**Parser**

**1. Preambolo**

Nell’ambito del problema della permanenza dei dati, abbiamo analizzato prima di tutto la complessità che un sistema di tipo Database dovrebbe rispettare per meglio rappresentare gli oggetti creati e manipolati dal sistema PCReady, come tale complessità potrebbe crescere al progredire delle varie iterazioni del processo software, e una stima del tempo necessario per realizzarlo e farlo evolvere di pari passo con il resto del software.

A conclusione delle nostre ricerche è stato valutato che la priorità assoluta vada all’obbiettivo di mostrare al cliente un software funzionante e dal numero di funzioni in costante aumento, e che quindi la costruzione di un database ben strutturato e articolato – seppur di rilevante importanza – possa tagliare fuori una parte importante del tempo a nostra disposizione fra un’iterazione e la successiva.

Un’ulteriore indagine ci ha portato alla conclusione che sia possibile commissionare il lavoro sul Database a terzi (in subappalto ad un’azienda o assumendo un “database engineer”), in modo tale da permettere al nostro team di sviluppatori di limitare la mole di lavoro allo sviluppo di un’interfaccia fra sistema PCReady e database.

Per lo sviluppo di tale interfaccia, si è presupposto che il Database fornirà un server costruito su approccio REST, e che quindi l’accesso ai dati o il loro salvataggio prenda la forma di una richiesta HTTP contenente un file di testo in formato JSON.

Suddetta interfaccia è quindi ciò a cui ci siamo riferiti nel resto della documentazione con il termine “Parser”, in quanto si occupa di analizzare e fare da tramite fra oggetti in memoria e oggetti testuali, e la sua implementazione è presente nell’omonima classe.

**2. Il database d’esempio**

In assenza dell’esistenza fisica del server collegato al Database, si è deciso – senza particolari restrizioni – di poter operare in maniera perfettamente analoga su uno pseudo-database formato da tre file in formato JSON, i quali possono simulare perfettamente il contenuto di un’eventuale richiesta HTTP al server effettivo. I file sono contenuti nella cartella **“data”**, esterna al source *“src”*.

Tali file sono, in ordine alfabetico: ***catalogo.json*** , ***ordini.json*** , ***utenti.json***.

**Catalogo:** il file *catalogo.json* rappresenta la permanenza in memoria di un oggetto appartenente alla classe Catalogo. In quanto tale, per rappresentare l’elenco dei componenti come una Mappa di Liste, il file rappresenta un singolo oggetto i cui attributi sono i nomi delle varie tipologie di Componente esistenti (“Storage”, “CPU”, etc.) e i valori corrispondenti sono degli array.

Tali array, a loro volta, rappresenteranno le Liste di Componenti di quella specifica categoria, e pertanto contengono oggetti contenenti tutti gli attributi necessari per istanziare un Componente (id, nome, descrizione, prezzo, consumo), più gli attributi aggiuntivi specifici di quella sotto-classe (i.e. la “potenza erogata” per i PSU).

Inoltre, ciascun oggetto JSON di Componente contiene anche un array, il cui contenuto sono i codici numerici delle sue CopieComponente presenti.

Configurazioni e Bundle costituiscono un’eccezione particolare, in quanto per essere istanziati richiedono l’esistenza apriori degli altri Componenti. Di conseguenza, questi due attributi saranno processati dal Parser per ultimi. Il contenuto di tali oggetti si limita al campo numerico “id” e ad un array di numeri interi rappresentanti gli “id” dei Componenti che costituiscono la Configurazione/il Bundle: essi possono essere usati per ottenere le effettive istanze di Componente interrogando il Catalogo parziale, ottenuto processando il resto del file.

**Utenti:** il file *utenti.json* rappresenta un oggetto con soli due attributi: “Clienti” e “Amministratori”. In corrispondenza di ciascuna delle due etichette è presente una lista di oggetti, i quali verranno utilizzati per creare – rispettivamente – un Cliente o un Amministratore.

In ciascuno dei due casi, l’oggetto conterrà i 5 campi necessari per istanziare l’oggetto in memoria: id, nome, cognome, email, password.

**Ordini:** il file *ordini.json* deve esplicitamente essere richiesto per ultimo, in quanto dipende dalla conoscenza apriori del Catalogo e della lista dei Clienti.

Tale fila contiene un array di oggetti, ciascuno rappresentante un Ordine a sé stante. Per far ciò, l’oggetto contiene un campo “cliente”, il cui contenuto equivale ad una email valida per poter accedere alla corrispettiva istanza di Cliente richiesta (N.B. la lista dei clienti è indicizzata sotto forma di mappa, dove la chiave d’accesso è la sua email: ciò permette, fra le altre cose, di velocizzare le operazioni di Login).

Inoltre, nel campo “carrello” è presente un ulteriore array: gli oggetti in esso contenuti presentano solo due campi numerici interi, “copia” e “componente”, i quali vengono usati per interrogare il Catalogo (pre-esistente) per trovare ciascuna istanza di Componente acquistata (cercandola per id) con la sua relativa CopiaComponente.

