**ARCHITETTURA CLIENT-SERVER:**

Il software realizzato è stato progettato e suddiviso in parti tenendo conto di criteri di suddivisione sia nel del dominio dello spazio (suddivisione in layer) che del tempo (in quali fasi del ciclo di vita del software operare la suddivisione).

-Dominio dello spazio (ovvero layer applicativi): sono previste due macro suddivisioni, ovvero un progetto Client e un progetto Server.

-Dominio del tempo (ovvero il Life cycle applicativo): la suddivisione client-server avviene a **Run Time** (altri momenti possibili potevano essere a Build Time o a Compile Time).

Abbiamo deciso questa suddivisione in quanto team di sei persone che poteva permettersi di suddividersi in sotto-team e creare due progetti sfruttando il “**parallelismo**” e facendoli comunicare tra loro a runtime attraverso **Api Http-Rest**.

La divisione tra Client e Server avviene secondo questo criterio:

* Il Server rappresenta l’**unica fonte di verità** (dati, entità, chiavi), gestisce la **persistenza** dei dati e la definizione di **logiche** di business (dati, auth, ecc.);
* il Client si interfaccia con l’utente e gestisce alcune logiche di **presentazione** e le **viste** relative all’interazione con l’utente.

Il server è stato sviluppato usando il framework SpringBoot, per la sola parte di interfacciamento con le librerie a basso livello. Le logiche applicative sono state comunque scritte in modo indipendente dal framework. Il principio su cui ci si basa l’intero progetto è quello di mantenere una suddivisione interna ad **astrazioni crescenti** (Model, Business Rules, Data Layer Access, Http Access) che comunicano tra loro tramite Inversion of Control, raggiunto tramite Dependency Injection, sfruttando il meccanismo di SpringBoot qui descritto.

L’Inversion of Control Container di SpringBoot è un oggetto ("Application Context”) che è responsabile di **istanziare, configurare ed assemblare** gli oggetti (“Beans”) gestire il loro ciclo di vita (es.costruzione, distruzione, singleton) e “**iniettarli**” (ovvero fornirne le istanze pronte) alle classi/istanze che li utilizzano. Inoltre crea classi concrete a partire da interfacce e decoratori, anch’esse iniettate dove necessario tramite Dependency Injection.

Dependency Injection: se con la programmazione ad oggetti classica dopo il settaggio di una dipendenza il programmatore deve preoccuparsi di istanziare l’oggetto A da cui B dipende all’interno di B (tramite creazione diretta o indirettamente tramite una Factory, in ogni caso in fase di sviluppo a compile time), con la DI possiamo iniettare B dentro A a runtime, e tramite moduli non conosciuti a compile time (ad esempio librerie incluse e build time, configurazioni a runtime, ecc.) tramite un meccanismo che possiamo definire a “**plugin**”. Siccome in realtà le dipendenze possono essere molteplici e a cascata, si deve parlare più propriamente di un albero di dipendenze. Il provider di SpringBoot quindi provvede, a runtime, a risolvere tutto l’albero delle dipendenze fino ai componenti indipendenti, crea le istanze dei Bean e li inietta nelle classi che li utilizzano. Non solo, ma dove SpringBoot ha la responsabilità di istanziare classi concrete, si occupa anche della creazione automatica del codice necessario per il loro funzionamento.

Ad esempio ciò avviene per le classi del layer Repository. Nell’UML condiviso questo layer mostra delle classi che in realtà non sono state implementate manualmente, infatti nel codice ci sono solo interfacce che estendono l'interfaccia (fornita sempre da SpringBoot) CrudRepository (che offre classici metodi CRUD, create, update, read, delete, find, ecc.). Quando viene compilato il progetto e poi eseguito, il Provider provvede a creare una classe concreta con lo stesso nome dell’interfaccia e che internamente implementa logiche di data access a basso livello (ad esempio le classi Hibernate, l’OR/M fornito con Springboot) di cui non si occupa l’applicazione Java. Il tutto gestito da una configurazione che se modificata (per esempio per accedere ad un’altra tecnologia di database) provoca la ricompilazione del codice per adattarsi ad un nuovo contesto di utilizzo.

**Le dipendenze sono quindi e perciò sempre verso l’interno,** ovvero verso le astrazioni e verso le logiche, completamente indipendenti da Springboot. Se domani decidessimo di cambiare db, o OR/M, o tecnologia, dovremmo semplicemente dire a Spring Boot di utilizzare un’altra tecnologia e lui compilerebbe nuove classi che implementano il CrudRepository.

