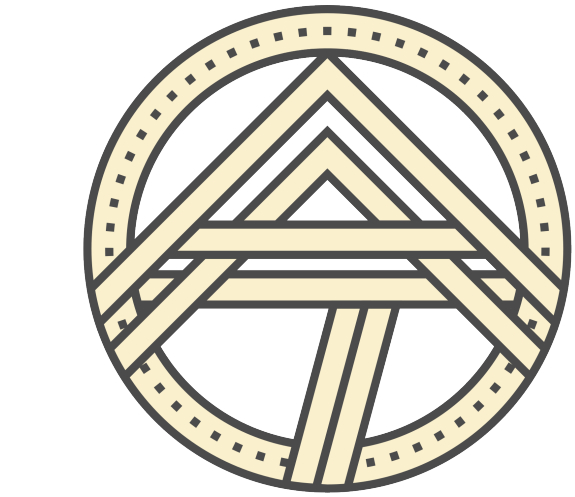

ARCHI7ECHS

Archi7echs - archi7echs@gmail.com

Progetto di **Ingegneria del Software**
A.A. 2024/2025

Specifica Tecnica

Autore: Il team

Ultima Modifica: 10/04/2025

Tipologia Documento: Interno

Tabella delle revisioni

Rev.	Data	Descrizione	Elaborazione	Verifica
0.5.0	10-04-2025	Fix e stesura architettura deployment	Giacomo Pesenato	Leonardo Lucato, Francesco Pozzobon
0.4.0	07-04-2025	Stesura sezione back-end descrizione moduli	Giacomo Pesenato	Leonardo Lucato, Francesco Pozzobon
0.3.0	02-04-2025	Stesura sezione back-end,tecnologie e architettura	Leonardo Lucato, Giacomo Pesenato	Gabriele Checchinato, Giovanni Salvò
0.2.0	31-03-2025	Stesura sezione componenti front-end	Gabriele Checchinato, Pietro Valdagno	Francesco Pozzobon, Giovanni Salvò
0.1.0	21-03-2025	Inizio stesura documento	Gabriele Checchinato	Giovanni Salvò, Pietro Valdagno

Indice

1) Introduzione

1.1) Finalità del documento

Questo documento ha l'obiettivo di fornire una descrizione dettagliata e strutturata degli aspetti tecnici fondamentali del progetto **3Dataviz**. In particolare, esso rappresenta una guida di riferimento per comprendere l'architettura del sistema, le scelte implementative adottate e le specifiche di deployment. Attraverso un'analisi approfondita, il documento illustra i principali componenti software e le tecnologie utilizzate. Inoltre, vengono descritte le motivazioni alla base delle decisioni progettuali, con un focus su scalabilità, manutenibilità e sicurezza del sistema. Gli obiettivi principali di questa specifica tecnica sono:

- Fornire una documentazione chiara e dettagliata a supporto dello sviluppo e della manutenzione del software.
- Garantire l'allineamento con i requisiti funzionali e non funzionali definiti nel documento *Analisi dei Requisiti v1.0.0*.
- Definire una base comune di conoscenza per tutti i membri del team, facilitando l'integrazione e l'evoluzione del sistema.

1.2) Scopo del progetto

L'obiettivo è realizzare una piattaforma web di visualizzazione tridimensionale dei dati, che consenta all'utente che la utilizza di navigare e interagire con grafici a barre verticali 3D rappresentanti dati complessi, utili per l'analisi e la presentazione di informazioni. Il prodotto deve essere progettato per poter rappresentare dati, in un modello 3D, navigabile e interattivo.
Dunque le sue funzionalità$_G$ principali includono:

- **Funzionalità di un ambiente 3D**:
  - **Rotazione**: permettere la rotazione del grafico per osservarlo da diverse angolazioni.
  - **Pan**: consentire lo spostamento del grafico sul piano orizzontale.
  - **Zoom**: abilitare l'avvicinamento e l'allontanamento dal grafico.
  - **Auto-positioning**: posizionare automaticamente il grafico in una vista ottimale.
- **Visualizzazione del valore medio globale**: il sistema deve consentire di visualizzare un piano parallelo alla base, che rappresenta il valore medio globale dei dati.
- **Opacizzazione o nascondimento delle barre**: il sistema deve offrire la possibilità di opacizzare o nascondere le barre con valori superiori o inferiori rispetto a:
  - una barra selezionata;
  - il valore medio globale rappresentato dal piano visualizzato.

  Inoltre, deve permettere di lasciare visibili o non opacizzati solo i valori di minimo o di massimo delle y, ossia i punti estremi.
- **Visualizzazione dei valori corrispondenti a una barra**: il sistema deve consentire di visualizzare i valori corrispondenti a una barra quando questa è soggetta a un evento « hover$_G$ « del mouse.
- **[Opzionale] Visualizzazione del valore medio del singolo elemento**: il sistema deve consentire di visualizzare un piano parallelo alla base, che rappresenta il valore medio di un singolo elemento di un asse (X o Z).

1.3) Glossario

All'interno del documento saranno spesso utilizzati degli acronimi o termini tecnici per semplificare la scrittura e la lettura. Per garantire che quanto scritto sia comprensibile a chiunque, è possibile usufruire del *glossario*. Tutte le parole consultabili nel glossario saranno identificate da una «G» in colore blu. Facendo click sul collegamento si aprirà una scheda del browser con il glossario

1.4) Riferimenti

1.4.1) Riferimenti normativi

- Norme di Progetto (v 1.0.0)
- Riferimento al capitolato$_G$ 5 di ***Sanmarco Informatica SPA - 3Dataviz***: https://www.math.unipd.it/~tullio/IS-1/2024/Progetto/C5.pdf - *Ultimo accesso 20/03/2025*
- Riferimento alle slide IS: ***Regolamento del progetto$_G$ didattico***: https://www.math.unipd.it/~tullio/IS-1/2024/Dispense/PD1.pdf - *Ultimo accesso 20/03/2025*

1.4.2) Riferimenti informativi

- Riferimento documentazione: ***Svelte***: https://svelte.dev/docs/svelte/overview
  Ultimo accesso 20/03/2025
- Riferimento documentazione: ***Threlte***: https://threlte.xyz/
  Ultimo accesso 20/03/2025
- Riferimento documentazione: ***Sping_Boot*** https://spring.io/projects/spring-boot
  Ultimo accesso 20/03/2025
- Riferimento documentazione: ***Maven***: https://maven.apache.org/
  Ultimo accesso 20/03/2025
- Riferimento documentazione: ***PostgreSQL***: https://www.postgresql.org/
  Ultimo accesso 4/10/2025
- Riferimento documentazione: ***Docker***: https://docs.docker.com/
  Ultimo accesso 10/04/2025
- Riferimento alle slide IS: ***Progettazione: le dipendenze tra componenti***:
  https://www.math.unipd.it/~rcardin/swea/2022/Dependency%20Management%20in%20Object-Oriented%20Programming.pdf - *Ultimo accesso 20/03/2025*
- Riferimento alle slide IS: ***Analisi e descrizione delle funzionalità: Use Case e relativi diagrammi UML***: https://www.math.unipd.it/~rcardin/swea/2022/Diagrammi%20Use%20Case.pdf - *Ultimo accesso 1/04/2025*
- Riferimento alle slide IS: ***Progettazione e programmazione: Diagrammi delle classi (UML)***:
  https://www.math.unipd.it/~rcardin/swea/2023/Diagrammi%20delle%20Classi.pdf
  - *Ultimo accesso 20/03/2025*
- Riferimento alle slide IS: ***Analisi e descrizione delle funzionalità: Diagrammi delle attività (UML)***: https://www.math.unipd.it/~rcardin/swea/2022/Diagrammi%20di%20Attivit%C3%A0.pdf
  - *Ultimo accesso 20/03/2025*
- Riferimento alle slide IS: ***Progettazione: I pattern architetturali***: https://www.math.unipd.it/~rcardin/swea/2022/Software%20Architecture%20Patterns.pdf
  - *Ultimo accesso 20/03/2025*

- Riferimento alle slide IS: ***Progettazione: Il pattern Dependency Injection***: https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Architetturali%20-%20Dependency%20Injection.pdf
  - *Ultimo accesso 27/03/2025*
- Riferimento alle slide IS: ***Progettazione: il pattern Model-View-Controller e derivati***: https://www.math.unipd.it/~rcardin/sweb/2022/L02.pdf
  - *Ultimo accesso 20/03/2025*
- Riferimento alle slide IS: ***Progettazione: i pattern creazionali (GoF)***: https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Creazionali.pdf
  - *Ultimo accesso 20/03/2025*
- Riferimento alle slide IS: ***Progettazione: I pattern strutturali (GoF)***: https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Strutturali.pdf
  - *Ultimo accesso 20/03/2025*
- Riferimento alle slide IS: ***Progettazione: I pattern di comportamento (GoF)***: https://drive.google.com/file/d/1cpi6rORMxFtC91nI6_sPrG1Xn-28z8eI/view?usp=sharing
  - *Ultimo accesso 20/03/2025*
- Riferimento alle slide IS: ***Programmazione: SOLID programming***: https://drive.google.com/file/d/1o1Xun2dVVc3mDiaGyN0FrDJhhoO3lfLQ/view?usp=sharing
  - *Ultimo accesso 27/03/2025*

2) Tecnologie

In questa sezione vengono elencate le tecnologie (e librerie) utilizzate all'interno del progetto **3Dataviz**, dalla fase di progettazione alla sua implementazione.
Ogni tecnologia o libreria utilizzata verrà descritta tramite:

1. Nome della tecnologia o libreria
2. Descrizione della tecnologia o libreria e del suo utilizzo
3. Versione della tecnologia o libreria utilizzata
4. Link di riferimento alla sua documentazione

2.1) Servizi e Strumenti

2.1.1) PostgreSQL

- **Descrizione della tecnologia e del suo utilizzo**: PostgreSQL è un sistema di gestione di database relazionali open-source, noto per la sua robustezza, scalabilità e conformità agli standard SQL. Supporta una vasta gamma di tipi di dati e consente l'uso di estensioni per funzionalità avanzate. Nel nostro progetto, PostgreSQL viene utilizzato per:
  1. Memorizzare i dati dell'applicazione in modo strutturato e persistente.
  2. Gestire le relazioni tra le entità del dominio.
  3. Eseguire query complesse per recuperare e manipolare i dati in modo efficiente.
  4. Utilizzare il tipo ENUM per definire valori predefiniti e limitare le opzioni disponibili per determinati campi.
- **Versione della tecnologia utilizzata**:
  - **PostgreSQL**: 17
- **Link di riferimento alla documentazione**:
  - **PostgreSQL**: https://www.postgresql.org/docs/

2.1.2) Maven

- **Descrizione della tecnologia e del suo utilizzo**: Maven è un sistema di gestione dei progetti e automazione della build per Java. Fornisce un modello di progetto standardizzato e gestisce le dipendenze tra librerie e componenti. Nel nostro progetto, Maven viene utilizzato per:
  1. Gestire le dipendenze del progetto tramite il file pom.xml, semplificando l'integrazione di librerie esterne.
  2. Automatizzare il processo di build, test e packaging dell'applicazione.
  3. Eseguire plugin per il testing, la generazione di report e altre attività di sviluppo.
- **Versione della tecnologia utilizzata**:
  - **Maven**: 3.9.9
- **Link di riferimento alla documentazione**:
  - **Maven**: https://maven.apache.org/

2.1.3) Docker

- **Descrizione della tecnologia e del suo utilizzo**: Docker è una piattaforma che consente di sviluppare, distribuire ed eseguire applicazioni in container. Un container è un'unità software che include tutto il necessario per eseguire un'applicazione, come codice, runtime, librerie e dipendenze, garantendo coerenza tra ambienti diversi. Nel nostro progetto, Docker viene utilizzato per:
  1. Costruire e pacchettizzare l'applicazione Spring Boot in un'immagine Docker tramite un Dockerfile.
  2. Gestire l'ambiente di sviluppo e test attraverso docker-compose, orchestrando i servizi necessari, tra cui:
     - Database PostgreSQL per la persistenza dei dati.
     - Ambiente di test per eseguire i test automatici prima della build finale.
     - Applicazione Spring Boot come servizio runtime.
- **Versione della tecnologia utilizzata**:
  - **Docker**: Ultima versione disponibile in ambiente di sviluppo
  - **Maven**: 3.9.9 con JDK Eclipse Temurin 23
  - **PostgreSQL**: 17
- **Link di riferimento alla documentazione**:
  - **Docker**: https://docs.docker.com/
  - **Docker Compose**: https://docs.docker.com/compose/
  - **PostgreSQL**: https://www.postgresql.org/docs/

2.2) Framework

2.2.1) Spring Boot

- **Descrizione della tecnologia e del suo utilizzo**: Spring Boot è un framework basato su Spring che semplifica lo sviluppo di applicazioni Java stand-alone e pronte per la produzione. Fornisce una configurazione automatica e un'architettura modulare per creare applicazioni enterprise in modo efficiente. Nel nostro progetto, Spring Boot viene utilizzato per:
  1. Sviluppare il backend dell'applicazione, implementando la logica di business e l'interfacciamento con il database.
  2. Gestire le connessioni al database PostgreSQL tramite il modulo Spring Data JPA.