Infine, l’oggetto JSON dell’Ordine contiene tutti gli altri campi costanti richiesti per la sua costruzione: id, metodoPagamento, numeroCarta, cvv, città, indirizzo, CAP.

Nella sotto-cartella **“modello\_oggetti\_dominio”** è presente una copia *“pulita”* dei tre file, riempita con dei dati d’esempio ideali per permettere la dimostrazione del funzionamento del software.

**3. La classe Parser**

Per costruire la classe Parser, si è deciso di utilizzare soli metodi e attributi *“static”*, in modo tale da renderla al pari di un servizio o una libreria di funzioni, anziché un effettivo oggetto software di cui andrebbe assegnata la responsabilità della gestione e creazione. In tale senso, non è stato previsto l’accesso esclusivo, e il Sistema e i suoi handler possono entrambi accedervi.

Per la scrittura dei metodi del Parser si è scelto di sfruttare la libreria esterna **org.json**.

Gli attributi al momento istanziati sono tre costanti, rappresentanti l’URL dei tre file citati al punto precedente. La classe è stata creata nel package **“data”**.

**3.1. Le funzioni di Caricamento**

Una famiglia di funzioni dal nome nella forma generica *“caricaOggetto”* è stata creata con lo scopo di leggere i file del database all’avvio del sistema PCReady e caricarvi all’interno tutti gli oggetti necessari per il suo corretto funzionamento.

Fa eccezione la funzione per il caricamento del Catalogo, che per comodità di lettura è stata suddivisa in due funzioni: la funzione **“createCatalogo”**, chiamata dal sistema PCReady per ottenere il Catalogo dal database, che a sua volta chiama la funzione **“parseCatalogo”**, contenente una lunga lista di controlli – dovendo differenziare fra le diverse sotto-classi di Componente.

Le funzioni **“caricaClienti”** e **“caricaAdmin”** agiscono con la stessa logica e sullo stesso file (utenti.json) per creare in memoria, rispettivamente, le mappe dei Clienti e degli Amministratori.

La funzione **“caricaOrdini”** viene chiamata per ultima dal sistema, e richiede in ingresso la mappa degli Utenti e il Catalogo inizializzati dalle precedenti funzioni.

La logica interna di ciascuna di queste funzioni si basa su una famiglia di funzioni di parsing dal nome generico nella forma *“processOggetto”*, che – preso in ingresso un oggetto JSON – forniscono in uscita un’istanza di classe *Oggetto* corrispondente.

Nel caso specifico di **“processComponente”**, la funzione si limita a costruire un generico Componente a partire dal JSON, mentre la logica interna alla già citata *“parseCatalogo”* effettua uno switch-case sull’attributo “categoria” dell’oggetto: così facendo, può creare un’istanza nella sotto-classe specializzata, partendo dal Componente generico ottenuto da *“processComponente”* e aggiungendo i campi extra richiesti.

**3.2. Le funzioni di Salvataggio**

In maniera perfettamente analoga, le funzioni di salvataggio hanno nome nella forma generica *“salvaOggetto”* (**salvaCatalogo**, **salvaUtenti**, **salvaOrdini**): prendendo le istanze degli oggetti tramite il sistema PCReady, si occupano di ri-generare i relativi file JSON partendo dai loro attributi, con una logica implementata attraverso una famiglia di funzioni apposite dal nome generico nella forma *“jsonOggetto”*.

Fa eccezione la funzione **“generalJson”**, che si occupa di fornire un JSON specializzato rispetto all’output della funzione **“jsonComponente”**, sulla base della sotto-classe specializzata del Componente che si vuole salvare su Database, completo degli attributi extra.

**3.3. Le funzioni di utility**

La classe Parser contiene, infine, una famiglia di funzioni di “utility”, il cui scopo è solo quello di gestire la lettura e scrittura sui file che momentaneamente simulano il Database. In futuro, tali funzioni andrebbero sostituite con una famiglia corrispettiva che si occupi di gestire richieste HTTP, e pertanto sono state scritte in maniera tale da rendere quanto più banale e intuitiva suddetta futura transizione.

La funzione **“getFileContent”** si occupa semplicemente di ottenere il contenuto testuale di un file: suddetta funzione è stata specializzata nelle due funzioni **“getArrayFromFile”** e **“getObjectFromFile”**, che in aggiunta convertono automaticamente il contenuto del file, rispettivamente in un oggetto JSONArray o JSONObject, entrambi classi della libreria **org.json**.

La funzione **“writeToFile”** si occupa di sovrascrivere un file con una nuova stringa di testo, e quindi effettua l’operazione inversa di *“getFileContent”*. Per ottenere la rispettiva forma testuale di un JSONObject o di un JSONArray si utilizza il metodo standard **“toString”** prima di passarli come parametro. In questo caso specifico, *“toString”* accetta un parametro numerico intero, che indica il numero di spazi di cui è costituito un carattere *“tabulatore”*, nella formattazione del contenuto testuale: si è scelto ovunque di porre tale parametro pari a 4, nel rispetto della formattazione originale dei file presenti nella cartella *“modello\_oggetti\_dominio”*.