MEIG SYSTEM

Introduzione

mEiG system nasce per fornire agli artisti multimediali uno strumento di sviluppo, gestione e produzione di progetti che coinvolgono strumenti interattivi (audio, video, physical computing). Con il *mEiG* è possibile tracciare tutte le fasi del lavoro, dalla concezione alla messa in esecuzione. Si tratta di un sistema di controllo integrato che permette di rappresentare e modellare divere entità (*device*) e di definirne il comportamento nel tempo grazie a specifiche configurazioni statiche (*preset*) e dinamiche (*automation*). Tali comportamenti e impostazioni possono essere aggregati per l'esecuzione in macrostrutture adatte alla messa in riproduzione (*marker*/*cue*).

Il luogo in cui le entità vengono posizionate è il *roll* che appare simile a una convenzionale timeline, ma che in realtà permette di gestire le componenti in modo non-lineare: i raggruppamenti (sia a livello di *cue* che di *event* - aggregatori di *automations*) possono essere *estrapolati* dal contesto globale e gestite come universi temporali a sé. In questo modo la timeline è piuttosto un contenitore di moduli identificati da *markers*, che possono essere diversamente arrangiati per l'esecuzione (*cue list*).

Il *mEiG* è un sistema distribuito e condiviso, sia in editing che playing-side. Utenti connessi in rete possono visualizzare, apportare modifiche e salvare uno stesso progetto in tempo reale, mentre all'atto dell'esecuzione tutti i messaggi in uscita dal sistema sono formattati come messaggi OSC (Open Sound Control) e instradati verso macchine diverse sulla base di una topologia definita in fase di configurazione (*network graph*).

L'attuale implementazione del sistema, che coinvolge tecnologie diverse basate su un'infrastruttura client-server, è ospite di una patch di **Max 8** (*Cycling74*), da cui è possibile gestire le pagine del front-end e installare il server alla pressione di un pulsante.

Tecnologia

Il sistema *mEiG* nasce esplicitamente come progetto di sviluppo dell'originale MEEG, dal quale mutua alcuni principi e funzionalità, ma da cui si discosta per altri aspetti, talora cruciali. Il sistema MEEG è basato su un'architettura web client-server, dove il backend è costruito su un server PHP e un database sql (MySql), mentre il frontend è basato su un'interfaccia html con il protocollo http come base di comunicazione.
Il sistema mEiG usa invece un ecosistema tecnologico che garantisce portabilità e leggerezza. E' mantenuta l'architettura client-server, ma sussiste un'accentramento delle componenti. Il software Max, che dalla versione 8 integra un server nodeJS, si pone al centro dell'ecosistema. Il server nodeJS (che sostituisce il PHP) è implementato all'interno dell'oggetto **node.script** e si installa con un click direttamente dall'interno di Max. Il client, costituito da una serie di componenti javascript/html, viene caricato *pezzo per pezzo* in oggetti **jweb**, all'interno dello stesso Max. Infine il database sql viene sostituito da una serie di dizionari (Max dictionaries) che, sostanzialmente, sono costituiti da strutture dati in formato JSON.

SQL/noSQL

La scelta di non utilizzare un *db engine* come SQL nasce dall'esigenza di snellire la portata dell'applicazione e di velocizzare le operazioni di interrogazione e di scambio dei dati. La struttura relazionale è sembrata eccessiva rispetto alla mole di dati richiesti da un singolo progetto. SQL tende ad essere molto efficiente su una grande mole di dati, ma perde la sua efficacia in presenza di un rapporto basso fra numero di righe e numero di colonne. Per arrivare alla soluzione adottata sono stati eseguiti alcuni passaggi concettuali (e pratici), condizionati dal punto di osservazione del problema:

