foreach(print)
```

This takes all the entities in level 1, and prints their id: before the execution of the *select*, the collection contains entities; after the execution of the *select*, the collection contains the id of the original entities.

Select has an alternate syntax, where it takes anything other than a function, and returns the table value with the specified key. This works only if the LuaLinq structure contains tables.

For example:

```
grimq.from(allEntities(1))
      :select("id")
      :foreach(print)
```

This, just as the previous example, takes all the entities in level 1, and prints their id.

The *selectMany* method is similar to the *select* method, but a single value in the original collection is transformed into many different values. *selectMany* takes a parameter, a *selector* function, which takes the element and should return an array of the "transformed" elements. There is no alternate syntax for *selectMany*.

For example:

```
grimq.from(allEntities(1))
  :selectMany(function(v) return { v.id, v.name }; end)
  :foreach(print)
```

This takes all the entities in level 1, and prints their id and name: before the execution of the *select*, the collection contains entities; after the execution of the *select*, the collection contains the id and name of the original entities (double the number of items).

FILTERING METHODS – WHERE, WHEREINDEX, TAKE, SKIP

The method *where*, filters the elements in the collection. It takes a *predicate* parameter, which is a function taking an element of the collection and returning *true* if the element should be included or *false* if the element should be filtered out.

For example:

```
grimq.from(allEntities(1))
  :where(function(v) return #v.name > 5; end)
  :select(function(v) return v.name; end)
  :foreach(print)
```

This prints the name of all the entities whose name is longer than 5 characters.

A more useful example:

```
grimq.from(allEntities(1))
  :where(grimq.isMonster)
  :foreach(function(m) m:destroy() end)
```

This destroys any monster in level 1.

Where has an alternate syntax, where it takes anything other than a function, and returns the table value with the specified key. This works only if the LuaLinq structure contains tables.

For example:

```
grimq.from(allEntities(1))
  :where("name", "secret")
  :select("id")
  :foreach(print)
```

This will print the id of all the secrets in level 1.

The method *whereIndex* does the same as *where* but the predicate takes two parameters: the index of the item and the value, instead of just the value.

```
local array = { "ciao", "hello", "au revoir" }
grimq.from(array)
  :whereIndex(function (i, v) return ((i % 2)~=0); end)
  :foreach(print)
```

This prints only the items in the array with an odd index: “ciao” and “au revoir”. There is no alternate syntax for *whereIndex*.

The methods *take* and *skip* take a number and they respectively changes the collection so that it contains the first *n* elements or has all except the first *n* elements.

```
local array = { "ciao", "hello", "au revoir" }
grimq.from(array)
  :take(2)
  :foreach(print)
```

This prints “ciao” and “hello”.

SET METHODS – UNION, EXCEPT, INTERSECTION, DISTINCT

Set methods are methods usually applied to sets – they represent operations performed according to set rules. All these methods take an optional *comparator* which takes two elements and should return true if they are equal.

The *distinct* method returns the collection with duplicates removed.

The *union*, *except* and *intersection* take a second collection as a parameter and return a new collection whose element are the set union, difference and intersection respectively. If the argument passed is not another GrimQ structure, *from* is automatically called.

COMBINATION METHODS – CONCAT, ZIP

Combination methods take two GrimQ structures and combine themselves together in some way. If the argument passed is not another GrimQ structure, *from* is automatically called.

The *concat* method appends element from another GrimQ structure to the current one.

```
local array = { "ciao", "hello", "bonjour" }
local array2 = { "arrivederci", "goodbye", "au revoir" }
grimq.from(array)
  :concat(grimq.from(array2))
  :foreach(print)
```

Prints "ciao", "hello", "bonjour", "arrivederci", "goodbye" and "au revoir".

The *zip* methods, joins pair of elements in the same position together (like a zip in a clothing does). It takes a function getting two elements and returning the resulting one.