3. Esporre API RESTful per l'interazione con il frontend e altri servizi.
4. Gestire la configurazione dell'applicazione tramite il file application.properties e le variabili d'ambiente definite in docker-compose.yml.
5. Facilitare i test automatizzati, sfruttando il supporto nativo per Testcontainers e altre librerie di testing.

L'integrazione con Docker consente di eseguire il backend in un container isolato, garantendo consistenza nell'ambiente di sviluppo e produzione.

- **Versione della tecnologia utilizzata**:
  - **Spring Boot**: 3.4.3
  - **Spring Data JPA**: 3.4.3
- **Link di riferimento alla documentazione**:
  - **Spring Boot**: https://docs.spring.io/spring-boot/docs/current/reference/html/
  - **Spring Data JPA**: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

2.2.2) Svelte

- **Descrizione della tecnologia e del suo utilizzo**: Svelte è un framework di front-end moderno che consente di sviluppare interfacce utente reattive e performanti compilando il codice in JavaScript ottimizzato. A differenza di altri framework come React o Vue, Svelte non utilizza un Virtual DOM, ma compila i componenti in codice JavaScript efficiente che aggiorna direttamente il DOM in modo minimale. Nel nostro progetto, Svelte viene utilizzato per:
  1. Sviluppare l'interfaccia utente in modo efficiente e performante.
  2. Gestire lo stato dell'applicazione attraverso il sistema di store di Svelte.
  3. Integrare API REST per recuperare e visualizzare i dati dinamicamente.
  4. Ottimizzare le prestazioni grazie alla sua architettura basata sulla compilazione.

  Il progetto è strutturato con:
  - Componenti Svelte modulari per una gestione chiara dell'UI.
  - Fetch API per comunicare con il backend in modo asincrono.
- **Versione della tecnologia utilizzata**:
  - **Svelte**: 5.0.0
- **Link di riferimento alla documentazione**:
  - Svelte: https://svelte.dev/docs

2.3) Test

2.3.1) JUnit 5

- **Descrizione della tecnologia e del suo utilizzo**: JUnit 5 è un framework di testing per Java che consente di scrivere e eseguire test automatizzati. È composto da tre moduli principali: JUnit Platform, JUnit Jupiter e JUnit Vintage. JUnit Jupiter è la parte principale del framework, fornendo le annotazioni e le API per scrivere test. Nel nostro progetto, JUnit 5 viene utilizzato per:
  1. Scrivere test unitari e di integrazione per il backend dell'applicazione.
  2. Eseguire test automatici in un ambiente Docker tramite Testcontainers.
  3. Integrare con Mockito per il mocking delle dipendenze durante i test.

4. Fornire report dettagliati sui risultati dei test, facilitando l'individuazione di errori e problemi nel codice.

- **Versione della tecnologia utilizzata**:
  - **JUnit**: 5.10.0
- **Link di riferimento alla documentazione**:
  - **JUnit**: https://junit.org/junit5/docs/current/user-guide/

2.3.2) PITest

- **Descrizione della tecnologia e del suo utilizzo**: PITest è un framework di test di mutazione per applicazioni Java. Il test di mutazione è una tecnica avanzata per valutare la qualità dei test unitari generando e iniettando mutazioni nel codice sorgente e verificando se i test sono in grado di rilevarle. Questo aiuta a identificare le debolezze nella suite di test e a migliorare la copertura e l'affidabilità del codice. Nel nostro progetto, PITest viene utilizzato per:
  1. Analizzare l'efficacia dei test unitari, verificando se riescono a rilevare mutazioni introdotte nel codice.
  2. Identificare punti deboli nella suite di test, segnalando eventuali scenari non coperti adeguatamente.
  3. Migliorare la qualità del codice, incentivando la scrittura di test più robusti.
  4. Integrare il testing nei processi CI/CD, garantendo un monitoraggio continuo della qualità del codice.

PITest viene configurato all'interno del progetto Maven ed eseguito automaticamente come parte del processo di testing, fornendo report dettagliati sui mutanti generati e uccisi.

Per garantire la compatibilità con JUnit 5 e Spring, nel progetto sono utilizzati i seguenti plugin:

1. **pitest-junit5-plugin**: Permette l'integrazione di PITest con JUnit 5.
2. **arcmutate-spring**: Estensione per migliorare il supporto ai test su applicazioni Spring.

- **Versione della tecnologia utilizzata**:
  - **PITest**: 1.19.0
  - **pitest-junit5-plugin**: 1.1.0
  - **arcmutate-spring**: 1.0.0
- **Link di riferimento alla documentazione**:
  - **PITest**: https://pitest.org/
  - **Plugin Maven per PITest**: https://plugins.pitest.org/maven/
  - **pitest-junit5-plugin**: https://github.com/pitest/pitest-junit5-plugin

2.3.3) Mockito

- **Descrizione della tecnologia e del suo utilizzo**: Mockito è un framework di mocking per Java utilizzato principalmente nei test unitari. Permette di simulare il comportamento di classi e dipendenze, consentendo di testare unità di codice in modo isolato senza dover dipendere da componenti reali come database o servizi esterni. Nel nostro progetto, Mockito viene utilizzato per:
  - Simulare dipendenze nelle classi testate, evitando la necessità di istanziare oggetti reali.
  - Verificare il comportamento del codice, assicurandosi che determinati metodi vengano chiamati con i parametri corretti.

    - Testare componenti Spring Boot, come service e repository, isolandoli dall'infrastruttura sottostante.
    - Migliorare la velocità dei test, riducendo il tempo di esecuzione rispetto a test che interagiscono con database o API reali.

L'integrazione con JUnit 5 e Spring Boot avviene tramite le annotazioni `@Mock`, `@InjectMocks` e `@ExtendWith(MockitoExtension.class)`, garantendo una configurazione semplice ed efficace.

- **Versione della tecnologia utilizzata**:
    - **Mockito Core**: 5.14.2
    - **Mockito JUnit Jupiter**: 5.14.2
- **Link di riferimento alla documentazione**:
    - **Mockito**: https://site.mockito.org/
    - **Mockito per JUnit 5**: https://javadoc.io/doc/org.mockito/mockito-junit-jupiter/latest/

2.3.4) Testcontainers

- **Descrizione della tecnologia e del suo utilizzo**: Testcontainers è una libreria Java che semplifica l'esecuzione di test di integrazione utilizzando container Docker. Consente di avviare istanze temporanee di database, servizi o altre dipendenze necessarie per i test, garantendo un ambiente isolato e riproducibile. Nel nostro progetto, Testcontainers viene utilizzato per:
    1. Eseguire test di integrazione con un'istanza PostgreSQL in un container Docker, garantendo che i test siano eseguiti in un ambiente simile a quello di produzione.
    2. Creare e gestire container in modo programmatico, evitando la necessità di configurazioni manuali.
    3. Garantire che i test siano indipendenti dall'ambiente locale, riducendo il rischio di errori dovuti a configurazioni diverse tra sviluppatori.
    4. Integrare facilmente con JUnit 5 e Spring Boot, sfruttando le annotazioni per configurare i container necessari.
- **Versione della tecnologia utilizzata**:
    - **Testcontainers Core**: 1.20.6
    - **Testcontainers JUnit Jupiter**: 1.20.6
    - **Testcontainers PostgreSQL**: 1.20.0
- **Link di riferimento alla documentazione**:
    - **Testcontainers**: https://www.testcontainers.org/
    - **Testcontainers JUnit 5**: https://java.testcontainers.org/test_framework_integration/junit_5/
    - **Testcontainers PostgreSQL**: https://www.testcontainers.org/modules/databases/postgres/

2.4) Linguaggi

2.4.1) Java

- **Descrizione della tecnologia e del suo utilizzo**: Java è un linguaggio di programmazione ad oggetti, ampiamente utilizzato per lo sviluppo di applicazioni enterprise. La sua portabilità,

grazie alla Java Virtual Machine (JVM), e la vasta gamma di librerie disponibili lo rendono una scelta popolare per progetti complessi. Nel nostro progetto, Java viene utilizzato per:

1. Sviluppare il backend dell'applicazione, implementando la logica di business e l'interfacciamento con il database.
2. Utilizzare il framework Spring Boot per semplificare la configurazione e la gestione delle dipendenze.
3. Eseguire test automatizzati tramite JUnit e Mockito.
4. Integrare librerie esterne come PITest per il test di mutazione e TestContainers per i test di integrazione.

- **Versione della tecnologia utilizzata**:
  - **Java JDK**: 23
- **Link di riferimento alla documentazione**:
  - **Java**: https://docs.oracle.com/en/java/

2.4.2) Typescript

- **Descrizione della tecnologia e del suo utilizzo**: TypeScript è un superset di JavaScript che aggiunge tipizzazione statica e altre funzionalità avanzate al linguaggio. È progettato per migliorare la produttività degli sviluppatori e la qualità del codice, rendendo più facile la gestione di progetti complessi. Nel nostro progetto, TypeScript viene utilizzato per:
  1. Sviluppare il frontend dell'applicazione, sfruttando le funzionalità di tipizzazione per garantire una maggiore sicurezza del codice.
  2. Integrare con Svelte per creare componenti reattivi e performanti.
  3. Utilizzare librerie esterne come Threlte per la visualizzazione 3D dei dati.
- **Versione della tecnologia utilizzata**:
  - **TypeScript**: 5.8.2
- **Link di riferimento alla documentazione**:
  - **TypeScript**: https://www.typescriptlang.org/docs/

3) Architettura

3.1) Architettura logica

Nel nostro progetto abbiamo scelto di adottare un'architettura esagonale, che ci permette di organizzare il codice in maniera ordinata e con una chiara separazione dei compiti tra le varie componenti. Al centro dell'architettura si trova il core domain, dove risiede tutta la logica di business. Questo cuore del sistema è progettato per essere indipendente da elementi esterni come database, API o librerie specifiche, rendendo così l'applicazione più semplice da manutenere, testare e far evolvere nel tempo.

I controller, che si trovano nelle zone più esterne dell'architettura (package controller), rappresentano il punto di contatto tra il mondo esterno e il nostro sistema. Sono loro a gestire le richieste in arrivo dai client e ad inoltrarle verso i servizi interni. Qui avvengono operazioni come il caricamento di file o la richiesta di dati da fonti esterne, gestite da classi come UploadController e ExternalDataController.

Il livello di servizio (package service) si occupa di coordinare le varie operazioni tra il dominio applicativo e le componenti più tecniche. Qui troviamo, ad esempio, CoordinateService o ExternalDataService, che elaborano i dati e orchestrano le logiche applicative. Le entità del dominio (package model), come CoordinateEntity e MatrixData, definiscono le strutture dati principali su cui si basa il funzionamento del sistema.