- *Portabilità*: sqlite
    - PRO: sembrava sufficiente utilizzare un'istanza di *sqlite* integrata nel server. In questo modo si evita il problema di una gestione separata del database (con tutto quello che ne consegue: accesso, utenti, installazione).
    - CONTRO: difficile scalabilità e inefficientza delle query. L'impossibilità di scrivere *stored procedures* all'interno del *db engine* aumenta considerevolmente le quantità di *join* lato query e rallenta di conseguenza la velocità di risposta alle interrogazioni.
- *Efficienza*: graph database
    - PRO: l'attenzione si è rivolta a una struttura di conservazione dei dati più flessibile e veloce: il database a grafo. Una struttura dati che rappresenti l'informazione con nodi e archi è sembrata garantire maggiore velocità di interrogazione in grafi come quelli di *mEiG*. Le *join* vengono sostituite da meccanismi di attraversamento (*traversing*), evitando lentissimi prodotti cartesiani fra le righe delle teballe relazionali.
    - CONTRO: la maggior parte dei database a grafo (OrientDB, Neo4J, etc...) non sono integrati, ma sussitono come *engine* standalone, quindi vanno installati e configurati autonomamente.
- *Soluzione adottata*: JSON *dictionaries*. Il grafo è sembrata comunque la struttura dati più vicina alla rappresentazione naturale dei dati di *mEiG*. In particolare l'*albero* (che di fatto è una forma di grafo orientato) sembrava sufficiente a rappresentare i dati e le loro relazioni. L'ultimo passo quindi è stato quello di implementare degli alberi tramite il formato JSON. Grazie alla possibilità di usare **array associativi** come tipo di dato primitivo, intrinseca nel JSON, rappresentare un albero tramite JSON è stato immediato. Il JSON inoltre, oltre a garantire arbitrari livelli di innesto, è il formato di interscambio nel web per eccellenza, e in prospettiva, sembra la struttura più adeguata per rendere il *mEiG* una piattaforma distribuita.

html/javascript

Max integra al suo interno un motore javascript basato sulla versione ES5 (ormai vecchia) e impedisce l'importazione di packages esterni. Per questo motivo, piuttosto che utilizzare il javascript integrato, si è preferito scrivere dei moduli esterni in html/js/css che vengono importati in Max tramite l'oggetto **jweb**. Questo scelta ha garantito la possibilità di importare alcuni packages esterni sia per la gestione grafica delle interfacce (jquery e bootstrap), sia per la manipolazione delle strutture dati (jqtree). Inoltre l'html con javascript permette di implementare un'architettura non sono scalabile, ma traslabile su altre piattaforme che non siano Max. Allo stato attuale del progetto in effetti il solo oggetto che implementa la timeline e le entità da temporizzare (**roll** e **slot** rispettivamente) non sono agnostici rispetto alla piattaforma. Tutte le strutture dati e le altre entità sono basate di linguaggi e tecnologie standard del web.

