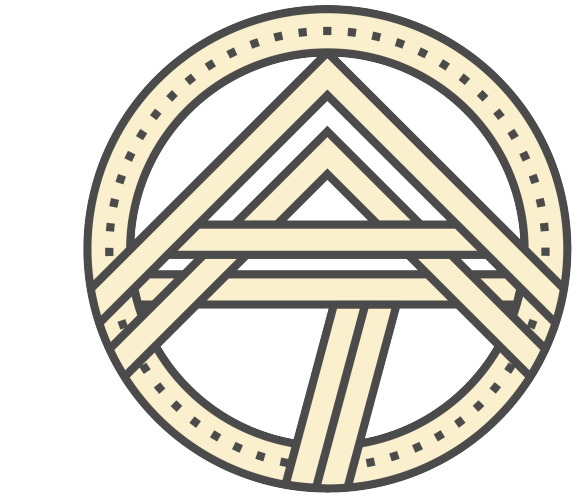

ARCHI7ECHS

Archi7echs - archi7echs@gmail.com

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

Specifica Tecnica

Autore: Il team

Ultima Modifica: 12/04/2025

Tipologia Documento: Interno

Tabella delle revisioni

Rev.	Data	Descrizione	Elaborazione	Verifica
0.6.0	12-04-2025	Aggiunta tecnologie e modifica sezione componenti front-end	Gabriele Checchinato	Pietro Valdagno, Leonardo Lucato
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 **4**
1.1) Finalità del documento 4
1.2) Scopo del progetto 4
1.3) Glossario 5
1.4) Riferimenti 5
1.4.1) Riferimenti normativi 5
1.4.2) Riferimenti informativi 5
2) Tecnologie **6**
2.1) Servizi e Strumenti 6
2.1.1) PostgreSQL 6
2.1.2) Maven 7
2.1.3) Docker 7
2.2) Framework 7
2.2.1) Spring Boot 7
2.2.2) Svelte 8
2.3) Librerie 9
2.3.1) Threlte 9
2.4) Test 9
2.4.1) JUnit 5 9
2.4.2) PITest 9
2.4.3) Mockito 10
2.4.4) Testcontainers 11
2.4.5) Vitest 11
2.4.6) Playwright 12
2.5) Linguaggi 12
2.5.1) Java 12
2.5.2) Typescript 12
3) Architettura **13**
3.1) Architettura logica 13
3.2) Architettura di deployment 13
3.2.1) Struttura a monolite containerizzato vs microservizi 14
3.2.2) Docker e Containerizzazione dell'Applicazione 14
3.3) Database 16
4) Front-end **17**
4.1) Utilities 17
4.1.1) index.svelte.ts 17
4.1.2) data.svelte.ts 19
4.2) Componenti 20
4.2.1) App.svelte 21
4.2.2) Bar.svelte 23
4.2.3) BarPane.svelte 24
4.2.4) CameraSettings.svelte 26
4.2.5) Chart.svelte 28

4.2.6) Color.svelte 30
4.2.7) DataSource.svelte 31
4.2.8) Export.svelte 33
4.2.9) Scene.svelte 34
4.2.10) SettingsPane.svelte 36
5) Back-end 38
5.1) Configurazione 38
5.2) Eccezioni 39
5.3) Repository 40
5.4) Model 41
5.5) Elaborazione dati Database 42
5.5.1) CoordinateController 42
5.5.2) CoordinateService 43
5.5.3) DefaultCoordinateService 43
5.5.4) CoordinateRepository 43
5.5.5) CoordinateEntity 44
5.5.6) MatrixData 44
5.5.7) MatrixDataImpl 44
5.6) Modulo API Esterno 45
5.6.1) ExternalDataController 45
5.6.2) ExternalDataService 46
5.6.3) DefaultExternalDataService 46
5.6.4) MatrixData 46
5.6.5) MatrixDataImpl 47
5.7) Modulo CSV 47
5.7.1) UploadController 47
5.7.2) CsvFileReader 48
5.7.3) DefaultCsvFileReader 48
5.7.4) MatrixData 48
5.7.5) MatrixDataImpl 49

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) Librerie

2.3.1) Threlte

- **Descrizione della tecnologia e del suo utilizzo**: Threlte[G] è una libreria per lo sviluppo di grafica 3D reattiva all'interno di progetti Svelte, basato su Three.js[G] . Consente di costruire scene 3D in modo dichiarativo e integrato con lo stato reattivo dell'applicazione. In questo progetto, Threlte è stato utilizzato per:
  1. Creare visualizzazioni 3D interattive.
  2. Gestire oggetti, luci e telecamere direttamente tramite componenti Svelte.
  3. Semplificare l'integrazione con la logica applicativa grazie alla sua sintassi dichiarativa.
  4. Ottimizzare l'esperienza utente mantenendo buone performance in contesti tridimensionali.