La gestione della persistenza dei dati è affidata al package repository, che contiene le interfacce per accedere al database. In questo modo, tutta la logica legata al recupero dei dati è isolata dal resto dell'applicazione, rendendola più pulita e flessibile. La configurazione delle risorse esterne è centralizzata nel package config, dove definiamo, ad esempio, le impostazioni per accedere a sorgenti dati esterne (ExternalAPIConfig) o per gestire proprietà relative ai file (DataProperties).

Abbiamo inoltre un package exception, dedicato alla gestione degli errori: qui centralizziamo la gestione delle eccezioni, migliorando così l'affidabilità del sistema e offrendo un'esperienza più stabile a chi utilizza l'applicazione.

In generale, questa struttura ci permette di mantenere il progetto modulare e ben organizzato. Ogni parte ha un ruolo preciso, e questo ci aiuta sia nello sviluppo che nella manutenzione, oltre a facilitare eventuali estensioni future del sistema.

3.2) Architettura di deployment

L'architettura di deployment scelta per **3Dataviz** si basa sull'utilizzo di container Docker orchestrati tramite Docker Compose. Questa decisione strategica permette di incapsulare l'applicazione backend (sviluppata con Spring Boot), il frontend (sviluppato con Svelte) e il database (PostgreSQL) in ambienti isolati, standardizzati e facilmente riproducibili.

3.2.1) Struttura a monolite containerizzato vs microservizi

Sebbene l'architettura logica interna del backend segua il pattern esagonale per promuovere la modularità, a livello di deployment l'applicazione viene distribuita come un'unica unità funzionale backend (assimilabile a un monolite) all'interno di un container Docker, affiancata da un container separato per il frontend. La scelta di questo approccio monolitico containerizzato per il backend,

rispetto a un'architettura a microservizi più granulare, è motivata da diversi fattori specifici del progetto **3Dataviz**, in linea con le considerazioni generali su semplicità e focus:

- **Semplicità e rapidità di Sviluppo**: L'obiettivo primario è realizzare una piattaforma web funzionale per la visualizzazione 3D dei dati. Un approccio monolitico per il backend semplifica il processo di sviluppo, build, testing e debugging, evitando la complessità aggiuntiva introdotta dalla gestione di servizi distribuiti comunicanti via API, permettendo al team di concentrarsi sulle funzionalità principali.
- **Ambito definito e manutenibilità**: **3Dataviz** è focalizzato sulla visualizzazione 3D dei dati. Dato che l'ambito è specifico e non ci sono piani imminenti per grandi espansioni che richiederebbero componenti gestiti separatamente, l'architettura monolitica si rivela un percorso più diretto per lo sviluppo, il debug e la manutenzione del backend. Con un monolite, le modifiche al codice sono naturalmente coese, evitando le complicazioni legate al coordinamento di deploy separati per più servizi o alla gestione della comunicazione tra di essi. Il risultato è un ciclo di sviluppo più snello e un sistema più facile da capire nel suo insieme, il che rende più agevoli anche il debug e la risoluzione dei problemi. Infine, avere il codice backend centralizzato facilita la manutenzione: diventa più semplice individuare e correggere bug, così come implementare nuove funzionalità o miglioramenti.
- **Gestibilità e competenze**: Operativamente, un monolite containerizzato è più semplice da gestire. Aggiornamenti, rollback, monitoraggio e troubleshooting sono più diretti, riguardando un'unica unità applicativa backend, a differenza della complessità distribuita dei microservizi che moltiplica queste attività e richiede pratiche e strumenti più avanzati (es. tracciamento distribuito). Di conseguenza, anche le competenze tecniche richieste sono meno specifiche; i microservizi necessitano di una profonda padronanza di progettazione distribuita, tecnologie cloud-native e pratiche operative complesse. Evitare questa complessità permette al team di **3Dataviz** di concentrarsi sulle funzionalità chiave della visualizzazione 3D e sulla qualità dell'applicazione.

Pur riconoscendo i vantaggi teorici dei microservizi (come scalabilità granulare e resilienza), abbiamo valutato che per lo scenario attuale di **3Dataviz** questi benefici non giustificano l'investimento significativo richiesto in termini di progettazione, infrastruttura e gestione della complessità distribuita. Pertanto, l'approccio monolitico containerizzato per il backend rappresenta la scelta più pragmatica ed efficiente al momento, garantendo la semplicità gestionale e la velocità di sviluppo necessarie per raggiungere gli obiettivi del progetto, senza precludere future evoluzioni architetturali se le esigenze cambieranno.

3.2.2) Docker e Containerizzazione dell'Applicazione

L'adozione della containerizzazione con Docker, orchestrata da Docker Compose, è un pilastro fondamentale della strategia di deployment, scelta per i seguenti vantaggi cruciali applicati al contesto **3Dataviz**:

- **Portabilità**: I container Docker pacchettizzano le applicazioni (Spring Boot backend, Svelte frontend) con tutte le loro dipendenze (runtime Java/Node.js, librerie specifiche) e la configurazione necessaria. Questo garantisce che l'applicazione **3Dataviz** possa essere eseguita in modo identico su qualsiasi macchina o ambiente che supporti Docker, eliminando le problematiche legate alle differenze di configurazione («it works on my machine»).

- **Consistenza Ambientale e Orchestrazione**: Il file `docker-compose.yml` definisce e gestisce l'intero stack applicativo, assicurando coerenza tra gli ambienti e facilitando l'avvio coordinato dei servizi. Definisce nel dettaglio:
  - Un servizio `db` basato su `postgres:17`, con configurazione (utente/password/DB) tramite variabili d'ambiente, persistenza dati su volume nominato (`postgres_data`) e inizializzazione tramite script SQL (`init-data.sql`).
  - Un servizio `test` basato su `maven:3.9.9-eclipse-temurin-23-alpine`, dipendente dal `db`, che esegue i test di integrazione (`mvn clean test`) nel contesto Docker, utilizzando Testcontainers (facilitato dal mount del socket Docker e dalle variabili d'ambiente specifiche) prima che l'applicazione principale venga avviata.
  - Il servizio `app` per il backend Spring Boot, costruito dal suo `Dockerfile`, dipendente da `db` e `test`, che riceve le credenziali del DB via environment ed espone la porta 8080.
  - Il servizio `frontend`, costruito dal suo `Dockerfile` Node.js, dipendente dal servizio `app`, che serve l'interfaccia Svelte sulla porta interna 4173, mappata sulla porta host 5173.

  Questa orchestrazione garantisce che tutti lavorino con la stessa configurazione e che l'ambiente di produzione rispecchi quello di test.
- **Isolamento e Gestione Dipendenze**: Ogni servizio (`db`, `app`, `frontend`, `test`) gira nel proprio container isolato, evitando conflitti di librerie o versioni e confinando le dipendenze all'interno delle rispettive immagini.
- **Efficienza delle Risorse**: I container condividono il kernel del sistema operativo host, risultando più leggeri e efficienti rispetto alle macchine virtuali.
- **Scalabilità Orizzontale (Potenziale)**: Anche se il backend è monolitico, Docker permette di scalare orizzontalmente avviando più istanze del container `app` (e potenzialmente `frontend`) dietro un load balancer, se necessario.

I Dockerfile per backend e frontend utilizzano build multi-stage per ottimizzare le immagini finali. Il **Dockerfile del backend** segue un processo in due fasi:

1. **Fase di compilazione (`builder`)**: utilizza un'immagine `maven:3.9.9-eclipse-temurin-23-alpine` per costruire il pacchetto JAR dell'applicazione usando `mvn package`.
2. **Fase di runtime**: utilizza un'immagine `eclipse-temurin:23-jdk-alpine` più leggera, copiando solo il JAR compilato dalla fase precedente per eseguirlo.

Analogamente, il **Dockerfile del frontend** usa `node:20-alpine` con una fase per installare dipendenze e buildare l'applicazione (`npm run build`) e una fase finale per servire i file generati (`npm run preview`).

In sintesi, la combinazione di un backend monolitico e un frontend separato, entrambi containerizzati insieme al database e a un ambiente di test integrato tramite Docker e Docker Compose, offre per **3Dataviz** un'architettura di deployment robusta, portabile, consistente e gestibile, bilanciando efficacemente le esigenze del progetto con l'efficienza operativa e di sviluppo.

3.3) Database

Nel nostro progetto utilizziamo PostgreSQL come database relazionale per gestire i dati, sfruttando la sua affidabilità, la possibilità di usare tipi personalizzati e il supporto avanzato per le query. Fin dall'avvio, il database è già popolato con un set di dati predefinito, organizzato nella tabella coordi-

nates. Questa tabella raccoglie le label x, label z, i valori y dati dall'incrocio di x e z e le classificazioni, utilizzando un tipo ENUM personalizzato (dataset_level) per definire il livello del dataset (SMALL, MEDIUM, LARGE), garantendo così una struttura più tipizzata e robusta rispetto all'uso di semplici stringhe.

Dal punto di vista applicativo, l'accesso al database è limitato a operazioni di sola lettura ed è completamente incapsulato nel package repository, in linea con i principi dell'architettura esagonale. La classe CoordinateRepository, che estende JpaRepository, offre un metodo personalizzato per recuperare i dati in base al livello del dataset, permettendo così di filtrare le coordinate e restituire solo quelle che soddisfano un certo livello minimo di granularità.

Questo approccio assicura un'interazione controllata ed efficiente con il database, separando nettamente la logica di accesso ai dati da quella di business. Inoltre, avere già a disposizione i dati predefiniti all'avvio semplifica notevolmente il processo di inizializzazione, rendendo più rapido il testing, la validazione e la visualizzazione dei risultati.

4) Front-end

4.1) Utilities

In questa sezione vengono documentate in dettaglio le utilities e i moduli di supporto utilizzati nell'applicazione. Questi file gestiscono la preparazione, il calcolo e la distribuzione dei dati e delle impostazioni globali, fornendo le basi per il funzionamento dell'interfaccia 3D e dei filtri. Per ciascuna utility verranno illustrati i seguenti aspetti:

- **Descrizione**: una breve spiegazione del ruolo e dello scopo della utility all'interno dell'applicazione.
- **Struttura e Funzionalità**: la composizione del file, evidenziando le funzioni chiave e la logica implementata per elaborare i dati e gestire lo stato globale.

4.1.1) index.svelte.ts

4.1.1.1) Descrizione

Questo file è un modulo per la gestione dello stato e la fornitura di dati computati per l'applicazione. Esso definisce i dati grezzi, ne calcola le proprietà (come la media, i valori minimi e massimi, e le dimensioni della matrice) e li espone tramite funzioni ed oggetti reattivi. Inoltre, il file gestisce la selezione degli elementi tramite la classe Selection e definisce un oggetto di filtro che raccoglie le impostazioni di visualizzazione e filtraggio.

4.1.1.2) Struttura e Funzionalità

- **Struttura**:
  - Vengono importati moduli fondamentali da **three** e funzioni di stato reattivo da Svelte.
  - Definisce una variabile `fetchedData` che contiene una matrice di dati grezzi.
  - Utilizza una store derivata per creare l'oggetto `data`, che include sia i valori originali che le proprietà computate, come:
    - La media dei valori,
    - I valori minimo e massimo,
    - Il numero di righe e colonne,
    - Il target predefinito per la visualizzazione e la posizione di default della camera.
  - Esporta la funzione `getData()`, che restituisce l'oggetto `data`, e la funzione `getValueFromId()`, utile per estrarre un valore dalla matrice in base a un identificatore.
  - Definisce la classe **Selection** che gestisce la selezione degli elementi con metodi per aggiungere, rimuovere, verificare e alternare gli elementi selezionati.
  - Crea un'istanza di **Selection** e definisce un oggetto di filtro che include impostazioni quali lo spacing, il range di valori, le opzioni di colorazione, i filtri per la media e le barre, e il flag per il display del filtro.
- **Funzionalità**:
  - Calcola in modo reattivo proprietà importanti dai dati grezzi per normalizzare e posizionare gli elementi 3D.
  - Fornisce un meccanismo per gestire la selezione degli elementi, utile per applicare filtri e evidenziare barre specifiche nel grafico.