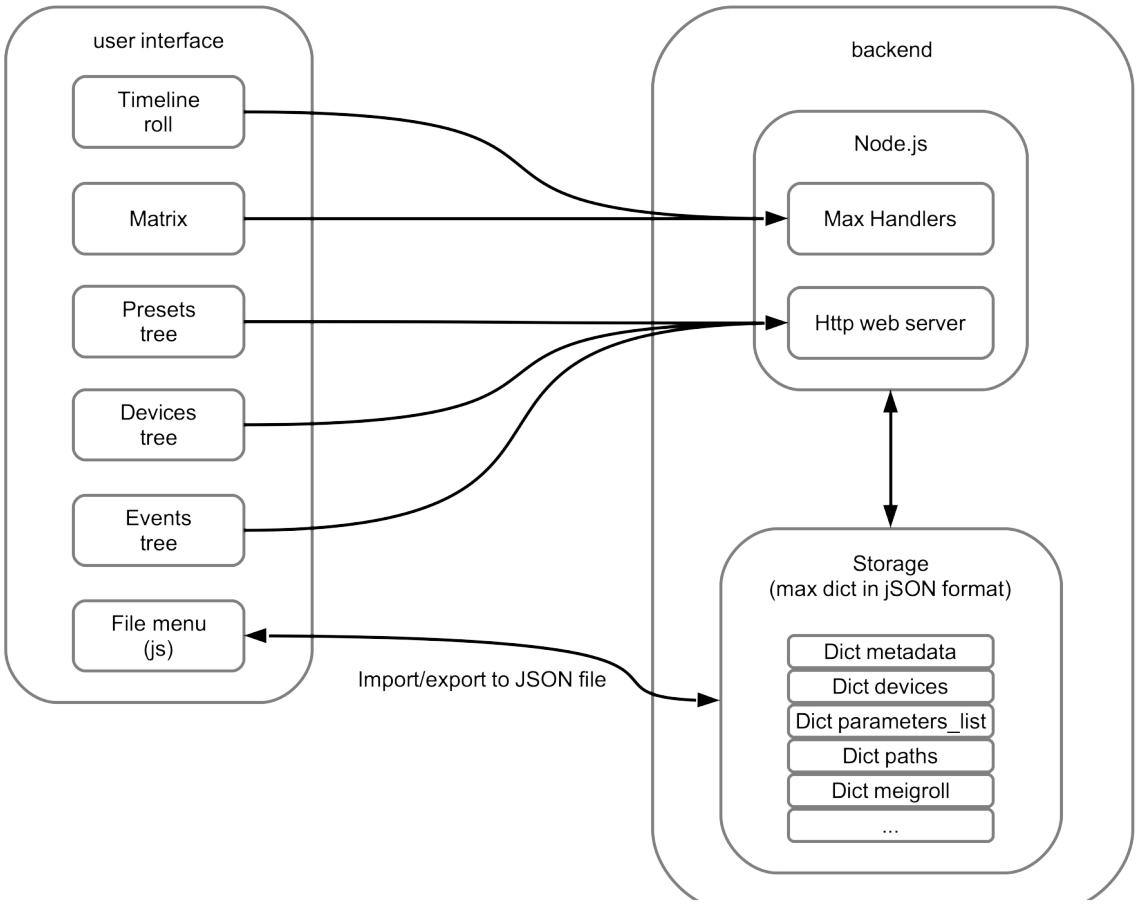
bach

strutture dati: *llll* conversione *llll* to *JSON* ... ...

Infrastruttura

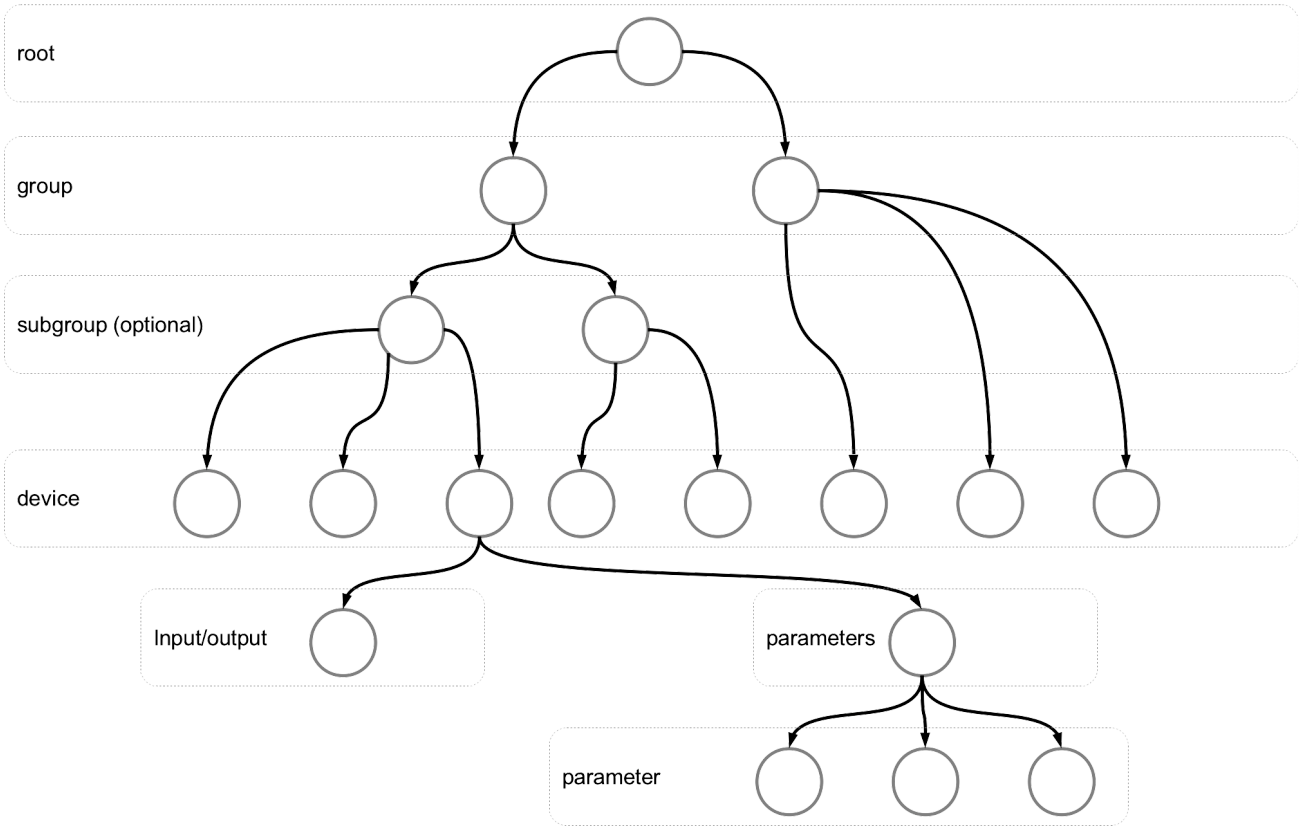
L'app ha una struttura ibrida: una parte delle componenti comunicano tramite il protocollo HTTP, trasmettendo dati in formato JSON; un'altra parte accede direttamente al backend tramite dei manipolatori *ad hoc* (Max handlers). Al centro dell'infrastruttura si trova un server implementato in node.js, che raccoglie le chiamate sia dai client http che dalle componenti ad accesso diretto. I dati vengono mantenuti localmente in oggetti Max di tipo **dict**. Si tratta *de facto* di strutture dati in formato JSON che possono essere aggregate al momento dell'esportazione su file, anch'essi in formato JSON.