  Il progetto è strutturato con:
  - Componenti Threlte per rappresentare entità 3D.
  - Utilizzo del sistema reactive di Svelte per controllare dinamicamente gli elementi della scena.
  - Caricamento asincrono di asset 3D attraverso strumenti integrati di Threlte.
- **Versione della tecnologia utilizzata**:
  - **Threlte**: 8.0.1
- **Link di riferimento alla documentazione**:
  - Threlte: https://threlte.dev/docs

2.4) Test

2.4.1) 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/

2.4.2) 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

2.4.3) 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/

2.4.4) 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/

2.4.5) Vitest

- **Descrizione della tecnologia e del suo utilizzo**: Vitest$_G$ è un framework di testing moderno pensato per ambienti basati su Vite. È veloce, semplice da configurare e supporta funzionalità avanzate come test asincroni, snapshot e mock. Nel nostro progetto, Vitest è stato adottato per:
  1. Scrivere test unitari e di integrazione in modo semplice e veloce.
  2. Verificare il comportamento delle funzioni logiche e dei componenti Svelte.
  3. Integrare facilmente il testing nel processo di sviluppo continuo.

  Il progetto include:
  - Struttura modulare per l'organizzazione dei test.
  - Coverage automatizzato per la misurazione della qualità del codice.
  - Integrazione con l'ambiente di sviluppo per l'esecuzione rapida dei test.
- **Versione della tecnologia utilizzata**:
  - **Vitest**: 3.0.9
- **Link di riferimento alla documentazione**:
  - Vitest: https://vitest.dev/guide/

2.4.6) Playwright

- **Descrizione della tecnologia e del suo utilizzo**: Playwright$_G$ è uno strumento di testing end-to-end sviluppato da Microsoft, che permette di automatizzare e testare applicazioni web su diversi browser. Nel progetto è stato utilizzato per:
  1. Simulare l'interazione dell'utente con l'interfaccia grafica.
  2. Verificare il comportamento dell'applicazione in scenari reali.
  3. Garantire la stabilità dell'interfaccia utente nelle modifiche successive.

  La struttura del progetto prevede:
  - Test di flusso utente completi su Chrome, Firefox e WebKit.
  - Configurazioni specifiche per ambienti di CI.
  - Screenshot e tracciamento video in caso di errori.
- **Versione della tecnologia utilizzata**:
  - **Playwright**: 1.51.1
- **Link di riferimento alla documentazione**:
  - Playwright: https://playwright.dev/docs/intro

2.5) Linguaggi

2.5.1) 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/

2.5.2) 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 coordinates. 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 all'interno dell'applicazione.
- **Struttura e Funzionalità**: la sua composizione, 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.
- **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.

4.1.1) index.svelte.ts

4.1.1.1) Descrizione

Il file **index.svelte.ts** gestisce i dati, i filtri e la selezione delle barre per l'interfaccia 3D. Contiene funzioni per il recupero dei dati dal database e da un'API esterna, per l'upload dei file CSV, e per l'applicazione dei filtri sui dati visualizzati. Le funzioni di gestione della selezione e dei colori delle barre, insieme a metodi utili per calcolare medie, minimi e massimi, sono anche incluse per rendere l'esperienza utente dinamica.

4.1.1.2) Struttura e Funzionalità

- **Struttura**:
  - Il file contiene funzioni per il recupero dei dati da **getDbData** e **getExternalData**, per il caricamento dei file CSV con **uploadCsvFile**, e per il reset dei filtri con **resetFilter**.
  - Definisce la variabile **fetchedData** per contenere i dati recuperati, insieme a configurazioni di visualizzazione come **spacing**, **xLabels**, e **zLabels**.
  - Il calcolo dei dati computati (media, min/max, ecc.) è gestito da **utils**, che derivano i valori da **fetchedData** e li aggiornano dinamicamente.
  - La logica di selezione è gestita da una classe **Selection**, che gestisce le barre selezionate e fornisce metodi per l'aggiunta, la rimozione e la verifica della selezione.
  - **filter** è uno store globale che contiene i parametri di filtro attivi, come l'intervallo di valori da visualizzare, la selezione dei colori, e i parametri di selezione delle barre.
- **Funzionalità**:
  - **fetchDb()** e **fetchExternal()** recuperano i dati dal server e li impostano in **fetchedData**.
  - **uploadFile(file)** consente all'utente di caricare un file CSV, che viene quindi analizzato e utilizzato per aggiornare **fetchedData**.