- Esporta uno stato globale (filter) che raccoglie tutte le impostazioni di filtraggio e visualizzazione, da utilizzare nei componenti dell'applicazione.

4.1.1.3) Props e Variabili Reattive

- **Variabili Reattive:**
  - **fetchedData:** contiene i dati grezzi iniziali.
  - **data:** store derivata che combina i dati grezzi con le proprietà computate (media, min, max, righe, colonne, defaultTarget e defaultPosition).
  - **utils:** raccoglie le proprietà computate, utili per il posizionamento e il filtraggio.
- **Oggetto di Filtro:**
  - L'oggetto filter include impostazioni quali:
    - spacing,
    - rangeValue (min e max),
    - colorSelection,
    - avgFilter e avgEnabled,
    - barFilterSelection,
    - displayBarFilter,
    - selection (istanza della classe Selection).

4.1.1.4) Eventi e Comunicazione

- Pur non gestendo direttamente eventi, questo modulo fornisce le funzioni e gli oggetti di stato che altri componenti importano per comunicare e sincronizzare i dati e le impostazioni a livello globale.

4.1.1.5) Stili e Layout

- Non applica stili, in quanto funge esclusivamente da modulo di gestione dei dati e dello stato.

4.1.1.6) Esempi di Utilizzo

- In altri componenti, come App.svelte o Bar.svelte, il modulo viene importato per accedere ai dati e alle impostazioni.

```
import { getData, filter, getValueFromId } from '$lib/index.svelte';
let data = getData();
```

- Questo esempio evidenzia come il modulo fornisca le basi per il calcolo dei dati e la gestione dei filtri che vengono poi utilizzati per aggiornare dinamicamente la visualizzazione del grafico 3D.

4.1.1.7) Dipendenze Esterne

- **three:** Utilizzato per operazioni matematiche e vettoriali (ad es. Vector3) necessarie per il calcolo delle proprietà della scena 3D.
- **svelte (State Management):** Le funzioni state (state, derived, effect) sono impiegate per gestire la reattività e aggiornare automaticamente i dati e le proprietà computate.
- **lib/index.svelte:** Il modulo stesso funge da fonte centrale per le funzioni getData, getValueFromId e per la gestione dei filtri tramite l'oggetto filter.

Questa descrizione strutturata fornisce una panoramica completa del file index.svelte.ts, evidenziando come esso gestisca la preparazione e la distribuzione dei dati e delle impostazioni globali, e fungendo da riferimento essenziale per gli sviluppatori e i manutentori dell'applicazione.

4.2) Componenti

In questa sezione vengono documentati in dettaglio i componenti front-end[G] sviluppati in Svelte per il prodotto. Ogni file .svelte rappresenta un componente autonomo che implementa una specifica funzionalità dell' interfaccia utente[G] . Per ciascun componente verranno illustrati i seguenti aspetti:

- **Descrizione**: una breve spiegazione del ruolo e dello scopo del componente all'interno dell'applicazione.
- **Struttura e Funzionalità**: la composizione del componente, evidenziando gli elementi chiave e la logica implementata.
- **Props e Variabili Reattive**: i dati in ingresso (props) e le variabili reattive che gestiscono lo stato interno.
- **Eventi e Comunicazione**: come il componente comunica con altri elementi, tramite eventi custom o binding.
- **Stili e Layout**: le regole CSS o l'utilizzo di framework di styling adottati per garantire un'interfaccia coerente.
- **Esempi di Utilizzo**: un esempio pratico su come il componente viene importato e integrato nell'applicazione.
- **Dipendenze Esterne**: eventuali librerie aggiuntive utilizzate e spunti per ottimizzazioni future.

Questa sezione permette di comprendere il funzionamento e l'interazione dei vari componenti, fornendo un riferimento utile per sviluppatori e manutentori.

4.2.1) App.svelte

4.2.1.1) Descrizione

Il componente App.svelte è il punto di ingresso dell'applicazione e coordina il rendering della scena 3D e dei pannelli di controllo. Gestisce il canvas principale, carica i dati tramite la funzione getData() e imposta il target della visualizzazione, offrendo inoltre la possibilità di resettare il target alla configurazione di default.

4.2.1.2) Struttura e Funzionalità

- **Struttura**:
  - Il componente è contenuto in un `<div>` che occupa l'intera finestra, con un background scuro per evidenziare la scena 3D.
  - All'interno del `<Canvas>` (fornito da threlte/core) vengono integrati tre componenti principali:
    - SettingsPane.svelte: pannello per il controllo delle impostazioni e per il reset del target.
    - BarPane.svelte: componente che gestisce i filtri e la selezione delle barre.
    - Scene.svelte: responsabile del rendering della scena 3D in base al target impostato.
- **Funzionalità**:
  - Recupera i dati e le configurazioni computate tramite getData() e li memorizza in una variabile derivata, utils.

    - Imposta il target della visualizzazione con il valore default ottenuto da getData().computed.defaultTarget e lo aggiorna in modo reattivo.
    - Fornisce la funzione resetTarget() che ripristina il target al valore predefinito in utils.

4.2.1.3) Props e Variabili Reattive

- **Props**:
    - I componenti SettingsPane.svelte, BarPane.svelte e Scene.svelte ricevono tramite props i dati e le funzioni necessari per il loro corretto funzionamento.
    - In particolare, Scene.svelte riceve il prop target, fondamentale per centrare la visualizzazione 3D.
- **Variabili reattive**:
    - **utils**: variabile derivata (derived) che contiene i valori computati ottenuti da getData().computed.
    - **target**: stato reattivo (state) inizializzato al valore predefinito e aggiornato dalla funzione resetTarget().

4.2.1.4) Eventi e Comunicazione

- Il componente comunica con i suoi figli tramite binding delle props:
    - SettingsPane.svelte utilizza resetTarget() per permettere agli utenti di ripristinare il target della camera.
    - BarPane.svelte e Scene.svelte ricevono i dati e le impostazioni necessari per sincronizzare il rendering della scena 3D e la gestione dei filtri.

4.2.1.5) Stili e Layout

- Il componente è avvolto in un `<div>` che imposta:
    - Posizione: relative
    - Dimensioni: height: 100vh e width: 100vw
    - Background: rgb(14, 22, 37) per creare un ambiente visivo immersivo che esalta gli elementi 3D.

4.2.1.6) Esempi di Utilizzo

- Integrazione nell'applicazione: Il componente App.svelte viene utilizzato come punto di ingresso principale.

```
<script>
  import App from './App.svelte';
</script>

<App />
```

4.2.1.7) Dipendenze Esterne

- **threlte/core**: fornisce il componente Canvas e le funzionalità per il rendering 3D integrando Three.js in Svelte.
- **lib/index.svelte**: contiene la funzione getData(), che restituisce i dati e le configurazioni computate necessarie per inizializzare lo stato del componente.
- Svelte State Management: utilizza state e derived per gestire lo stato reattivo e derivare i valori computati nel componente.

4.2.2) Bar.svelte

4.2.2.1) Descrizione

Il componente Bar.svelte rappresenta una singola barra del grafico 3D. Esso visualizza un dato specifico mediante la sua altezza e il colore dinamico, rispondendo alle interazioni dell'utente (hover e click) per applicare filtri e gestire la selezione.

4.2.2.2) Struttura e Funzionalità

- **Struttura**:
  - Il componente utilizza un elemento T.Mesh per creare il mesh della barra, basato su una BoxGeometry e un MeshStandardMaterial, che gestisce trasparenza e colorazione dinamica.
  - Un elemento Text, posizionato sopra la barra, mostra il valore numerico della barra.
- **Funzionalità**:
  - Calcola l'opacità della barra in base a condizioni di filtraggio: se l'altezza rientra nell'intervallo definito e rispetta i filtri (basati sulla media e sui filtri specifici per le barre), l'opacità è impostata a 1, altrimenti a 0.2.
  - Utilizza un raycaster per rilevare le interazioni del mouse e applica un effetto hover mediante un tween definito con durata ed easing personalizzati.
  - La funzione getBarColor determina il colore della barra in base alla selezione del criterio di colorazione (per righe, per colonne o in base al valore normalizzato).

4.2.2.3) Props e Variabili Reattive

- **Props**:
  - **id**: identificatore univoco della barra, utilizzato per la gestione delle selezioni.
  - **coordinates**: posizione 3D della barra.
  - **height**: valore che determina l'altezza della barra.
  - **currentCameraQuaternionArray**: array che rappresenta l'orientamento corrente della camera, usato per orientare il testo.
- **Variabili reattive**:
  - **utils**: derivata contenente dati calcolati tramite `getData().computed`.
  - **inRange**: derivata che verifica se l'altezza rientra nei limiti definiti da `filter.rangeValue`.
  - **passesFilter**: calcola, in modo derivato, se la barra soddisfa ulteriori condizioni di filtro (in base a media e selezione).
  - **opacity**: impostata in modo derivato in base alla combinazione dei filtri `inRange` e `passesFilter`.
  - **mesh e text**: riferimenti allo stato dei componenti `T.Mesh` e `Text` per applicare gli effetti di interazione.

4.2.2.4) Eventi e Comunicazione

- Il componente gestisce eventi di pointer (movimento e uscita) per attivare o disattivare l'effetto hover, modificando il valore del tween hover.
- Gli eventi click, applicati sia al mesh della barra che al testo, invocano metodi di toggle o set sulla proprietà `filter.selection`, aggiornando lo stato di selezione.
- La comunicazione con il componente genitore avviene tramite il binding delle props essenziali (id, coordinates, height), mentre i filtri sono ora gestiti centralmente dal modulo `filter`.

4.2.2.5) Stili e Layout

- La posizione della barra è determinata dalle props coordinates e height, che definiscono il posizionamento 3D e la scala del mesh.
- Il componente Text è posizionato sopra la barra (offset in altezza) e ruotato in modo da garantire la leggibilità, in funzione dell'orientamento della camera.

4.2.2.6) Esempi di Utilizzo

- Il componente Bar.svelte viene utilizzato all'interno di un ciclo, dove per ogni elemento della matrice di dati viene creato un componente Bar.

```
{#each data as row, rowIndex}
  {#each row as height, colIndex}
    <Bar
      id={`${rowIndex}-${colIndex}`}
      coordinates={[
        rowIndex * spacing,
        height / 2,
        colIndex * spacing
      ]}
      {height}
      {currentCameraQuaternionArray}
    />
  {/each}
{/each}
```

4.2.2.7) Dipendenze Esterne

- **threlte/core**: Fornisce il framework per la creazione degli elementi 3D, come T.Mesh, BoxGeometry e MeshStandardMaterial.
- **threlte/extras**: Offre il componente Text e le funzioni per abilitare l'interattività nella scena.
- **three**: Libreria base per la manipolazione degli oggetti 3D, utilizzata per Raycaster, Vector2 e per operazioni vettoriali.
- **svelte/motion** e **svelte/easing**: Utilizzate per creare animazioni fluide (tweening) per l'effetto hover.
- **three/tsl**: Fornisce, se necessario, funzioni aggiuntive per la gestione delle selezioni.
- **lib/index.svelte**: modulo contenente `getData` e `filter`, che centralizza la gestione dei filtri applicati al grafico

4.2.3) BarPane.svelte

4.2.3.1) Descrizione

Il componente BarPane.svelte gestisce il filtro di selezione per le barre del grafico 3D. Se l'opzione di visualizzazione del filtro è attiva, viene mostrato un pannello fisso che contiene quattro pulsanti, ciascuno dei quali imposta un criterio di filtro diverso per le barre (solo la barra selezionata, barre con valori maggiori, barre con valori minori, o reset del filtro).