In figura si può osservare la struttura generale dell'applicazione.



Per quanto riguarda la persistenza dei dati, è sembrato opportuno utilizzare una struttura dati ad albero, la cui rappresentazione più naturale fosse lo stesso JSON.

I dati vengono rappresentati in strutture ad albero con radice. Ogni nodo deriva da uno e un solo genitore, mentre può avere nessuno, uno o più figli. La figura seguente mostra la struttura (parziale) dell'albero dei *devices*.



Ogni nodo è identificato da un *id* univoco nel sistema, quindi può essere modificato senza dover essere duplicato e nuovamente referenziato. L'albero viene rappresentato all'interno dei *dict* di Max in formato JSON: Ogni nodo ha almeno un *id* (l'unico elemento non modificabile), una *label* e una chiave *type*. Se non si tratta di una *foglia* ha la chiave **children**, che contiene un array dei nodi figli:

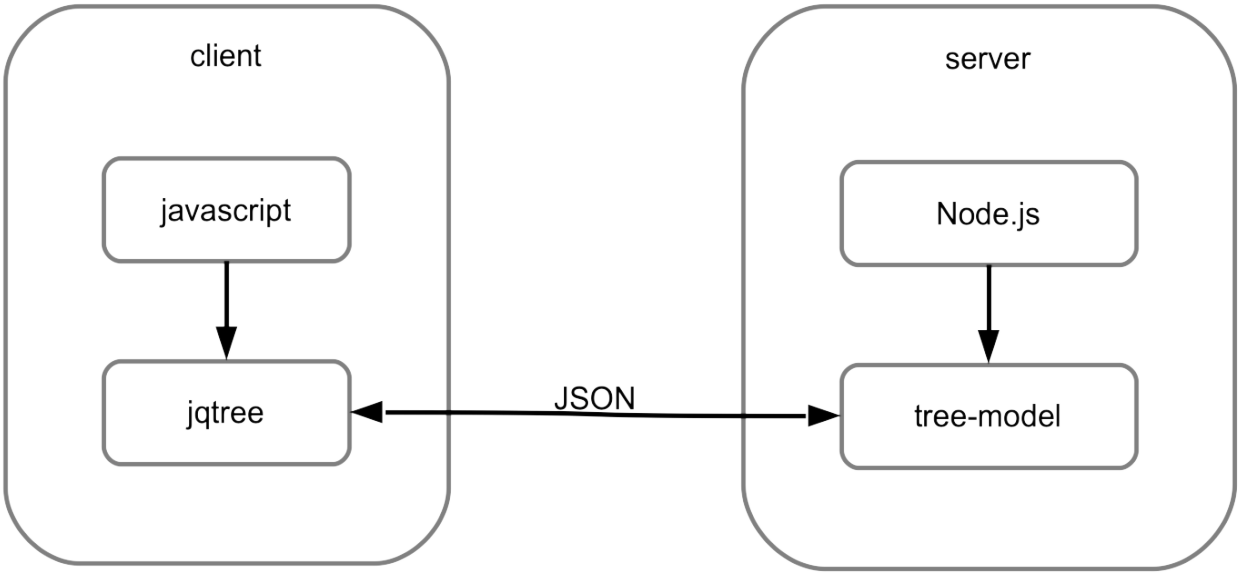
```
{
    "label": "main",
    "id": "1234abcd5678efgh",
    "type": "devices",
    "children": [
        {
            "label": "group_1",
            "id":"id": "0000abcd5678asdf",
            "type": "group",
            "children": [
                {"...":"..."}
            ]
        },
        {
            "label": "group_2",
            "id": "2222sdas12312yrue333",
            "type": "group",
            "children": [
                {"...":"..."}
            ]
        }
```

```
    ]
}
```