- **utils** calcola variabili computate come la media globale, i minimi e massimi, e la posizione di default della telecamera.
- La classe **Selection** gestisce la selezione delle barre, inclusi metodi per aggiungere, rimuovere e verificare la selezione, nonché per ottenere il valore dell'ultima barra selezionata.
- **filter** memorizza i parametri di filtro, inclusi i valori di intervallo, la selezione del colore, e l'attivazione di filtri per la media globale.

4.1.1.3) Props e Variabili Reattive

- **Props:**
  - Il file non riceve props direttamente, ma interagisce con lo store globale **fetchedData** e **filter** per sincronizzare i dati e i parametri di filtro.
  - La variabile **fetchedData** contiene i dati e le configurazioni visive, e **filter** gestisce le preferenze di visualizzazione.
- **Variabili reattive:**
  - **fetchedData**: variabile reattiva che memorizza i dati recuperati (valori, etichette, ecc.).
  - **utils**: variabile derivata che calcola valori computati dai dati.
  - **filter**: store globale che contiene le opzioni di filtro attive.
  - **selection**: oggetto che gestisce lo stato delle barre selezionate.

4.1.1.4) Eventi e Comunicazione

- Il file gestisce gli eventi di selezione tramite la classe **Selection**, che fornisce metodi per aggiornare le barre selezionate, come **add**, **remove**, **clear**, e **toggle**.
- I parametri di filtro sono controllati tramite **filter**, che interagisce con altri componenti per applicare e ripristinare i filtri (es. selezione delle barre, intervallo di visualizzazione, selezione dei colori).
- I dati e i filtri sono sincronizzati con la scena 3D attraverso l'uso di **utils** e **fetchedData**.

4.1.1.5) Stili e Layout

- Poiché questo file è principalmente di logica, non contiene stili o layout visivi. Tuttavia, le variabili e le funzioni calcolate sono utilizzate per sincronizzare la visualizzazione dei dati e la scena 3D nel componente **Scene.svelte**.

4.1.1.6) Esempi di Utilizzo

- **Integrazione nell'applicazione:** Le funzioni di **index.svelte.ts** vengono utilizzate per gestire la selezione delle barre, applicare filtri dinamici e aggiornare la visualizzazione della scena in base alle preferenze dell'utente.

```
import { filter, passesBarFilter, setBarFilterSelection, getValueFromId } from
'./index.svelte.ts';

// Impostazione di un filtro per le barre
const filterSelection = (selectionType: number) => {
  setBarFilterSelection(selectionType);
};

// Applicazione del filtro a una barra specifica
const barId = "2-3";  // esempio di ID di una barra
const barHeight = getValueFromId(barId);  // Ottieni l'altezza della barra
```

```
const isVisible = passesBarFilter(barId, barHeight);  // Verifica se la barra
passa il filtro

if (isVisible) {
  console.log(`Bar ${barId} is visible`);
} else {
  console.log(`Bar ${barId} is hidden`);
}

// Aggiungere o rimuovere barre dalla selezione
filter.selection.add(barId);
filter.selection.remove(barId);
```

4.1.1.7) Dipendenze Esterne

- **three**: utilizzato per la gestione delle geometrie, dei mesh e delle interazioni con la scena 3D (Raycaster, Mesh, Vector3).
- **svelte-tweakpane-ui**: fornisce i controlli UI come **FileValue** per la gestione dei file.
- **lib/data.svelte**: fornisce le funzioni per il recupero dei dati dal backend e per l'upload dei file.
- **Svelte reactivity**: il file sfrutta il sistema reattivo di Svelte per gestire lo stato e la sincronizzazione dei dati e dei filtri.

4.1.2) data.svelte.ts

4.1.2.1) Descrizione

Il file **data.svelte.ts** contiene funzioni per il recupero dei dati da un database e da un'API esterna, nonché per l'upload di file CSV al server. Gestisce la logica di fetching dei dati e il controllo dei file prima di inviarli, fornendo un'interfaccia semplice per interagire con il backend.