4.2.3.2) Struttura e Funzionalità

- **Struttura:**
  - Il componente importa i componenti Pane e Button dalla libreria **svelte-tweakpane-ui**.

    - Utilizza un controllo condizionale per rendere il pannello solo se la proprietà filter.displayBarFilter è attiva.
    - All'interno del pannello vengono resi quattro pulsanti, ognuno con una specifica etichetta e azione associata.
- **Funzionalità:**
    - Ogni pulsante, tramite l'evento on:click, modifica direttamente il valore di filter.barFilterSelection, impostando il criterio di filtraggio per le barre del grafico.
    - Il pulsante «Filter reset» ripristina il filtro annullando ogni criterio precedentemente applicato.

4.2.3.3) Props e Variabili Reattive

- **Props:**
    - Il componente utilizza l'oggetto filter importato da lib/index.svelte, che contiene le impostazioni per il filtraggio delle barre.
- **Variabili reattive:**
    - **filter.displayBarFilter**: determina se il pannello di filtro deve essere visualizzato.
    - **filter.barFilterSelection**: viene aggiornato in base al pulsante cliccato e controlla il criterio di filtro applicato al grafico.

4.2.3.4) Eventi e Comunicazione

- Il componente gestisce gli eventi on:click per ciascun pulsante, aggiornando il valore di filter.barFilterSelection.
- Queste modifiche vengono propagate al sistema, in modo che il grafico 3D si aggiorni dinamicamente in base al filtro selezionato.

4.2.3.5) Stili e Layout

- Il pannello è posizionato in maniera fissa (position fixed) con coordinate specifiche (y = 210, x = 120), garantendo che rimanga visibile nell'interfaccia utente.
- I pulsanti ereditano gli stili predefiniti della libreria **svelte-tweakpane-ui**, assicurando un aspetto coerente con il resto dell'interfaccia.

4.2.3.6) Esempi di Utilizzo

- Il componente BarPane.svelte viene renderizzato condizionatamente nel layout principale dell'applicazione quando il filtro delle barre è attivo.

```
<div>
  <Canvas>
    <BarPane />
  </Canvas>
</div>
```

4.2.3.7) Dipendenze Esterne

- **svelte-tweakpane-ui**: Fornisce i componenti Pane e Button, essenziali per la creazione dell'interfaccia di filtro.
- **lib/index.svelte**: Fornisce l'oggetto filter, che contiene le impostazioni e i metodi per la gestione dei filtri e delle selezioni.

4.2.4) CameraSettings.svelte

4.2.4.1) Descrizione

Il componente CameraSettings.svelte fornisce i controlli per regolare la posizione della camera nella scena 3D. Permette all'utente di effettuare lo zoom in, lo zoom out e di resettare la posizione della camera alla configurazione predefinita, influenzando direttamente la visualizzazione dell'ambiente 3D.

4.2.4.2) Struttura e Funzionalità

- **Struttura:**
  - Il componente importa il Button dalla libreria **svelte-tweakpane-ui** e utilizza il hook useThrelte per accedere al riferimento della camera.
  - Utilizza una variabile zoomValue e una costante zoomStep per controllare l'incremento o il decremento dello zoom.
  - Le funzioni zoomIn() e zoomOut() aggiornano zoomValue e richiamano updateCamera() per modificare la posizione della camera lungo la sua direzione attuale.
  - La funzione resetPosition() ripristina la posizione della camera al valore default ottenuto da utils.defaultPosition e invoca la funzione resetTarget() passata tramite props.
- **Funzionalità:**
  - **Zoom In**: diminuisce zoomValue e sposta la camera in avanti lungo la sua direzione corrente.
  - **Zoom Out**: aumenta zoomValue e sposta la camera indietro lungo la stessa direzione.
  - **Reset**: ripristina la posizione della camera e aggiorna il target della visualizzazione.

4.2.4.3) Props e Variabili Reattive

- **Props:**
  - **resetTarget**: funzione passata tramite props per ripristinare il target della camera.
- **Variabili reattive:**
  - **utils**: variabile derivata che contiene i valori computati ottenuti da getData().computed, inclusa la posizione default della camera.
  - **zoomValue**: stato numerico che controlla il livello di zoom corrente.
  - **zoomStep**: costante che definisce l'incremento/decremento per ogni operazione di zoom.

4.2.4.4) Eventi e Comunicazione

- Il componente gestisce gli eventi on:click sui pulsanti:
  - Il pulsante «Zoom In» invoca la funzione zoomIn(), spostando la camera in avanti.
  - Il pulsante «Zoom Out» invoca la funzione zoomOut(), spostando la camera indietro.
  - Il pulsante «Resetta» invoca resetPosition(), ripristinando la posizione della camera al valore di default e aggiornando il target.
- La comunicazione con il componente genitore avviene tramite la funzione resetTarget() e il binding dello stato della camera, garantendo un aggiornamento dinamico della visualizzazione.

4.2.4.5) Stili e Layout

- Il componente utilizza i pulsanti forniti da **svelte-tweakpane-ui**, che mantengono uno stile coerente con il resto dell'interfaccia.

- Non sono specificati ulteriori stili custom; lo styling è gestito dalla libreria e dal tema globale dell'applicazione.

4.2.4.6) Esempi di Utilizzo

- Integrazione in un pannello di controllo: Il componente viene inserito all'interno di un pannello di impostazioni (ad es. in SettingsPane.svelte) per consentire agli utenti di regolare la posizione della camera.

```
<TabPage title="Camera options">
      <CameraSettings {resetTarget} />
</TabPage>
```

- Questo esempio mostra come il componente permetta di controllare lo zoom e di ripristinare la vista della scena 3D.

4.2.4.7) Dipendenze Estere

- **threlte/core:** Fornisce il hook useThrelte e il contesto necessario per accedere alla camera e agli altri elementi della scena 3D.
- **svelte-tweakpane-ui:** Fornisce i componenti Button per la creazione dei controlli dell'interfaccia.
- **three:** Utilizzato per la gestione di vettori nello spazio 3D (Vector3) e per operazioni geometriche sulla posizione della camera.
- **lib/index.svelte:** (indiretto) Fornisce la funzione getData() per ottenere le configurazioni computate necessarie per inizializzare il valore default della camera.

4.2.5) Chart.svelte

4.2.5.1) Descrizione

Il componente Chart.svelte è responsabile del rendering della visualizzazione 3D del grafico. Esso gestisce la costruzione della scena, inclusa la griglia di fondo, il piano medio (quando abilitato), le etichette per righe e colonne e la generazione dinamica delle barre 3D tramite il componente Bar.svelte. Il componente aggiorna inoltre in tempo reale l'orientamento della camera per garantire che le etichette siano sempre leggibili.

4.2.5.2) Struttura e Funzionalità

- **Struttura:**
  - Il componente si struttura all'interno di un elemento T.Group, che raggruppa tutti gli elementi 3D della scena.
  - Viene renderizzata una Mesh che funge da griglia di fondo, creata con PlaneGeometry e MeshStandardMaterial, posizionata centralmente in base al numero di righe, colonne e allo spacing.
  - Se il flag `filter.avgEnabled` è attivo, viene renderizzato un piano medio (Mesh) posizionato a livello della media dei dati, con un materiale semitrasparente.
  - Le etichette delle righe e delle colonne vengono generate utilizzando il componente Text, posizionato e ruotato per garantire la leggibilità nell'ambiente 3D.
  - Infine, il componente itera sui dati per creare, per ogni elemento della matrice, un componente Bar.svelte che visualizza la barra corrispondente.
- **Funzionalità:**

- Calcola dinamicamente le dimensioni della griglia (rows, cols) e normalizza i valori dei dati per definire l'altezza delle barre.
- Utilizza la funzione `truncateText` per abbreviare le etichette se troppo lunghe.
- Aggiorna in modo reattivo i dati, i valori computati (`utils`), lo spacing e il target della scena, basandosi sui dati ottenuti tramite `getData()`.
- Monitora lo stato di `filter.displayBarFilter` aggiornando in base allo stato della selezione.

4.2.5.3) Props e Variabili Reattive

- **Props**:
  - Nessuna props per i filtri, in quanto ora vengono gestiti direttamente tramite `filter`.
- **Variabili reattive**:
  - **dt** derivato da `getData()`, contiene i dati grezzi e i valori computati.
  - **data**: matrice dei dati numerici, usata per definire le altezze e il posizionamento delle barre.
  - **utils**: derivata dai dati computati tramite `getData()`, contiene informazioni come la media e le dimensioni della griglia.
  - **spacing**: valore derivato da `filter.spacing` che determina la distanza tra le barre.
  - **rows e cols**: ottenuti da utils e rappresentano il numero di righe e colonne della matrice data.
  - **currentCameraQuaternionArray**: array che conserva l'orientamento corrente della camera, aggiornato continuamente per sincronizzare le etichette.

4.2.5.4) Eventi e Comunicazione

- Il componente utilizza `onMount` per avviare un ciclo di aggiornamento che cattura l'orientamento della camera (quaternion) in tempo reale, utilizzando `requestAnimationFrame`.
- Su `onDestroy`, l'animazione viene interrotta per liberare le risorse.
- Le variabili reattive vengono propagate ai componenti figli, in particolare a Bar.svelte, per sincronizzare la visualizzazione delle barre.

4.2.5.5) Stili e Layout

- Gli elementi 3D sono organizzati all'interno di T.Group per mantenere una struttura gerarchica chiara della scena.
- La griglia (Mesh con PlaneGeometry) e il piano medio sono centrati in base a rows, cols e spacing, garantendo un layout bilanciato.
- Le etichette (Text) sono posizionate e ruotate in modo da essere sempre leggibili, indipendentemente dall'orientamento della camera.

4.2.5.6) Esempi di Utilizzo

- Il componente Chart.svelte viene solitamente integrato all'interno della scena principale dell'applicazione per visualizzare il grafico 3D.

```
<Chart />
```

4.2.5.7) Dipendenze Esterne

- **threlte/core**:
  - Fornisce il framework per il rendering 3D, compreso il componente T.Group e altri elementi base come Mesh e PlaneGeometry.

- **threlte/extras**:
  - Offre il componente Text per la creazione di etichette 3D e funzioni per l'interattività.
- **three**:
  - Libreria fondamentale per la manipolazione degli oggetti 3D, utilizzata per operazioni come il calcolo delle dimensioni della griglia e per il raycasting.
- **svelte**:
  - Gestisce il ciclo di vita del componente tramite `onMount` e `onDestroy`, e supporta la reattività attraverso state, derived ed effect.
- **lib/index.svelte**:
  - Fornisce la funzione `getData()` e l'oggetto `filter`, che contengono i dati, le configurazioni computate e le impostazioni di filtraggio per la scena.

4.2.6) Color.svelte

4.2.6.1) Descrizione

Il componente **Color.svelte** fornisce un'interfaccia per la selezione del criterio di colorazione utilizzato nella visualizzazione 3D. Utilizza un controllo List della libreria **svelte-tweakpane-ui** per offrire opzioni predefinite come colorazione per righe, colonne o valori. Il valore selezionato è ora gestito tramite l'oggetto filter importato da `$lib/index.svelte`.

4.2.6.2) Struttura e Funzionalità

- **Struttura**:
  - Il componente importa e utilizza il componente `List` da **svelte-tweakpane-ui**, che fornisce un'interfaccia utente sotto forma di un menu a tendina per la selezione di un'opzione.
  - L'opzione selezionata è collegata a `filter.colorSelection`, che gestisce lo stato centralizzato del filtro.
  - L'oggetto `options` definisce le modalità di selezione disponibili per la colorazione.
- **Funzionalità**:
  - Ogni volta che l'utente cambia la selezione nel menu, il valore di `colorSelection` si aggiorna automaticamente grazie al binding bidirezionale, riflettendo la scelta dell'utente.
  - Il valore selezionato viene visualizzato in tempo reale all'interno di un elemento `<pre>`, che permette di visualizzare il numero corrispondente all'opzione scelta, fornendo un feedback immediato all'utente.