Possiamo idealmente rappresentare questa architettura come un insieme di cerchi concentrici in cui il layer del Model sta all’interno, subito dopo abbiamo il layer Repository e all’esterno il layer di Data Access. Questi cerchi sono attraversati dai dati e ad ogni layer corrispondono trasformazioni, logiche di funzionamento e comunicazione verso gli altri layer.

Stesso discorso per il layer API, che implementa delle logiche di business, ma è completamente agnostico rispetto alle chiamate HTTP, che infatti vengono aggiunte da Springoot tramite oppurtuni decoratori (@GetMapping, @Request, ecc.) che consentono a Spring Boot di estendere le funzionalità delle API nel contesto specifico di una applicazione web. Il meccanismo del decoratore è complementare rispetto a quello della realizzazione automatica di classi a partire da interfacce visto con i CrudRepository, e l’obiettivo è lo stesso: aggiungere codice in fase di compilazione togliendo allo sviluppatore l’onere di riscrivere codice di libreria ma dandogli l’opportunità di configurarlo in base alle proprie esigenze (es. definizione rotte, mappature, dati passati, ecc.). E’ poi Springboot a scrivere le funzioni di serializzazione, deserializzazione, e le classi che costruiscono/decostruiscono la comunicazione tramite HTTP Request/Response.

Se un domani tuttavia si decidesse di fornire le dette funzionalità anzichè tramite HTTP tramite altro tipo di interfaccia (ad esempio compilando l'applicazione come JAR per essere utilizzata come libreria in una applicazione mobile, in quel caso anche con un opportuno data access layer specifico per il mobile come sqlite) le api sarebbero sempre perfettamente funzionanti in quanto non contengono dettagli implementativi dipendenti dal contesto.

Spring boot in definitiva fornisce supporto dove non è richiesta logica ma solo “lavoro di manovalanza” ma non interviene ed anzi offre flessibilità dove c’è “lavoro di concetto”.

Hibernate

Lo scopo di questa libreria è quello di fornire un mapping diretto delle classi Java in tabelle del Database (attraverso il decoratore @Entity di Springboot). Ogni entità del model è transiente e la classe corrisponde in prima approssimazione (in un database SQL) ad una tabella, le proprietà della classe sono le colonne della tabella, e la singola istanza una riga. Questo meccanismo si basa sul concetto di OR/M (Object Relational Mapping), che disaccoppia il model dal sistema di persistenza dei dati, da come questo sia realizzato, e da come interfacciarsi con esso. Nel pattern OR/M (detto anche “Data Mapper”), sono previste delle classi (ad esempio usando il pattern Repository come in questo progetto) che eseguono le operazioni di save, get, delete e update in modo trasparente rispetto al model. Siccome inoltre si deve propriamente parlare di grafo delle classi, l’OR/M mappa anche le dipendenze sotto forma di relazioni in un classico db SQL, e provvede a risolvere le constraint previste. Inoltre per evitare che venga caricato l’intero albero delle dipendenze l’OR/M utilizza il pattern Proxy/Lazy Loading che forza la lettura di oggetti di altre classi/tabelle solo se effettivamente sono accedute a runtime. In altri termini Hibernate e gli OR/M svolgono la funzione prevista dal pattern Adapter.

Le entità del Model che vengono trasformate in tabelle del DB sono state progettate attraverso una struttura ad albero la cui radice è il ComponentType, avente come figlio l’entità ComponentFamily, la quale ha come figli le entità Component e Compatibility Constraints. Il Component ha come figlio il ReadySetUp. Sulla base di questa struttura ad albero sono state sviluppate manualmente le operazioni di cancellazione a cascata nel DB ponendo attenzione ai vincoli di integrità referenziale, cancellando quindi nel caso del ComponentType tutti gli oggetti che ad esso fanno riferimento (Families, Components, ReadySetups, CompatibilityConstraints) in modo da evitare oggetti privi di consistenza nel db.

DTO (Data Transfer Object):

Layer che contiene delle classi di trasferimento dati da un layer all’altro, è una classe POJO con getters() and setters(). Il DTO non è mai visto dal Client direttamente e serve per mappare i campi del Json effettivamente trasmesso. Il suo scopo è mantenere la Business Logic indipendente dal Client. Le API andranno ad utilizzare i DTO e non le classi del Model, questo favorisce la creazione di API stabili e non soggette ad eventuali cambiamenti che avvengono nello strato di Business.