```
local array = { "ciao", "hello", "bonjour" }
local array2 = { "arrivederci", "goodbye", "au revoir" }
grimq.from(array)
  :zip(grimq.from(array2), function(a,b) return a .. "/" .. b; end)
  :foreach(print)
```

Prints “ciao/arrivederci”, “hello/goodbye” and “bonjour/au revoir”.

TERMINATING METHODS – GETTING A RESULT

After we queried the collection to our tastes, we are ready to terminate the query and get a result. Several choices are available:

- Picking methods – they pick one element out of the collection: *first, last, random, min, max*
- Inspection methods – they check if the collection satisfies some condition: *any, all, contains, count*
- Aggregation methods – they get some result by automatically aggregating the items together: *sum, average*
- Visitor methods – they apply an operation to all items: *foreach, map, xmap*
- Conversion methods – they convert the collection to an output: *toArray, toDictionary, toIterator*

Note to advanced nitpickers: there is a lot of redundancy in these methods: all of them can technically be implemented through *xmap*. This library is meant to simplify work and improve readability, using *xmap* everywhere doesn't go towards that goal.

PICKING METHODS – FIRST, LAST, RANDOM, MIN, MAX

Picking methods pick one element out of the collection.

The methods *first, last* and *random* do not take any parameter and simply return the first, the last and a random element from the collection.

```
local array = { "ciao", "hello", "bonjour" }  
print(grimq.from(array):random())
```

This prints a random salutation from the list.

The methods *min* and *max* return respectively the minimum and maximum elements of the collection; if the elements are numbers you can call them without parameters, otherwise you need to provide a selector function which takes an element and returns the corresponding value.

```
local hc = grimq.fromAliveChampions()  
:max(function(c) return c:getStat("health"); end)  
  
print("Healthier is ".. hc:getName())
```

This takes the alive champions (through a Grimrock specific initializer) and returns the healthier champion (the print then prints his/her name).

INSPECTION METHODS – ANY, ALL, COUNT, CONTAINS

The method *any* and *all* return true if any, or all, of the collection's elements respect a given condition (expressed through a predicate). If the predicate is not specified, they return true if the collection has at least one element (*any*) or is empty (*all*).

```
print(grimq.from(allEntities(1))
      :any(function(v) return #v.name > 5; end))
```

Prints *true* if at least one entity in level 1 has a name longer than 5 characters. This is the same as:

```
print(grimq.from(allEntities(1))
      :where(function(v) return #v.name > 5; end)
      :any())
```

just shorter.

The method *count* is similar to *any* and *all*, but it returns the number of items satisfying the condition; it may also be a little tad slower because *any* and *all* will return early when they know the result.

The method *contains* checks instead if a collection contains a specific item.

```
local array = { "ciao", "hello", "au revoir" }
print(grimq.from(array):contains("hello"))
```

This prints true as the array contains the word “hello”.

AGGREGATION METHODS – SUM, AVERAGE

Use aggregation methods to aggregate the elements of the collection in a single result, through predefined functions.

The methods *sum* and *average* return respectively the sum and average of elements; if the elements are numbers you can call them without parameters, otherwise you need to provide a selector function which takes an element and returns the corresponding value.

```
local array = { "ciao", "hello", "au revoir" }
local sum = grimq.from(array):sum(function(e) return #e; end)
local avg = grimq.from(array):average(function(e) return #e; end)
print("Sum = ".. sum .. " Avg = " .. avg)
```

This prints “Sum = 18 Avg=6” which are the sum and average of the lengths of the salutation words provided.

VISITOR METHODS – FOREACH, MAP, XMAP

These methods will call some operation for each element of the collection.

The *foreach* method calls the function specified as a parameter for each element of the collection, the function receives the element as a parameter. See most entries in this document for an example.

The *map* method calls the accumulator function specified as a parameter for each element of the collection. Accumulator takes 2 parameters: the element and the previous result of the accumulator itself (firstvalue for the first call) and returns a new result.

```
local values = {1, 2, 3, 4}
```

```
print(grimq.from(values)
      :map(function(n, r) r = r * n; return r; end, 1))
```

This prints 24, that is the product of numbers from 1 to 4 (or 4!).

The *xmap* method – which is very complex to use – calls the accumulator for each element; the accumulator receives the element, the previous result of the accumulator, and the previous “extra” result of the accumulator. On first item it receives *nil* as an element, and firstvalue as “extra”.

As an example of *xmap*, following is the implementation of *max*, implemented using *xmap*:

```
function max(self, selector)
  if (selector == nil) then
    selector = function(n) return n; end
  end
  return self:xmap(function(v, r, l) local res = selector(v); if (l == nil or res >
  l) then return v, res; else return r, l; end; end, nil)
end
```

If you don’t get how *xmap* work, don’t get worried, it’s very advanced and totally optional for getting a good use out of GrimQ.

CONVERSION METHODS - TOARRAY, TODICTIONARY, TOITERATOR

Conversion methods are used to get Lua data structures (arrays, dictionaries and iterators) back from a GrimQ structure.

The *toArray* method is simple, it returns an array of the elements.

