ISXEQ2 - Lavishscript Query Strings

Description

A Lavishscript query string is a math formula where all variables are relative to a specific object, and results in a boolean value -- the query is either true, or it is false. That is to say, it efficiently determines whether one object meets various conditions.   Query math can compare and manipulate text, decimals, or integers (note that bool counts as an integer for this purpose) with standard math operators. Text comparisons are not case sensitive.

Expression/Operator Notes

- [redacted]
    - a !~ NULL is true if ${a(exists)}
    - a =~ NULL is true if !${a(exists)}

- [redacted]
    - =- Case Insensitive Substring Search   (*Example*:   Name =- "Boobies" is true if Name.Find[Boobies])
    - = Weak compare, value only
    - == Strong compare, value + type
    - =~ Strong compare, value + type (same as ==)
    - !~ Strong compare, NOT value or type

Technical Notes

- [redacted]

Example 1

```
function main()
{
    variable index:actor Actors
    variable iterator ActorIterator

    EQ2:QueryActors[Actors, Type  =- "NPC" && Distance <= 20 && Level > 90 && (IsHeroic == 1 || IsEpic == 1)]
    Actors:GetIterator[ActorIterator]

    if ${ActorIterator:First(exists)}
    {
        do
        {
            echo "${ActorIterator.Value.Name}"
        }
        while ${ActorIterator:Next(exists)}
    }
}
```

Example 2

```
function main()
{
    variable index:item Items
    variable iterator ItemIterator
    variable int Counter = 1

    Me:QueryInventory[Items, Name =- "Incandescent" && Location == "Bank" && Quantity > 1]
    Items:GetIterator[ItemIterator]

    if ${ItemIterator:First(exists)}
    {
        do
        {
            echo "${Counter}. ${ItemIterator.Value.Name} [${ItemIterator.Value.Location}] [Level: ${ItemIterator.Value.Quantity}]"

            Counter:Inc
        }
        while ${ItemIterator:Next(exists)}
    }
}
```