LAYER API:

L’annotazione @RestController su ciascuna classe API trasforma i metodi in servizi Web e le classi del DTO in oggetti JSON. I principali oggetti gestiti dalle classi di questo Layer sono:

-ResponseEntity : entità che il server manda al Client in metodi POST(ritorna l’entità creata sul Database), PUT(ritorna l’entità modificata sul database) e DELETE(ritorna l’entità booleana che segnala il successo o meno della cancellazione).

-ResponseBody: parametro di alcuni metodi GET attraverso cui il server manda un Json al client con all’interno gli oggetti richiesti

-PathVariable: parametro che rappresenta una variabile (id) mandata dal Client al server attraverso la quale si effettua la ricerca di un oggetto del DB.

-RequestParam: si trova per lo più nei GET e serve per mappare richieste di tipo Querystring (per ora non previste)

-RequestBody: mandato dal Client al Server per mappare i campi del Json con quelli del DTO. Si usa principalmente nei metodi POST e PUT.

-RequestHeader: mandato da Client a Server ed usato per mappare le informazioni dell’Header del Json con i parametri del metodo Api stesso (nel nostro caso Map<String, String> headers). Ad esempio tutte le operazioni sotto autenticazione hanno bisogno di un controllo a priori in cui il Client invia al server un Json con il Token di sessione nell’intestazione (coppia chiave valore), per fare in modo che il server verifichi la validità del Token.

-Metodo save(Entity entity): salva nel DB la row e ritorna l’oggetto di quell’Entity.

**UNIT TEST:**

In questo progetto sono stati previsti unit test. Ad ogni classe viene associata una classe di Test che contiene dei metodi. Lo scopo di ciascun metodo di test è testare la logica di un metodo specifico della classe testata. Gli unit test si possono suddividere in due categorie:

-Test **deterministici**: test di metodi deterministici (ovvero dato uno stato S si ha sempre F(A, S)=B qualsiasi valore di S). Questi test sono effettuati gestendo sia l’**input coverage** tramite partizionamento dei casi di input (casi logici possibili da testare) che il **code coverage**, determinando cioè tutti i possibili cammini di esecuzione del codice (if, loop, ecc.);

-Test **non deterministici** (ovvero si ha F(A,S) = B, dove B dipende sia da A che da S): in questo caso il partizionamento è insufficiente, e può diventare molto complesso individuare un sottoinsieme di stati significativo. In questo caso l’obbiettivo principale rimane solo il code-coverage. In questo caso si parla di unit test che verificano la transizione da uno stato iniziale a uno stato finale tramite opportuni partizionamenti dell’input. Quello che si testa non è il fatto che, in seguito ad un passaggio da Stato1 a Stato2, si riesca a tornare allo Stato1 (non è infatti previsto in un ambiente non deterministico), ma si testa che la variazione di stato sia sempre la stessa.

I test del Model e DTO sono semplicemente costituiti per la maggior parte dei casi da metodi di verifica delle proprietà (ad eccezione dell’entità Session che ha qualche metodo di verifica logica).

Per quanto riguarda invece il Repository Layer è composto da Test che verificano i soli metodi descritti nelle interfacce e per cui è stata implementata una query (nel linguaggio HQL di Hibernate). Per eseguire i test, per mantenere il più possibile un ambiente non condizionabile dall’esterno come può essere un DB SQL (a cui è possibile accedere dall’esterno inserendo dati che possono invalidare il test) viene utilizzato un database in memoria (H2, compatibile con Hibernate).

L’aspetto più importante nella gestione dei test è quello di isolare la classe da testare dalle sue dipendenze. Se venissero utilizzate le dipendenze reali (che come si è visto vengono iniettate dal provider) in caso di test fallito (o anche riuscito) non si saprebbe quale componente e quale strato ha fallito. Fornendo invece un contesto simulato alla classe da testare, in caso di fail (o success) la responsabilità sarebbe solo ed unicamente di quel componente.

A livello di codice, per eseguire i test sono state utilizzate la libreria JUnit e la libreria Mockito. JUnit è una libreria che permette di creare classi e metodi di test, dove può essere prevista una fase di inizializzazione, e che offre un insieme di metodi per eseguire il test, tramite il concetto di “assertion”: se l’assertion è vera, il test è superato, altrimenti è fallito.