4.2.6.3) Props e Variabili Reattive

- **Props**:
  - **filter.colorSelection**: ora il valore selezionato è gestito attraverso l'oggetto filter importato, invece che come prop indipendente..
- **Variabili reattive**:
  - **options**: oggetto che contiene le possibili scelte per la modalità di colorazione.

4.2.6.4) Eventi e Comunicazione

- L'uso di bind:value={filter.colorSelection} permette al componente di aggiornare direttamente lo stato del filtro colore all'interno di `$lib/index.svelte`.
- Il valore selezionato viene immediatamente riflesso nella UI e nello stato centralizzato.

4.2.6.5) Stili e Layout

- Il componente non definisce stili personalizzati, poiché si affida alla libreria **svelte-tweakpane-ui** per la gestione dell'interfaccia utente.

4.2.6.6) Esempi di Utilizzo

Esempio di integrazione:

```
<TabPage title="Color filter">
      <Color />
</TabPage>
```

Nell'esempio il componente è incluso come una delle schede (TabPage) all'interno del pannello delle impostazioni, e permette di modificare il tipo di colore utilizzato nell'applicazione.

4.2.6.7) Dipendenze Esterne

- **svelte-tweakpane-ui:** fornisce il componente List per la selezione delle opzioni in modo strutturato e interattivo.
- **lib/index.svelte**: fornisce la gestione centralizzata dello stato del filtro colore.

4.2.7) DataFilter.svelte

4.2.7.1) Descrizione

Il componente **DataFilter.svelte** permette di filtrare i dati in base a intervalli specifici e valori relativi alla media. Gestisce un intervallo visibile di dati e consente all'utente di filtrare i valori inferiori o superiori alla media globale e include un'opzione per visualizzare il piano della media.

4.2.7.2) Struttura e Funzionalità

- **Struttura:**
  - Utilizza un `IntervalSlider` per consentire la selezione di un intervallo di valori da visualizzare.
  - Fornisce un `Checkbox` per attivare o disattivare la visualizzazione del piano della media.
  - Fornisce tre `Button`:
    - Uno per filtrare i **valori inferiori alla media**.
    - Uno per filtrare i **valori superiori alla media**.
    - Uno per **resettare** il filtro e riportare i valori all'intervallo completo.
- **Funzionalità:**
  - I valori minimi e massimi dell'intervallo vengono ottenuti dinamicamente tramite la funzione `getData().computed` e assegnati a `filter.rangeValue`.
  - Il `Checkbox` consente di attivare o disattivare la visualizzazione del piano della media attraverso la variabile `filter.avgEnabled`.
  - La variabile `value` controlla l'intervallo visualizzato.
  - I pulsanti modificano il valore di `filter.avgFilter`, che definisce i valori da visualizzare in base alla media globale.
  - Il reset del filtro riporta i valori di `filter.rangeValue` ai limiti minimi e massimi calcolati dinamicamente.

4.2.7.3) Props e Variabili Reattive

- **Props:**

    - **filter.rangeValue**: oggetto che contiene i valori minimo e massimo per l'intervallo di visualizzazione, aggiornato dinamicamente in base ai dati.
    - **filter.avgFilter**: variabile che memorizza il tipo di filtro applicato (0 = nessun filtro, 1 = sotto media, 2 = sopra media).
    - **filter.avgEnabled**: booleano che abilita/disabilita la visualizzazione del piano della media.
- **Variabili reattive:**
    - **utils:** contiene i dati calcolati dinamicamente tramite `getData().computed`.

4.2.7.4) Eventi e Comunicazione

- Il `Checkbox` modifica il valore di `filter.avgEnabled`, permettendo di attivare/disattivare la visualizzazione del piano della media.
- I Button aggiornano `filter.avgFilter` per applicare i filtri o resettare l'intervallo.
- L'intervallo di valori può essere modificato tramite il `IntervalSlider`, con il valore legato reattivamente a `filter.rangeValue`.

4.2.7.5) Stili e Layout

- Non sono specificati stili personalizzati. Viene utilizzato **svelte-tweakpane-ui** per il layout e l'interfaccia utente del filtro e dei pulsanti.

4.2.7.6) Esempi di Utilizzo

Esempio di integrazione:

```
<TabPage title="Data filter">
      <DataFilter />
</TabPage>
```

Nell'esempio il componente **DataFilter** viene utilizzato in una TabPage all'interno di un'interfaccia a schede (Tab Group), permettendo la gestione dei filtri dei dati nell'applicazione.

4.2.7.7) Dipendenze Esterne

- **svelte-tweakpane-ui:** non sono specificati stili personalizzati. Viene utilizzato **svelte-tweakpane-ui** che fornisce i componenti `IntervalSlider`, `Checkbox` e `Button` per la selezione e gestione dei filtri.
- **lib/index.svelte**: fornisce la gestione dei dati e delle variabili di filtro tramite `filter` e `getData().computed`.

4.2.8) Scene.svelte

4.2.8.1) Descrizione

Il componente **Scene.svelte** è responsabile per la visualizzazione della scena 3D, includendo la gestione della fotocamera, delle luci, e della rappresentazione grafica dei dati attraverso il componente **Chart**. Permette l'interazione tramite controlli di orbita e visualizza i dati in un contesto 3D dinamico.

4.2.8.2) Struttura e Funzionalità

- **Struttura:**
    - **Camera e controlli:**

    - Utilizzo di una `PerspectiveCamera` per visualizzare la scena 3D da una posizione predefinita.
    - `OrbitControls` per abilitare l'interazione dell'utente con la scena, includendo il damping e la rotazione automatica.
    - `Gizmo` per visualizzare gli assi e migliorare la comprensione spaziale della scena.
  - **Luci:** due luci direzionali e una luce ambientale, per creare un'illuminazione dinamica e realistica nella scena.
  - **Componente grafico:** `Chart` è il componente visualizza i dati in un grafico 3D, permettendo l'interazione.

- **Funzionalità:**
  - La fotocamera `PerspectiveCamera` viene utilizzata per configurare la visualizzazione della scena da un'angolazione iniziale predefinita, con la possibilità di manipolare la scena grazie agli `OrbitControls`.
  - `OrbitControls` permette di interagire con la scena tramite il mouse, consentendo operazioni come zoom, panoramica e rotazione.
  - Il `Gizmo` fornisce una rappresentazione visiva degli assi XYZ, migliorando l'interazione e la comprensione della scena da parte dell'utente.

4.2.8.3) Props e Variabili Reattive

- **Props:**
  - **target**: posizione della fotocamera, passata dinamicamente per centrare la scena 3D.

- **Variabili reattive:**
  - **autoRotate:** flag (settato su `false`) che abilita o disabilita la rotazione automatica della scena.

4.2.8.4) Eventi e Comunicazione

- Il componente **Scene.svelte** riceve il parametro `target` tramite props e lo passa al componente `OrbitControls` per manipolare la posizione della fotocamera..
- `OrbitControls` consente di interagire con la scena tramite il mouse per zoom, rotazione e panoramica.

4.2.8.5) Stili e Layout

- Il layout della scena è gestito tramite il sistema di **Threlte** e **Three.js**, con una configurazione di luci e camera integrata. Non sono presenti stili CSS personalizzati nel componente.

4.2.8.6) Esempi di Utilizzo

Esempio di integrazione:

```
<div>
  <Canvas>
    <Scene {target} />
  </Canvas>
</div>
```

L'esempio utilizzato in App.svelte mostra l'integrazione del componente **Scene** tramite il tag `<Scene>` all'interno del `<Canvas>` per rendere la scena 3D interattiva e visibile nell'applicazione.

4.2.8.7) Dipendenze Esterne

- **threlte/core**: gestisce la scena 3D, la fotocamera e il rendering.
- **threlte/extras**: fornisce i controlli `OrbitControls` e il `Gizmo` per la manipolazione della scena.
- **Three.js**: usato per la gestione di oggetti 3D come la fotocamera e le luci.

4.2.9) SettingsPane.svelte

4.2.9.1) Descrizione

Il componente **SettingsPane.svelte** gestisce la sezione di impostazioni dell'applicazione, organizzando diverse opzioni tramite una serie di schede. Fornisce agli utenti il controllo su vari parametri come le impostazioni della fotocamera, la fonte dei dati, i filtri, e la selezione del colore.

4.2.9.2) Struttura e Funzionalità

- **Struttura**:
  - Utilizza il componente `Pane` di **svelte-tweakpane-ui** per creare un pannello fisso con un gruppo di schede (TabGroup).
  - Ogni **scheda** (TabPage) contiene un componente separato per gestire un gruppo di impostazioni, tra cui:
    - `CameraSettings` per il controllo della posizione della fotocamera
    - `DataSource` per la gestione della fonte dei dati
    - `DataFilter` per il controllo dei filtri sui dati
    - `Color` per selezionare il tipo di colorazione
- **Funzionalità**:
  - Le schede offrono un'interfaccia interattiva per modificare e configurare parametri come la posizione della fotocamera, i filtri sui dati e la colorazione.
  - Sincronizzazione dei dati tra il componente e il resto dell'app tramite binding bidirezionale, permettendo che le modifiche alle impostazioni influenzino la visualizzazione dei dati in tempo reale.

4.2.9.3) Props e Variabili Reattive

- **Props**:
  - **resetTarget**: variabile passata da componenti esterni, utilizzata per ripristinare la posizione della fotocamera.
- **Variabili reattive**:
  - **bindable**: per legare variabili reattive come `mediaFilter`, `colorSelection`, `rangeValue`, `avgEnabled` a proprietà e parametri specifici, aggiornando automaticamente la visualizzazione quando cambiano.

4.2.9.4) Eventi e Comunicazione

- Il componente comunica con gli altri tramite le props, passando la variabile `resetTarget` al componente `CameraSettings`.

4.2.9.5) Stili e Layout

- Il pannello delle impostazioni ha una posizione fissa, impostato tramite la proprietà `position="fixed"`.

- Le schede sono organizzate in un layout tabellare tramite `TabGroup`, con ciascuna scheda (`TabPage`) che consente l'accesso a diverse configurazioni.

4.2.9.6) Esempi di Utilizzo

Esempio di integrazione:

```
<div>
  <Canvas>
    <SettingsPane {resetTarget} />
  </Canvas>
</div>
```

Nel'"esempio, il componente **SettingsPane** viene usato all"interno di un , con il parametro `resetTarget` passato tramite props.

4.2.9.7) Dipendenze Esterne

- **svelte-tweakpane-ui:** libreria per creare interfacce utente con pannelli di configurazione avanzati.
- **ThemeUtils:** usato per applicare temi globali e predefiniti.

5) Back-end

Come indicato nella sezione Sezione 3.1, il pattern architetturale utilizzato è quello *esagonale*, viste le funzionalità, la robustezza e la manutenibilità che il prodotto software necessita. In questa parte della *Specifica tecnica* si motivano le scelte implementative effettuate nel lato back-end, riportando esempi di codice dove vengono applicate le *best practice*.