La struttura ricorsiva di ogni nodo permette di rendere alcuni *types* innestabili a livelli diversi. L'entità di tipo *group* ad esempio, può avere come *children* altre entità sinonime:

```
{
    "label": "group_1",
    "id":"id": "0000abcd5678asdf",
    "type": "group",
    "children": [
        {
            "label": "subgroup_1",
            "id":"id": "0000abcd5678asdf",
            "type": "group",
            "children": [
                {"...":"..."}
            ]
        }
    ]
}
```

Tali strutture dati vengono gestite sia lato client che lato server con *packages* che permettono di modellizzare le strutture dati ad albero; come si vede in figura il pacchetto javascript **jqtree** si occupa di gestire l'albero lato client, mentre il pachetto **tree-model** gestisce gli alberi lato server.



Graphical User Interfaces

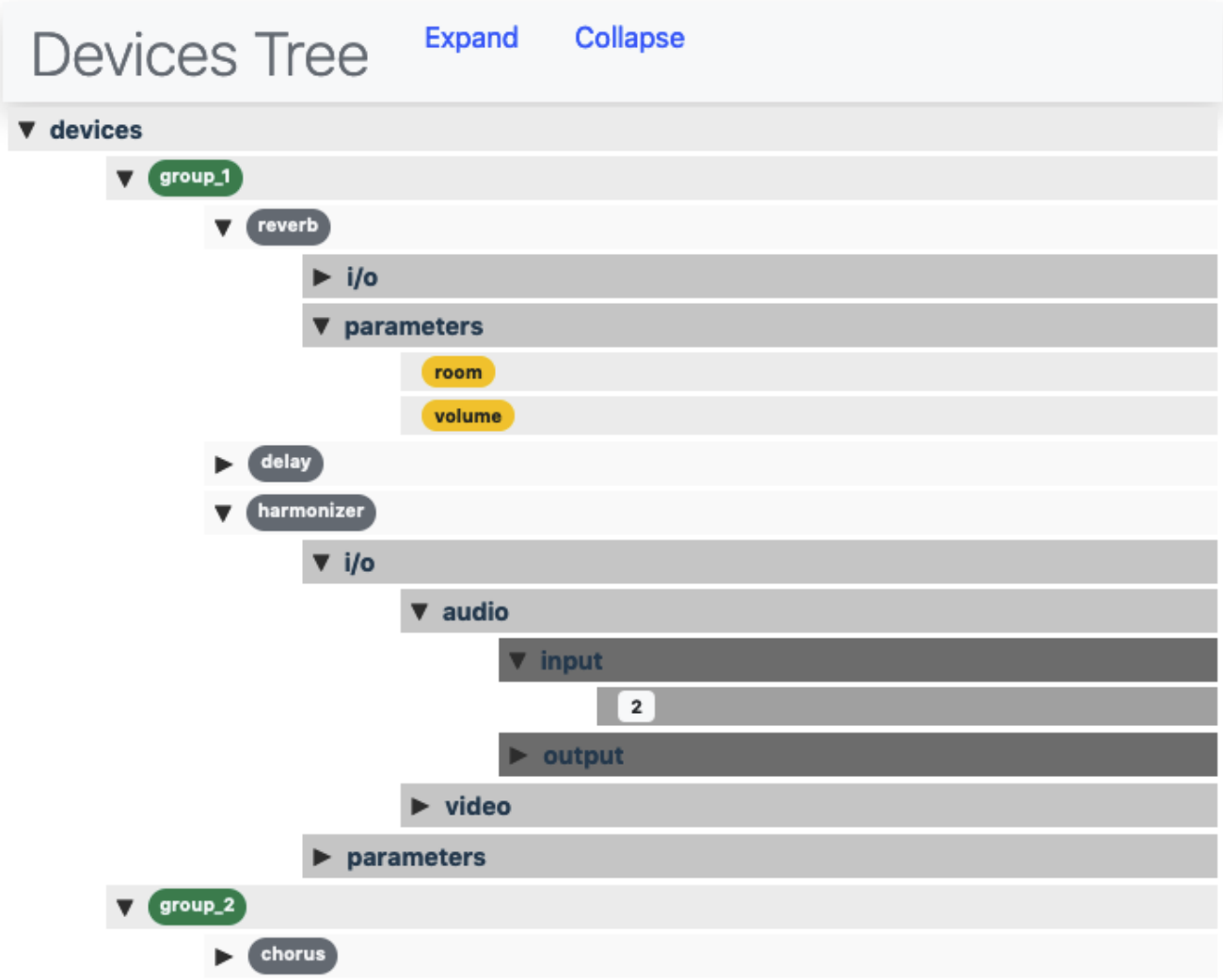
roll

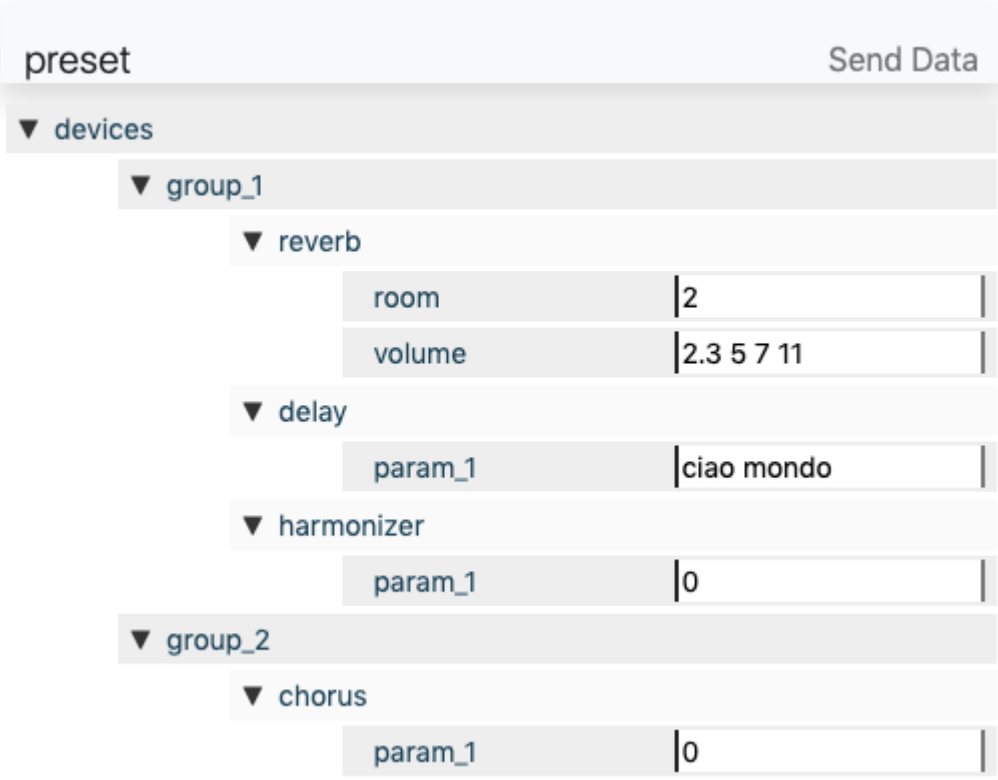
...