Example 3

```
function main()
{
    variable index:effect MyEffects
    variable iterator MyEffectsIterator
    variable int Counter = 1

    ;;;;;;;;;;;;;;
    ;; In this sample script, we will be using the .ToExamineInfo MEMBER of all the character's
    ;; effects, and the current target's effects.   The routines below will properly handle waiting
    ;; until the effectinfo object is cached and available; however, since there is only ever a maximum
    ;; of 30 beneficial, 30 detrimental, and 30 actor effects available to the client at any given time,
    ;; ISXEQ2 provides a useful method to start the process of getting the details cached for all effects
    ;; and buffs at once.   This should save time in the iteration routines below.
    ;;;;;;;;;;;;;;
    Me:RequestEffectsInfo
    if (${Target(exists)})
        Target:RequestEffectsInfo

    ;;;;;;;;;;;;;;
    ;; Beneficial Effects
    ;;;;;;;;
    ;;;;
    ;;;; The following routine illustrates how to iterate through beneficial effects.  To return a single
    ;;;; effect, you can do so by using the "Query" argument along with a lavishsoft Query String.  For example
    ;;;; to check if the character has a beneficial effect with a MainIconID of 234, simply use:
    ;;;; "if ${Me.Effect[Query, Type == "Beneficial" && MainIconID == "234"](Exists)}"
    ;;;;;;;;

    echo "Beneficial Effects:"
    Me:QueryEffects[MyEffects, Type == "Beneficial"]
    MyEffects:GetIterator[MyEffectsIterator]

    if ${MyEffectsIterator:First(exists)}
    {
        do
        {
            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
            ;; This routine is echoing the effect's "Name", so we must ensure that the effectinfo
            ;; datatype is available.
            if (!${MyEffectsIterator.Value.IsEffectInfoAvailable})
            {
                ;; When you check to see if "IsEffectInfoAvailable", ISXEQ2 checks to see if it's already
                ;; cached (and immediately returns true if so).  Otherwise, it spawns a new thread
                ;; to request the details from the server.
                do
                {
```

```
                waitframe
                ;; It is OK to use waitframe here because the "IsEffectInfoAvailable" will simple return
                ;; FALSE while the details acquisition thread is still running.   In other words, it
                ;; will not spam the server, or anything like that.
            }
            while (!${MyEffectsIterator.Value.IsEffectInfoAvailable})
        }
        ;;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;; At this point, the "ToEffectInfo" MEMBER of this object will be immediately available.  It should
        ;; remain available until the cache is cleared/reset (which is not very often.)

        echo "- ${Counter}. ${MyEffectsIterator.Value.ToEffectInfo.Name} (ID: ${MyEffectsIterator.Value.ID}, MainIconID: ${MyEffectsIterator.Value.MainIconID}, BackDropIconID: ${MyEffectsIterator.Value.BackDropIconID})"
        Counter:Inc
    }
    while ${MyEffectsIterator:Next(exists)}
}
else
    echo "NONE"
echo "--------------------"
Counter:Set[1]
;;
;;;;;;;;;;;;;;

;;;;;;;;;;;;;;
;; Detrimental Effects
;;;;;;;;
;;;;
;;;; The following routine illustrates how to iterate through detrimental effects.  To return a single
;;;; effect, you can do so by using the "Query" argument along with a lavishsoft Query String.  For example
;;;; to check if the character has a detrimental effect with a MainIconID of 234, simply use:
;;;; "if ${Me.Effect[Query, Type == "Detrimental" && MainIconID == "234"](Exists)}"
;;;;;;;;

echo "Detrimental Effects:"
Me:QueryEffects[MyEffects, Type == "Detrimental"]
MyEffects:GetIterator[MyEffectsIterator]

if ${MyEffectsIterator:First(exists)}
{
    do
    {
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;; This routine is echoing the effect's "Name", so we must ensure that the effectinfo
        ;; datatype is available.
        if (!${MyEffectsIterator.Value.IsEffectInfoAvailable})
        {
            ;; When you check to see if "IsEffectInfoAvailable", ISXEQ2 checks to see if it's already
            ;; cached (and immediately returns true if so).  Otherwise, it spawns a new thread
            ;; to request the details from the server.
            do
            {
                waitframe
                ;; It is OK to use waitframe here because the "IsEffectInfoAvailable" will simple return
                ;; FALSE while the details acquisition thread is still running.   In other words, it
                ;; will not spam the server, or anything like that.
            }
            while (!${MyEffectsIterator.Value.IsEffectInfoAvailable})
        }
        ;;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;; At this point, the "ToEffectInfo" MEMBER of this object will be immediately available.  It should
        ;; remain available until the cache is cleared/reset (which is not very often.)

        echo "- ${Counter}. ${MyEffectsIterator.Value.ToEffectInfo.Name} (ID: ${MyEffectsIterator.Value.ID}, MainIconID: ${MyEffectsIterator.Value.MainIconID}, BackDropIconID: ${MyEffectsIterator.Value.BackDropIconID})"
        Counter:Inc
    }
    while ${MyEffectsIterator:Next(exists)}
}
else
    echo "NONE"
echo "--------------------"
Counter:Set[1]
;;
;;;;;;;;;;;;;;


;;;;;;;;;;;;;;
;; Maintained Effects (Buffs)
;;;;;;;;
;;;;
;;;; The following routine illustrates how to iterate through maintained effects (buffs).  To return a single
;;;; buff, you can do so by searching for a particular buff name or the index of the buff in the array (i.e.,
;;;; ${Me.Maintained["NAME"]} or ${Me.Maintained[#]})
;;;;;;;;

echo "Maintained Effects (Buffs):"
variable string Duration
variable string EffectTarget
variable int CountMaintained

CountMaintained:Set[${Me.CountMaintained}]

if (${CountMaintained} > 0)
 {
    do
    {
        if (${Me.Maintained[${Counter}].Duration.Equal[-1]})
            Duration:Set["N/A"]
        else
            Duration:Set[${Me.Maintained[${Counter}].Duration.Precision[2]}]

        if (${Me.Maintained[${Counter}].Target(exists)})
            EffectTarget:Set[${Me.Maintained[${Counter}].Target.Name}]
        else
            EffectTarget:Set["N/A"]

        echo "- ${Counter}. ${Me.Maintained[${Counter}].Name} (Duration: ${Duration}, Target: ${EffectTarget})"

    }
    while (${Counter:Inc} <= ${CountMaintained})
 }
else
    echo "NONE"
echo "--------------------"
Counter:Set[1]
;;
;;;;;;;;;;;;;;

;;;;;;;;;;;;;;
;; ActorEffects (Current Target)
;;;;;;;;
;;;;
;;;; The following routine illustrates how to iterate through actor effects.  To return a single
;;;; actor effect, you can do so by using the "Query" argument along with a lavishsoft Query
;;;; string.   For example, to check if the current target has the effect with an ID of 123456, one could
;;;; check "if ${Target.Effect[Query,ID = "123456"](Exists)}"
;;;;;;;;

echo "ActorEffects on the current target:"
variable int NumActorEffects = 0

if (${Target(exists)})
{
    NumActorEffects:Set[${Target.NumEffects}]

    if (${NumActorEffects} > 0)
    {
        do
        {
            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
            ;; This routine is echoing the effect's "Name", so we must ensure that the effectinfo
            ;; datatype is available.
            if (!${Target.Effect[${Counter}].IsEffectInfoAvailable})
            {
                ;; When you check to see if "IsEffectInfoAvailable", ISXEQ2 checks to see if it's already
                ;; cached (and immediately returns true if so).  Otherwise, it spawns a new thread
                ;; to request the details from the server.
                do
                {
                    waitframe
                    ;; It is OK to use waitframe here because the "IsEffectInfoAvailable" will simple return
                    ;; FALSE while the details acquisition thread is still running.   In other words, it
                    ;; will not spam the server, or anything like that.
                }
                while (!${Target.Effect[${Counter}].IsEffectInfoAvailable})
            }
            ;;
            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
            ;; At this point, the "ToEffectInfo" MEMBER of this object will be immediately available.  It should
            ;; remain available until the cache is cleared/reset (which is not very often.)

            echo "- ${Counter}. ${Target.Effect[${Counter}].ToEffectInfo.Name} (ID: ${Target.Effect[${Counter}].ID}, MainIconID: ${Target.Effect[${Counter}].MainIconID})"
        }
        while (${Counter:Inc} <= ${NumActorEffects})
    }
    else
        echo "NONE"
}
else
    echo "NO TARGET"
echo "--------------------"
;;
;;;;;;;;;;;;;;
}
```