5.1) Configurazione

Per cercare di mantenere i parametri di configurazione in un luogo centralizzato, così da non aver problemi in un futuro ad integrare dell'altro codice, si è deciso di utilizzare l'annotazione `@Configuration`. Una classe con questa annotazione viene trattata come una classe di configurazione dove al suo interno si possono dichiarare metodi annotati con `@Bean`, che creano e configurano i bean da registrare nel *ApplicationContext* di Spring. Nel nostro caso specifico è stato fondamentale nel contesto della richiesta all'API esterno Weather Forecast, in quanto è necessaria la costruzione di un `RestTemplate` con le proprietà necessarie. Nello specifico:

```
@Configuration
@EnableConfigurationProperties(ExternalAPIProperties.class)
public class ExternalAPIConfig {

    ExternalAPIProperties properties;

    public ExternalAPIConfig(ExternalAPIProperties properties) {
        this.properties = properties;
    }

    @Bean
    public RestTemplate restTemplate() {
        SimpleClientHttpRequestFactory factory = new SimpleClientHttpRequestFactory();
        factory.setConnectTimeout(properties.getTimeout());
        factory.setReadTimeout(properties.getTimeout());
        return new RestTemplate(factory);
    }
}
```

Come indicato da questo esempio, abbiamo combinato le funzionalità di `@Configuration` con `@EnableConfigurationProperties(ExternalAPIProperties.class)` che permette di abilitare e caricare una classe di configurazione basata su properties all'interno del contesto di Spring. La motivazione di questa scelta è sempre quella di cercare di mantenere le configurazioni il più centralizzate possibili, all'interno di un file chiamato *application.properties* dove è possibile aggiungere un *prefisso* che identifica un parametro di configurazione.

```
@ConfigurationProperties(prefix = "external.api")
public class ExternalAPIProperties {
    private String url;
    private int timeout;
    private int maxNumData;

    public int getTimeout() { return timeout; }
    public void setTimeout(int timeout) { this.timeout = timeout; }
    public String getUrl() { return url; }
```

```
    public void setUrl(String url) { this.url = url; }
    public int getMaxNumData() { return maxNumData; }
    public void setMaxNumData(int maxNumData) { this.maxNumData = maxNumData; }
}
```

Con l'aggiunta di queste annotazioni, si è in grado di indicare nuovi parametri di configurazione andandole ad aggiungere in maniera incrementale in questo modo: `external.api.nuovo_parametro=valore`, implementando di conseguenza i nuovi metodi *getter* e *setter*.

5.2) Eccezioni

Come indicato nel documento *Analisi dei Requisti v2.0.0*, ci sono diversi errori da gestire come un `NetworkError` o `FileTooBig`. Viste queste necessità, si è deciso di implementare una best practice di *Spring Boot* tramite l'aggiunta dell'annotazione `@ControllerAdvice` che è utilizzata per gestire centralmente le eccezioni e la logica di preprocessing per tutti i controller. Nel nostro caso specifico è stata aggiunta in combinazione l'annotazione `@ExceptionHandler()` per poter gestire le eccezioni globalmente.

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(InvalidCsvException.class)
    public ResponseEntity<String> handleInvalidCsvException(InvalidCsvException ex) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(ex.getMessage());
    }

    @ExceptionHandler(FileTooBigException.class)
    public ResponseEntity<String> handleFileTooBigException(FileTooBigException ex) {

return ResponseEntity.status(HttpStatus.PAYLOAD_TOO_LARGE).body(ex.getMessage());
    }

    @ExceptionHandler(APITimeoutException.class)
    public ResponseEntity<String> handleAPITimeoutException(APITimeoutException ex) {

return ResponseEntity.status(HttpStatus.REQUEST_TIMEOUT).body(ex.getMessage());
    }

    @ExceptionHandler(NetworkErrorException.class)
     public ResponseEntity<String> handleNetworkErrorException(NetworkErrorException
ex) {
                                                                        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(ex.getMessage());
    }
}
```

Con questa implementazione è possibile andare ad indicare un modo di fare l'handling dell'eccezioni in maniera differente in base alle varie casistiche. Permette inoltre di centralizzare la gestione delle eccezioni e di aggiungerne delle ulteriori in maniera incrementale (e molto facilmente).

5.3) Repository

Da capitolato, come indicato nell'UC2.3 all'interno del documento *Analisi dei Requisiti v2.0.0* , l'utente deve essere in grado di caricare i dati tramite una connessione ad un database SQL. Per poter implementare questo requisito seguendo tutte le best practice (e il modo di seguire il modello *esagonale*) è stato necessario creare delle classi con le annotazioni `@Repository`, `@Entity` e `@Table`. Andando per ordine, `@Repository` permette di indicare che una classe è un repository, ovvero un componente responsabile dell'interazione con il database (fa parte dello Spring Data JPA e fornisce un livello di astrazione per operazioni CRUD senza dover scrivere query SQL manualmente), `@Entity` che rappresenta una tabella del database con cui Spring Boot riesce a collegare automaticamente il repository e l'entità usando Spring Data JPA. Utilizzando quest'ultima, Spring Boot riesce a semplificare la gestione dei database e collegare automaticamente i componenti attraverso una serie di step:

1. Scansiona le annotazioni (`@Repository`, `@Entity`, ecc..)
2. Crea il repository (infatti grazie a Spring Data JPA non serve implementare manualmente la classe in questione)
3. Crea automaticamente il database utilizzando JPA e Hibernate (che può essere configurato con il modo indicato in precedenza, andando a definire nel file *application.properties* le configurazioni necessarie)

Nel nostro caso specifico:

```
@Repository
public interface CoordinateRepository extends JpaRepository<CoordinateEntity, Long> {

    @Query("SELECT c FROM CoordinateEntity c WHERE c.datasetType >= :type")
    List<CoordinateEntity> findAllByDatasetType(@Param("type") String type);
}
```

Dove si può notare come, integrando l'annotazione `@Query` è possibile generare un database con una funzione che richiama una query. Come oggetto di ritorno viene generata un'entità personalizzata, chiamata `CoordinateEntity` definita come:

```
@Entity
@Table(name = "coordinates")
public class CoordinateEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "x_label")
    private String xLabel;

    @Column(name = "z_label")
    private String zLabel;

    @Column(name = "y_value")
    private Double yValue;

    @Column(name = "dataset_type")
```

```
    private String datasetType;

    public CoordinateEntity() { }

    public CoordinateEntity(String xLabel, String zLabel, Double yValue, String
datasetType) {
        this.xLabel = xLabel;
        this.zLabel = zLabel;
        this.yValue = yValue;
        this.datasetType = datasetType;
    }

    ---- altri metodi setter e getter
}
```

5.4) Model

All'interno di model ci sono tutte quelle classi definite come oggetti di business. Troviamo infatti `CoordinateEntity.java` e `MatrixData.java`. Come indicato in precedenza, nella Sezione 5.3, utilizzando Spring Boot è possibile definire un'entità tramite l'annotazione `@Entity`. Per vederne l'implementazione si riporta alla Sezione 5.3. `MatrixData.java` è l'interfaccia di ritorno dei metodi per il reperimento dei dati:

`DefaultExternalDataService.java`

```
@Override
public MatrixData fetchData() { ... }
```

`DefaultCsvFileReader.java`

```
@Override
public MatrixData parseCsv(MultipartFile file) throws InvalidCsvException { ... }
```

`DefaultCoordinateService.java`

```
@Transactional(readOnly = true)
@Override
public MatrixData getCoordinates(String datasetType) { ... }
```

Le motivazioni di un tipo di implementazione come questa sono molteplici tra cui:

1. **Astrazione e Flessibilità**: Se una funzione ritorna un'interfaccia invece di una classe specifica, si può cambiare l'implementazione senza modificare il codice che utilizza il risultato della funzione. Il chiamante non ha bisogno di conoscere i dettagli dell'implementazione, ma solo i metodi e le proprietà definiti dall'interfaccia.
2. **Inversione di Dipendenza (Principio DIP - SOLID)**: La funzione che restituisce l'interfaccia segue il principio dell'inversione di dipendenza: il codice dipende da un'astrazione (interfaccia) e non da una concreta implementazione e questo riduce l'accoppiamento tra i componenti, rendendo il sistema più mantenibile e scalabile.
3. **Facilitare il Polimorfismo**: Se diverse classi implementano la stessa interfaccia, possono essere trattate in modo uniforme senza dover conoscere quale specifica classe sta usando. Si può scri-

vere del codice generico che lavora con l'interfaccia, indipendentemente dall'implementazione concreta.

4. **Mocking e Testing**: Se il codice dipende da un'interfaccia piuttosto che da una classe concreta, si può facilmente sostituire l'implementazione reale con un mock o un fake per i test. Questo migliora l'isolamento dei test e riduce la necessità di dipendenze complesse nei test unitari.
5. **Estensibilità**: Se in futuro si devono aggiungere nuove implementazioni, si può farlo senza modificare il codice esistente che utilizza l'interfaccia e di conseguenza permette di implementare nuovi comportamenti senza rompere il codice già scritto.

Nel nostro caso specifico:

```
public interface MatrixData {
    List<String> xLabels();
    List<String> zLabels();
    double[][] yValues();
}
```

ed implementata come:

```
public record MatrixDataImpl(List<String> xLabels, List<String> zLabels, double[][]
yValues) implements MatrixData { }
```

Questo ci permetterà in un futuro di poter aggiungere in maniera incrementale delle ulteriori implementazioni, senza dover andare a riscrivere il vecchio codice. Questo è un'importante aspetto che abbiamo tenuto in grande considerazione durante tutta la parte di MVP$_G$, dalla progettazione all'implementazione.

5.5) Elaborazione dati Database

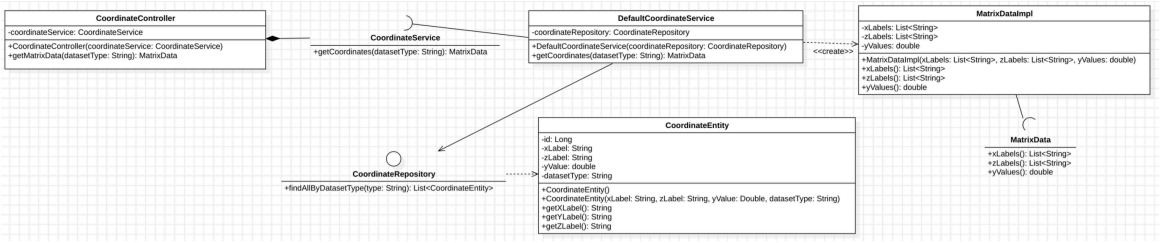

Figura 3: Modulo di interazione con il database

Questo modulo di elaborazione dei dati attraverso database gestisce la raccolta di coordinate (CoordinateEntity) da un repository (CoordinateRepository) e le rielabora tramite un servizio (CoordinateService/DefaultCoordinateService) per restituire all'utente finale dati sotto forma di una matrice (MatrixData/MatrixDataImpl). Il controller (CoordinateController) funge da punto di accesso per l'applicazione, invocando i metodi del servizio e ritornando i risultati.

L'obiettivo di questa struttura è fornire un flusso completo dal livello di accesso ai dati (Repository), passando per la logica di business (Service), sino all'output finale (Controller), incapsulato in una rappresentazione astratta di dati (MatrixData).

5.5.1) CoordinateController

1. **Attributi**:

- **coordinateService: CoordinateService**. Riferimento all'interfaccia del service che fornisce la logica per ottenere le coordinate ed elaborarle.

2. **Costruttore**:

- **CoordinateController(coordinateService: CoordinateService)**. Inietta l'implementazione di CoordinateService necessaria al controller.

3. **Metodi**

- **getMatrixData(datasetType: String): MatrixData**. Chiama il servizio per ottenere i dati (filtrati o identificati da datasetType) e restituisce un oggetto MatrixData che incapsula le coordinate in forma di matrici.

4. **Note**

- Il controller costituisce il layer più esterno, tipicamente l'ingresso da parte di un client (es. chiamata HTTP).

5.5.2) CoordinateService

1. **Metodi**

- **getCoordinates(datasetType: String): MatrixData**. Definisce la firma del metodo che dovrà fornire le coordinate sotto forma di `MatrixData`. Non contiene logica implementativa, solo la specifica del contratto.

5.5.3) DefaultCoordinateService

1. **Attributi**

- **coordinateRepository: CoordinateRepository**. Riferimento al repository che fornisce l'accesso ai dati (entità `CoordinateEntity`).

2. **Costruttore**

- **DefaultCoordinateService(coordinateRepository: CoordinateRepository)**. Inietta l'istanza del repository necessaria per interrogare il database o la sorgente dati.

3. **Metodi**

- **getCoordinates(datasetType: String): MatrixData**. Implementa la logica per:
  - Richiamare il repository e ottenere la lista di entità relative a `datasetType`.
  - Estrarre da ogni `CoordinateEntity` i campi x, y e z.
  - Popolare e restituire un oggetto di tipo `MatrixDataImpl` (che è l'implementazione concreta di `MatrixData`).

4. **Note**

- In uno scenario reale, qui si potrebbe gestire anche caching, mapping più complesso degli attributi, validazione del `datasetType`, ecc.

5.5.4) CoordinateRepository

1. **Metodi**

- **findAllByDatasetType(type: String): List**. Permette di recuperare tutte le entità che corrispondono a uno specifico `datasetType`.

2. **Note**

- In un contesto tipico (es. Spring Data JPA), questa interfaccia sarebbe automaticamente implementata dal framework, fornendo query su un database relazionale (o altro storage).
- L'approccio a repository promuove il pattern di "separazione delle responsabilità": la logica di persistenza è centralizzata in un solo punto.

5.5.5) CoordinateEntity

1. **Attributi**

- **id**: Long — Identificatore univoco dell'entità.
- **xLabel**: String — Etichetta associato all'asse X.
- **zLabel**: String — Etichetta associato all'asse Z.
- **yValue**: double — Valore numerico associato all'asse Y.
- **datasetType**: String — Categoria a cui appartiene l'entità.

2. **Costruttore**

- **CoordinateEntity(xLabel: String, zLabel: String, yValue: Double, datasetType: String)**. Inizializza i campi principali necessari all'entità.

3. **Metodi (Getter)**

- **getXLabel(): String**
- **getYValue(): double**
- **getZLabel(): String**

4. **Note**

- In un'applicazione di persistenza reale, potremmo aggiungere annotazioni (es. `@Entity`, `@Id`, ecc.) per la mappatura con il database.

5.5.6) MatrixData

1. **Metodi**

- **xLabels(): List**
- **zLabels(): List**
- **yValues(): double**

2. **Note**

- Definisce il contratto di come dev'essere rappresentato il set di dati in forma di matrici o liste coordinate.
- Può variare a seconda che si gestiscano singoli valori Y o matrici X-Y-Z.

5.5.7) MatrixDataImpl

1. **Attributi**

- **xLabels: List**
- **zLabels: List**

- **yValues: double**

2. **Costruttore**

- **MatrixDataImpl(xLabels: List, zLabels: List, yValues: double)**. Inizializza i vari campi con i dati estratti dalle entità tramite il servizio.

3. **Metodi**

- **xLabels(): List**
- **zLabels(): List**
- **yValues(): double**

4. **Note**

- Qui avviene la concretizzazione di come i dati di `CoordinateEntity` vengono trasformati in una struttura di output leggibile dall'esterno.

5.6) Modulo API Esterno

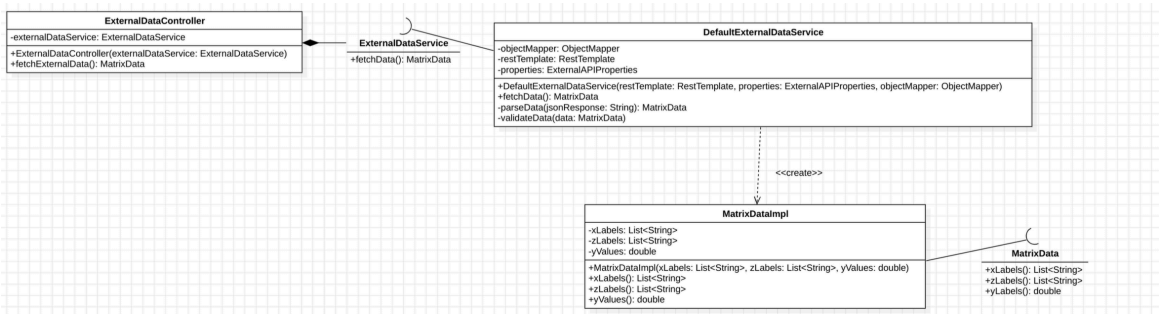

Figura 4: Modulo di interazione con le API esterne

Il **Modulo API Esterno** è responsabile dell'interazione con una sorgente dati remota, accessibile tramite protocollo HTTP. L'obiettivo è quello di integrare nel sistema dati provenienti dal servizio Weather Forecast. Questo modulo segue lo stesso principio architetturale del modulo interno basato su database: utilizza un controller per esporre l'endpoint, un service per incapsulare la logica di accesso e trasformazione, e un'interfaccia comune (`MatrixData`) per unificare il formato del dato.

Il componente principale che espone l'endpoint verso il client è `ExternalDataController`, che si occupa di inoltrare la richiesta al service e restituire il risultato in un formato standardizzato.

5.6.1) ExternalDataController

1. **Attributi**

- **externalDataService: ExternalDataService**. Riferimento al servizio che gestisce la logica di comunicazione con una fonte dati esterna.

2. **Costruttore**

- **ExternalDataController(externalDataService: ExternalDataService)**. Inietta l'istanza del servizio che si occupa di ottenere dati esterni.

3. **Metodi**

- **fetchExternalData(): MatrixData**. Chiama il servizio per ottenere i dati esterni e restituisce il risultato incapsulato in un oggetto `MatrixData`.

4. **Note**

- Come nel modulo interno, il controller funge da punto d'ingresso per i client esterni all'applicazione (es. chiamate HTTP REST).

5.6.2) ExternalDataService

1. **Metodi**

- **fetchData(): MatrixData**. Definisce la firma del metodo incaricato di recuperare dati da una fonte esterna.

2. **Note**

- L'interfaccia consente di astrarre la logica di fetch, permettendo implementazioni alternative in futuro (es. GraphQL, file, ecc.).

5.6.3) DefaultExternalDataService

1. **Attributi**

- **objectMapper**: ObjectMapper — Per la deserializzazione del JSON in oggetti Java.
- **restTemplate**: RestTemplate — Per effettuare le richieste HTTP verso l'API esterna.
- **properties**: ExternalAPIProperties — Contiene le configurazioni (endpoint, headers, ecc.) dell'API esterna.

2. **Costruttore**

- **DefaultExternalDataService(restTemplate: RestTemplate, properties: ExternalAPIProperties, objectMapper: ObjectMapper)**. Inizializza i componenti necessari per l'interazione con l'API esterna.

3. **Metodi**

- **fetchData(): MatrixData**. Esegue una richiesta all'API esterna, riceve una risposta JSON, la deserializzazione e restituisce i dati in formato `MatrixData`.
- **parseData(jsonResponse: String): MatrixData**. Metodo interno privato usato per convertire la risposta JSON in un oggetto `MatrixData`.
- **validateData(data: MatrixData)**. Metodo interno privato che verifica la correttezza dei dati ricevuti prima di restituirli.

5.6.4) MatrixData

1. **Metodi**

- **xLabels(): List**
- **zLabels(): List**
- **yValues(): double**

2. **Note**

- Rappresenta una struttura astratta per contenere e manipolare i dati in forma tabellare o matriciale, utile per visualizzazioni o calcoli successivi.

5.6.5) MatrixDataImpl

1. **Attributi**

- **xLabels**: List
- **zLabels**: List
- **yValues**: **double**

2. **Costruttore**

- **MatrixDataImpl(xLabels: List, zLabels: List, yValues: double)**. Costruisce l'oggetto a partire dai dati ricevuti (tipicamente elaborati da JSON esterni).

3. **Metodi**

- **xLabels(): List**
- **zLabels(): List**
- **yValues(): double**

4. **Note**

- L'implementazione concreta di `MatrixData` usata per rappresentare in modo coerente i dati provenienti dal servizio API esterno.

5.7) Modulo CSV

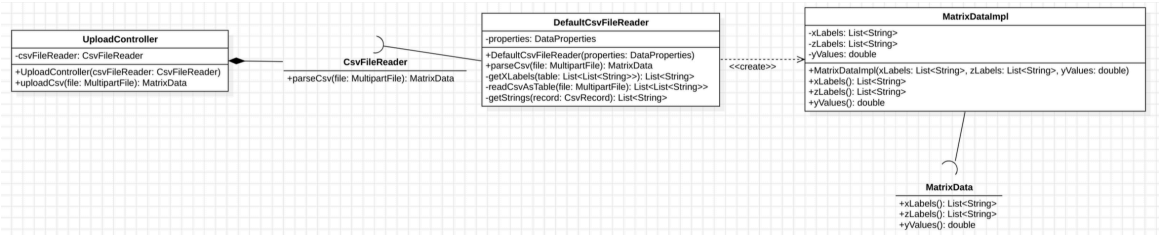

Figura 5: Modulo di interazione con il file CSV

Il modulo CSV ha il compito di ricevere in input un file caricato dall'utente, leggerne il contenuto (tipicamente in formato tabellare), estrarre le informazioni necessarie e trasformarle in una struttura dati `MatrixData` coerente con l'interfaccia comune dell'applicazione. Questo permette di trattare i dati importati come se provenissero da una qualsiasi altra fonte (database, API, ecc.), mantenendo l'uniformità del sistema.

5.7.1) UploadController

1. **Attributi**

- **csvFileReader: CsvFileReader**. Riferimento all'interfaccia che incapsula la logica di parsing dei file CSV.

2. **Costruttore**

- **UploadController(csvFileReader: CsvFileReader)**. Inietta un'istanza dell'interfaccia `CsvFileReader` nel controller.

3. **Metodi**

- **uploadCsv(file: MultipartFile): MatrixData**. Riceve un file caricato tramite richiesta HTTP (tipicamente da un form), lo passa al `CsvFileReader` per il parsing e restituisce l'oggetto `MatrixData`.

4. **Note**

- Usa `MultipartFile`, un tipo comune in Spring per rappresentare i file caricati.

5.7.2) CsvFileReader

1. **Metodi**

- **parseCsv(file: MultipartFile): MatrixData**. Definisce il contratto per trasformare un file CSV in una struttura `MatrixData`.

2. **Note**

- L'interfaccia permette flessibilità: in futuro si potrebbero aggiungere nuovi reader per altri formati senza modificare il controller.

5.7.3) DefaultCsvFileReader

1. **Attributi**

- **properties: DataProperties**. Parametri o configurazioni utilizzate per l'analisi dei file (es. delimitatori, righe da saltare, header, ecc.).

2. **Costruttore**

- **DefaultCsvFileReader(properties: DataProperties)**. Inizializza la classe con le proprietà necessarie alla configurazione del reader.

3. **Metodi**

- **parseCsv(file: MultipartFile): MatrixData**. Metodo principale che esegue:
  - Parsing del contenuto CSV.
  - Estrazione di etichette X/Z e valori Y.
  - Costruzione dell'oggetto `MatrixDataImpl`.
- **getXLabels(table: `List<List<String>>`): List**. Estrae le etichette da usare come asse X dalla tabella CSV.
- **readCsvAsTable(file: MultipartFile): `List<List<String>>`**. Converte il file CSV in una lista di righe (ogni riga è una lista di stringhe).
- **getStrings(record: CsvRecord): List**. Estrae i campi testuali da un record CSV.

4. **Note**

- Incapsula completamente la logica di parsing del file e la trasformazione dei dati grezzi in una struttura coerente.
- Può essere facilmente estesa per supportare più formati.

5.7.4) MatrixData

1. **Metodi**

- **xLabels(): List**

- **zLabels(): List**
- **yValues(): double**

2. **Note**

- Interfaccia comune che rappresenta una matrice di dati indipendentemente dalla loro origine.

5.7.5) MatrixDataImpl

1. **Attributi**

- **xLabels: List**
- **zLabels: List**
- **yValues: double**

2. **Costruttore**

- **MatrixDataImpl(xLabels: List, zLabels: List, yValues: double)**. Costruisce l'oggetto dati completo per la successiva visualizzazione o analisi.

3. **Metodi**

- **xLabels(): List**
- **zLabels(): List**
- **yValues(): double**

4. **Note**

- È la rappresentazione concreta dei dati tabellari letti dal CSV e convertiti in un formato uniforme.