Mockito si occupa di creare classi o metodi simulati che rispettano lo stesso contratto di quella vera e di iniettarle nella classe da testare al posto di quella vera (che non è quindi consapevole di utilizzare un “**mock**”). Tramite lo stesso meccanismo è possibile simulare non solo le dipendenze sviluppate manualmente (metodi con logica), ma anche quelle di libreria (repository) e anche di sistema. Ad esempio tramite @MockMvc è possibile simulare l’intero contesto del web server e una HTTP request intorno ad una API Rest, in modo da testare la API in un contesto simile a quello reale.

MODELLO DI COMPATIBILITA’:

Il modello di compatibilità si basa sulla suddivisione dei componenti in famiglie, che non sono necessariamente criteri esogeni (come marca o produttore) ma opportune tassonomie decise in modo tale da poter definire in modo comodo vincoli di compatibilità tra di esse.

E’ prevista quindi una classe “compatibilità” con due proprietà, famiglia 1 e famiglia 2, in cui ogni entry rappresenta la compatibilità tra i componenti della famiglia 1 e i componenti della famiglia 2. Perciò quando si vogliono ricavare tutti i componenti compatibili con un determinato componente selezionato dall’utente si filtrano tutte le famiglie compatibili con la famiglia di quel componente e poi si restituiscono (tramite una operazione di flatmap delle foglie) tutti i componenti appartenenti a tali famiglie.

Le compatibilità vengono inoltre filtrate rispetto al tipo del componente, in quanto la sequenza di selezione dei componenti segue un ordine prestabilito. L’ordine non viene definito nel codice ma viene deciso in base all’attributo sortOrder di componentType in modo tale che l’admin possa modificarlo facilmente se lo ritiene opportuno e non è vincolato all’ordine di default del primo inserimento.

E’ comunque possibile implementare una funzionalità che permetta di far scegliere all’utente l’ordine di inserimento dei componenti, tuttavia in quel caso è essenziale anche completare l’associazione di compatibilità non per componenti a due a due ma per ogni possibile coppia.

In questo modello non abbiamo previsto compatibilità multiple tra i tipi di componenti, perché aggiungerebbero complessità al modello che non è necessaria dato che nella maggior parte dei casi abbiamo trovato che definire le compatibilità a due a due si rivela sufficiente.

Idea per estendere il modello : avere righe di compatibilità doppie (ad esempio se c’è compatibilità tra CPU-MOTHERBOARD si mette anche la compatibilità CPU-CASE) e le compatibilità non sono solo a due a due ma tra più tipi di componenti. Quando l’utente seleziona un componente e il frontend chiama il metodo dell’api, quest’ultimo deve calcolarsi anche il nextType che è il prossimo tipo di componente nell’ordine deciso, in modo che i componenti restituiti appartengano solo al nextType. Al termine della selezione di tutti i componenti, l’idea è quella di effettuare un ciclo sull’array dei componenti del setup ed effettuare eventuali query di verifica di compatibilità multipla (ad esempio controllo CPU-CASE che prima non era stato effettuato in quanto il CASE non era il nextType per la CPU) e in caso di problemi (la CPU non è compatibile con il CASE selezionato perché la compatibilità del CASE è stata data da un previousType che non era la CPU) allora viene segnalato all’utente. Questa soluzione non è stata implementata in quanto è richiesta solo una modifica del modello e lo sviluppo di un algoritmo su entrambe le applicazioni, ma anche l’inserimento di molti dati delle compatibilità, sforando quindi le date della consegna.

SICUREZZA

L’applicazione prevede un insieme di API non sicurizzate da un accesso autenticato (per la lettura dei dati dal sistema di persistenza), delle API di accesso tramite login, e infine delle API sicurizzate tramite condivisione tra client e server di una sessione identificata tramite token. Non è stato previsto per semplicità un meccanismo di autenticazione Http (come basic authentication) ma è stato semplicemente condiviso il token tramite header nelle chiamate.

A livello di applicazione è stato utilizzato uno Strategy pattern che verificava in modo centralizzato l’autenticazione ed eseguiva la funzionalità richiesta solo se in possesso di Token (tramite una interfaccia Callable).

CONCLUSIONI

Questo progetto ha permesso di:

* legare tra loro concetti di programmazione di base ad oggetti in un unico disegno complessivo;
* approfondire tematiche legate alla persistenza dei dati e all’accesso ad un sistema informativo tramite http;
* imparare ad utilizzare design pattern (come Object Mapper, Strategy pattern, Decorator, Repository, ecc.)
* imparare e approfondire concetti legati al mondo delle architetture del software (come l’architettura ad astrazioni crescenti).