Example 4

```
function main()
{
    variable index:ability MyAbilities
    variable iterator MyAbilitiesIterator
    variable int Counter = 1
    variable int Timer = 0

    ;;;;;;;;;;;;;;
    ;;;;
    ;;;; The following routine illustrates how to iterate through abilities.  To return a single
    ;;;; ability, you can do so by using the "Query" argument along with a lavishsoft Query String.  For example
    ;;;; to check if the character has an ability with a ID of 234, simply use:
    ;;;; "if ${Me.Ability[Query, ID =- "546331599"](Exists)}"
    ;;;;;;;;

    echo "Abilities (Total ${Me.NumAbilities}):"
    Me:QueryAbilities[MyAbilities, ID =- "546331599" || ID =- "478256501"]
    MyAbilities:GetIterator[MyAbilitiesIterator]

    if ${MyAbilitiesIterator:First(exists)}
    {
        do
        {
            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
            ;; This routine is echoing the ability's "Name", so we must ensure that the abilityinfo
            ;; datatype is available.
            if (!${MyAbilitiesIterator.Value.IsAbilityInfoAvailable})
            {
                ;; When you check to see if "IsAbilityInfoAvailable", ISXEQ2 checks to see if it's already
                ;; cached (and immediately returns true if so).  Otherwise, it spawns a new thread
                ;; to request the details from the server.
                do
                {
                    wait 2
                    ;; It is OK to use waitframe here because the "IsAbilityInfoAvailable" will simple return
                    ;; FALSE while the details acquisition thread is still running.   In other words, it
                    ;; will not spam the server, or anything like that.
                }
                while (!${MyAbilitiesIterator.Value.IsAbilityInfoAvailable} && ${Timer:Inc} < 1500)
            }
            ;;
            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
            ;; At this point, the "ToAbilityInfo" MEMBER of this object will be immediately available.  It should
            ;; remain available until the cache is cleared/reset (which is not very often.)

            echo "- ${Counter}. ${MyAbilitiesIterator.Value.ToAbilityInfo.Name} (ID: ${MyAbilitiesIterator.Value.ID}, IsReady: ${MyAbilitiesIterator.Value.IsReady})"
            Counter:Inc
            Timer:Set[0]
        }
        while ${MyAbilitiesIterator:Next(exists)}
    }
    else
        echo "NONE"
    echo "---------------------"
    ;;
    ;;;;;;;;;;;;;;
}
```