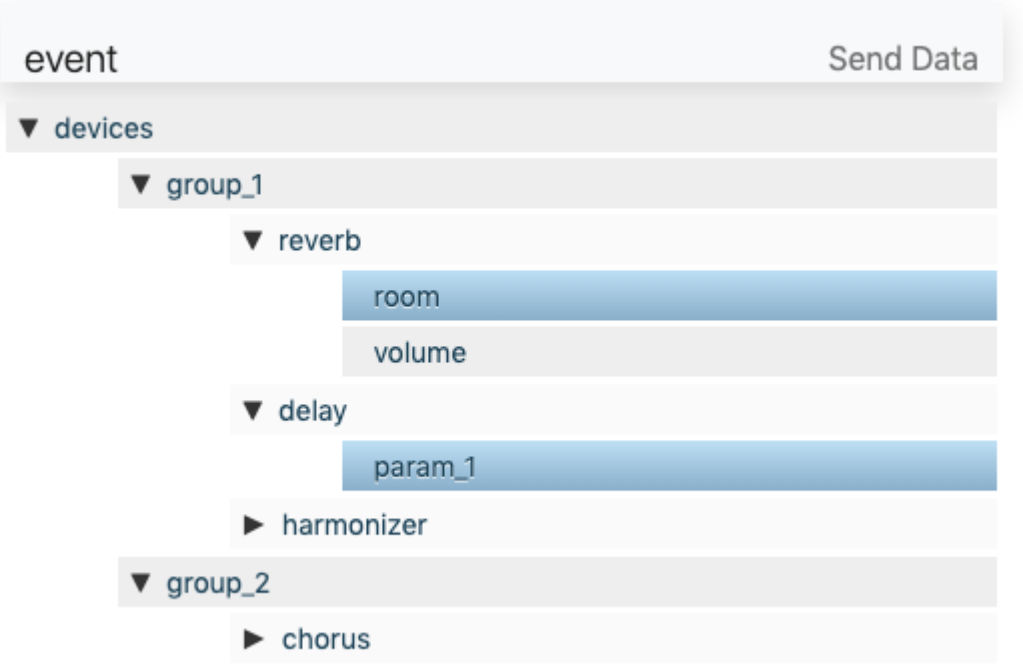
Devices, Preset, Event

Allo stato attuale del progetto, l'entità che rappresenta l'albero dei *Devices* ha una rappresentazione grafica realizzata in html/css/javascript e contenuta all'interno della *patch* di Max nell'oggetto **jweb**. A partire da questa entità vengono creati altri due alberi, *Presets* ed *Events*, con strutture grafiche analoghe. L'albero dei *Devices* viene servito all'utente all'atto dell'apertura di un file esistente o della creazione di un file nuovo e permette di inserire nel sistema nuovi dispositivi. L'entità centrale di questa struttura è il **device** che contiene gli input/output e i parametri, e che può essere a sua volta contenuto in gruppi o sottogruppi. Per aggiungere, rimuovere o rinominare un'entità dall'albero è sufficiente usare i menu contestuali attivabili col destro del mouse.

Dall'albero dei *Devices* derivano le altre due strutture, *Preset* ed *Event*, che permettono, rispettivamente, di impostare i valori di tutti i parametri in un preset e di scegliere quali parametri utilizzare all'interno di un event. Graficamente il *Preset Tree* è composto da una vista ad albero con tutti i parametri provvisti di un *form* in cui inserire il valore (*Number*, *String* o *Array*). L'*Event Tree* invece è un mero elenco di tutti i parametri, con la possiblità di selezionare quelli interessati.


Devices Tree
Expand
Collapse
devices
group_1
reverb
i/o
parameters
room
volume
delay
harmonizer
i/o
audio
input
2
output
video
parameters
group_2
chorus


preset
Send Data
devices
group_1
reverb
room
2
volume
2.3 5 7 11
delay
param_1
ciao mondo
harmonizer
param_1
0
group_2
chorus
param_1
0



TODO

- Implementazione marker (queue)
- Implementazione matrice
- Implementazione di un sistema per "inizializzare" eventi e preset sulla base dell'ultimo evento precedente
- Implementazione di un motore di play (unità di misura del tempo, scalatura della velocità di riproduzione, playback su queue, etc...)
- Formalizzazione di un sistema di messaggistica OSC
- Sistema distribuito (centralizzato o decentralizzato) -> Sistema di condivisione e sincronizzazione dei progetti in tempo reale
- Connessione con interfacce *reali* audio, video, MIDI e physical computing (Arduino, Raspberry, ESP32, Bèla)