```
local array = { "ciao", "hello", "au revoir" }
local output = grimq.from(array)
      :where(function(v) return #v >= 5; end)
      :toArray()
print(output[1])
```

This takes the salutation array, filters out the items whose length is smaller than 5 characters, converts it to an array named output, and then prints the first element – “hello”.

The *toDictionary* method returns a table using key/value pairs returned by a function.

```
local dict = grimq.from(allEntities(1))
      :toDictionary(function(v) return v.id, v.name; end)
print(dict["spider_1"])
```

This builds a table whose key is the id of level-1 entities and the value is the name. Then it prints the name of the entity with id “spider_1”.

Finally the *toIterator* method returns an iterator for items in the collection.

```
local monsters = grimq.from(allEntities(1)):where(grimq.isMonster):toIterator()  
  
for m in monsters do  
  m:destroy()  
end
```

This gets all monsters of level 1 and converts the collection to an iterator. Then in the for loop, it loops over the monsters destroying them.

GRIMROCK SPECIFIC FUNCTIONS

GrimQ contains a number of Grimrock specific functions which can be accessed through the grimq scripting entity. Some of them can even be used outside grimq queries.

These functions are available:

- Enumerations
- Query initializer functions: *fromChampions*, etc
- Predefined predicates: *isMonster*, *isDoor*, etc
- Utility functions: *loadItem*, *saveItem*, *isToorumMode*, etc
- String functions
- Auto objects

ENUMERATIONS

The following enumerations are defined:

```
inventory =
{
    head = 1,
    torso = 2,
    legs = 3,
    feet = 4,
    cloak = 5,
    neck = 6,
    handl = 7,
    handr = 8,
    gauntlets = 9,
    bracers = 10,

    hands = { 7, 8 },
    backpack = { 11, ... 31 },
    armor = { 1, 2, 3, 4, 5, 6, 9 },
    all = { 1, ... 31 },
}

facing =
{
    north = 0,
    east = 1,
    south = 2,
    west = 3
}
```

QUERY INITIALIZER FUNCTIONS

These functions serve to create GrimQ queries from Legend of Grimrock entities.

- *fromChampions* – returns a GrimQ collection with all the champions in the party
- *fromAliveChampions* – returns a GrimQ collection with all the alive and enabled champions in the party
- *fromChampionInventory(champion, recurseIntoContainers, [inventorySlots], [includeMouse])* – returns a GrimQ collection with all the items in the inventory of the given *champion*. If

recurseIntoContainers is true, containers like sack will be opened and the content returned. The optional parameter *inventorySlots* is a table of inventory slots to consider; if not specified or nil, it will check all slots. If *includeMouse* is true, the mouse cursor is also inspected.

- *fromPartyInventory*(*recurseIntoContainers*, [*inventorySlots*] , [*includeMouse*]) – similar to the *fromChampionInventory* method, but this checks the entire party.
- *fromChampionInventoryEx*(*champion*, *recurseIntoContainers*, [*inventorySlots*], [*includeMouse*]) – same as *fromChampionInventory* but instead of returning items, objects are returned with these properties:
 - *slot* : the inventory slot of the item or -1
 - *item*: the item itself
 - *champion*: the champion holding the item, or nil
 - *container*: the container containing the item, or nil
 - *ismouse*: true if it was on the mouse cursor
 - *destroy()* : method which can be called to automatically destroy the item using the appropriate technique
 - *replace(itemname, [itemid])*: method which can be called to automatically replace the item with a new one using the appropriate technique
- *fromPartyInventoryEx* - similar to the *fromChampionInventoryEx* method, but this checks the entire party.
- *fromAllEntitiesInWorld* – returns all entities in the dungeon
- *fromEntitiesInArea*(*level*, *x1*, *y1*, *x2*, *y2*, [*skipx*], [*skipy*]) – returns all entities in a rectangle specified by *x1*, *y1*, *x2* and *y2*. If *skipx* and *skipy* are specified, that specific tile is skipped.
- *fromEntitiesAround*(*level*, *x*, *y*, *radius*, *includecenter*) – returns all the entities in the area surrounding a center, optionally including it
- *fromEntitiesForward*(*level*, *x*, *y*, *facing*, *distance*, *includeorigin*) – returns all the entities in a given direction from a point

PREDEFINED PREDICATES

These functions can be used as they are as *where* predicates in functions. They can also be used in independent code, outside of queries.

Example of predicate use:

```
grimq.from(allEntities(1))
  :where(grimq.isMonster)
  :foreach(function(m) m:destroy() end)
```

This destroys any monster in level 1.

Example of independent use:

```
foreach(m in allEntities(1))
  if (grimq.isMonster(m)) then
    m:destroy()
  end
end
```

List of available predicates:

- `isMonster(entity)`
- `isItem(entity)`
- `isAlcoveOrAltar(entity)`
- `isDoor(entity)`
- `isLever()`
- `isLock(entity)`
- `isPit(entity)`
- `isScript(entity)`
- `isSpawner(entity)`
- `isPressurePlate(entity)`
- `isTeleport(entity)`
- `isTimer(entity)`
- `isTorchHolder(entity)`
- `isWallText(entity)`
- `match(attribute, pattern)` – checks if the string attribute matches a pattern using Lua pattern syntax (see <http://lua-users.org/wiki/PatternsTutorial> and <http://www.lua.org/pil/20.2.html>). For example `match("name", "^gem_")` matches all entities whose name starts with `"^gem_"`.
- `has(attribute, value)` – checks if an attribute has a specific value; for example `has(id, "myid")` matches the entity whose id is `"myid"`.

UTILITY FUNCTIONS

These functions are not specific to queries, but can be useful to duplicate items, store a copy and re-spawning it later, stealing the party inventory, etc.

- `saveItem(item)` – returns a table (compatible with save games) which completely describes the item
- `loadItem(itemTable, [level, x, y, facing, [id]])` – loads an item from *itemTable* which is a table returned by the `saveItem` function. The *level*, *x*, *y* and *facing* parameters are optional (though if one of them is specified, all of them must be). In case they are specified, the item is spawned at the given location; if they aren't a simple call to spawn is called and the item returned so that it could be added to an inventory, a monster or to the mouse cursor.
- `copyItem(item)` – creates a copy of an item
- `moveFromFloorToContainer(container, item)` – moves an item from the floor to the specified container
- `moveItemsFromTileToAlcove(alcove)` – moves all items in the same tile of the specified alcove, into the alcove itself
- `isToorumMode()` – returns true the game was started with Toorum
- `dezombifyParty()` – hackish function to replace "zombi" champions with weak champions when the game is started in Toorum mode
- `reverseFacing(facing)` – returns the opposite direction
- `getChampionFromOrdinal(ord)` – returns the champion with the given ordinal
- `directionFromPos(fromx, fromy, tox, toy)` - returns a facing value given starting and end positions
- `directionFromDelta(dx, dy)` - returns a direction given the differences in x and y (the opposite of `getForward`)
- `destroy(entity)` - can be called on any item and most entities and automatically destroys the entity in the best way, without concerns about where the entity is or what the entity is

- `replace(entity, entityToSpawn, desiredId)` - can be called on any item and most entities and automatically replace the entity with another in the best way, without concerns about where the entity is or what the entity is
- `find(id)` - equivalent of `findEntity`, but works also for items in inventory or mouse cursor
- `gameover()` - kills the party (equivalent to `destroy(party)`)
- `isContainerOrAlcove(entity)` - returns true if entity is either a container or an alcove/altar

STRING FUNCTIONS

These functions allow for easier operation on strings:

- `strformat(string, ...)` – transform the string by replacing tokens starting with \$ with an appropriate substitution text:
 - -- \$1.. \$9 -> are replaced with the optional parameters which can be passed to the function
 - -- \$champ1..\$champ4 -> are replaced with name of champion of the appropriate slot
 - -- \$CHAMP1..\$CHAMP4 -> are replaced with name of champion of the appropriate ordinal
 - -- \$rchamp -> a random champion
 - -- \$RCHAMP -> a random champion which is alive and enabled
- `strstarts(string, start)` – returns true if string starts with the “start” string
- `strends(string, end)` – returns true if string starts with the “end” string
- `strmatch(string, pattern)` – returns true if the string matches the Lua pattern specified (see <http://lua-users.org/wiki/PatternsTutorial> and <http://www.lua.org/pil/20.2.html>)

AUTO OBJECTS

Automatic objects are provided to simplify dungeon creation:

- Any *secret* whose **id** starts with “auto_secret” is activated automatically when the party steps on it. If `AUTO_ALL_SECRETS` is true, all secrets are automatically activated when the party steps on them.
- An “auto_printer” object is provided, when the party steps over it its content are hudPrint-ed. The text is passed to `strformat` before displaying, so \$ tokens can be included.
- Any torch holder whose **name** starts with “auto_” is given a torch
- Any alcove whose **name** starts with “auto_” is filled with all the items found in the same tile
- Any scripting entity containing a method named “autoexec” have that method called at startup. This happens after all the scripting entities have been loaded (and thus is better than code outside of any entity).

These objects are defined (where needed) in a “grimq_objects.lua” file. At the top of the file, there is a “__itemstoautomate” table containing all items which will be cloned in an automated version. Feel free to change at your pleasure.

If the JKos framework is integrated (see its section), it’s possible to define hooks in dynamic yet automatic way, by including them in an *autohook* table. For example:

```
autohook =
{
    party =
    {
        onPickUpItem = function(self, item)
```



```

        print("onPickUpItem")
    end,

    onMove = function(self, direction)
        print("onMove")
    end,

    onTurn = function(self, direction)
        print("onTurn")
    end,
}

```

defines three hooks for the party. At startup, any scripting entity containing an *autohook* table will be automatically added to hooks through the JKos framework. The id under which these hooks are registered is *<entity-name>_<target>_<hook>*.

CONFIGURATION

At the top of the GrimQ script you can find some variables for configuration:

- MAXLEVEL – This should be always set to the number of the maximum level in the dungeon
- AUTO_ALL_SECRETS – If this is true, all secrets are treated as auto-secrets (see auto objects, above)
- USE_JKOS_FRAMEWORK – If this is true, GrimQ will take care of JKos framework initialization and will offer enhanced integration
- DEBUG_MODE – If this is true, some debug information may be printed on the console

JKOS FRAMEWORK INTEGRATION

GrimQ is capable of offering some enhanced feature if JKos framework is used in the dungeon.

Features:

- Simplified JKos framework setup (no need of defining the logfw_init entity in the dungeon)
- Automatic hooks (see autohooks) can be defined in scripting entities
- Initialization order features – the *autoexec* method of scripting entities will be called after both GrimQ and JKos framework are initialized and ready
- Non-delayed initialization – the initialization of JKos framework does not use a timer at this moment

To setup:

- Setup GrimQ and set its USE_JKOS_FRAMEWORK option to true
- Follow the setup instructions at <https://sites.google.com/site/jkosgrimrock2/home> but do NOT create the logfw_init entity
- Put any code adding hooks etc. in an *autoexec* method in any scripting entity you want.

KNOWN ISSUES

- Scrolls with an image are not saved and restored correctly, as there is no *setScrollImage* method on items
- Containers are saved and restored correctly, but the order/position of items inside them might change
- The replace and destroy function of objects returned by *fromChampionInventoryEx* and *fromPartyInventoryEx* fail if the object is in a subcontainer. This is due to missing methods in the engine
- It's necessary to update the MAXLEVEL variable everytime the number of level changes
- All the above problems should be solvable when the new version of the game is out

COMMON QUERY EXAMPLES

These are common used queries which can be used as reference. No comment is provided, just code. Differently from the code of the rest of this document – which was tested – this code is not tested.

```
-- Check if a container has any item. Works with sacks, altars, alcoves, etc.
function checkContainerHasAnyItem(cont)
    return grimq.from(cont:containedItems()):any()
end

-- Check if a container has a specific item. Works with sacks, altars, alcoves, etc.
function checkContainerHasItem(cont, itemName)
    return grimq.from(cont:containedItems()):any(function(v) return v.name == itemName; end)
end

-- Count the entities in a level of a given type
function countEntitiesOfTypesInLevel(entityNames, level)
    return grimq.from(allEntities(level)):count(function(v) return v.name == itemName; end)
end

-- Opens all doors of a level
function openAllDoorsOfLevel(level)
    grimq.from(allEntities(level))
        :where(function(v) return grimq.isDoor(v); end)
        :foreach(function(d) d:open(); end)
end
```

CREDITS, LICENSE AND DISCLAIMERS

This project is developed by Marco Mastropaolo (Xanathar) as a personal project and is in no way affiliated with Almost Human.

You can use this script library in any Legend of Grimrock dungeon you want; credits are appreciated though not necessary.

You cannot use this library outside Legend of Grimrock scripting. If you want to use this code in a Lua project outside Grimrock, please refer to the files and license included at <http://lualinq.sourceforge.net/>

Thanks goes to:

- My wife&family for the time I can dedicate to these side projects.
- Almost Human for the great game Grimrock is
- John Wordsworth from which I copied part of the disclaimer :)
- Lark, for showing me how to understand which type of entity is of a given type
- Grimwold – he set an example script-wise
- Komag for testing of the loadItem/saveItem/copyItem set of functions