4.1.2.2) Struttura e Funzionalità

- **Struttura**:
  - **fetchDbData()**: funzione asincrona che recupera i dati dal database tramite una richiesta GET all'endpoint `/api/coordinates`.
  - **fetchExternalData()**: funzione asincrona che recupera i dati esterni tramite una richiesta GET all'endpoint `/api/external/data`.
  - **uploadCsvFile(file)**: funzione che gestisce il caricamento di un file CSV al server, con controlli per assicurarsi che il file sia del tipo corretto e non superi una dimensione di 10MB.
  - **init()**: funzione di inizializzazione che recupera sia i dati dal database che quelli esterni al caricamento dell'applicazione, chiamando rispettivamente `fetchDbData()` e `fetchExternalData()`.
  - **getDbData()** e **getExternalData()**: funzioni che restituiscono i dati dal database e dall'API esterna, rispettivamente, e caricano i dati se non sono già stati recuperati.
- **Funzionalità**:
  - Le funzioni di fetching recuperano i dati dai rispettivi endpoint API e gestiscono gli errori tramite `try-catch`, mostrando messaggi di errore appropriati se la rete o il server non sono raggiungibili.

- La funzione di upload **uploadCsvFile** gestisce la validazione del file prima di inviarlo, controllando il tipo e la dimensione del file, e invia il file al server tramite una richiesta POST.
- Le funzioni **getDbData()** e **getExternalData()** verificano se i dati sono già stati recuperati prima di fare una nuova richiesta.
- **init()** avvia automaticamente il recupero dei dati al caricamento del file.

4.1.2.3) Props e Variabili Reattive

- **Props:**
  - Il file non riceve props direttamente, ma esporta le funzioni per l'accesso ai dati esterni e al database.
- **Variabili reattive:**
  - **externalData**: variabile che memorizza i dati esterni recuperati tramite **fetchExternalData()**.
  - **dbData**: variabile che memorizza i dati del database recuperati tramite **fetchDbData()**.

4.1.2.4) Eventi e Comunicazione

- Il file non emette eventi custom, ma fornisce funzioni per il recupero e l'upload dei dati che possono essere utilizzate in altre parti dell'applicazione.
- La comunicazione avviene tramite le variabili **externalData** e **dbData**, che sono utilizzate per popolare i componenti dell'applicazione che richiedono questi dati.

4.1.2.5) Esempi di Utilizzo

- Integrazione nell'applicazione: Il file **data.svelte.ts** viene utilizzato per recuperare i dati da backend e file CSV, utilizzando le funzioni esportate come **getDbData()** e **getExternalData()**.

```
import { getDbData, getExternalData, uploadCsvFile } from './data.svelte.ts';

// Recupero dati dal database
const dbData = await getDbData();

// Recupero dati esterni
const externalData = await getExternalData();

// Caricamento file CSV
const file = ... // un file CSV da un input
await uploadCsvFile(file);
```

4.1.2.6) Dipendenze Esterne

- **svelte-tweakpane-ui**: fornisce il tipo **FileValue**, utilizzato per il controllo del file nel caricamento del CSV.
- **fetch API**: utilizzata per effettuare richieste HTTP verso il backend.
- **FormData**: utilizzata per inviare il file CSV al server tramite una richiesta POST multipart.

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 principale dell'applicazione, che coordina il rendering della scena 3D e degli altri componenti di configurazione. Gestisce la visualizzazione del grafico, il pannello delle impostazioni e il controllo della telecamera, garantendo un'esperienza utente completa e interattiva.

4.2.1.2) Struttura e Funzionalità

- **Struttura:**
  - Il componente è racchiuso all'interno di un `<div>` che occupa l'intera finestra del browser (100vh, 100vw), con uno sfondo scuro per mettere in risalto la scena 3D.
  - Utilizza il componente **Canvas** di **threlte/core** per gestire il rendering 3D e ospitare la scena, i pannelli di controllo e gli altri componenti.
  - I componenti integrati includono:
    - **SettingsPane.svelte**: pannello per la configurazione delle impostazioni, come la fotocamera e i filtri.
    - **BarPane.svelte**: pannello per le informazioni delle barre selezionate e l'interazione con le barre.
    - **Scene.svelte**: gestisce la scena 3D, compreso il rendering dei dati e la visualizzazione interattiva.
- **Funzionalità:**
  - Il componente **App.svelte** inizializza e gestisce lo stato del target della telecamera, tramite il binding della variabile `target` con il valore di **utils.defaultTarget** ottenuto da `createUtils()`.
  - Due funzioni sono esportate per interagire con il target della telecamera:

    - **resetTarget()**: ripristina il target della telecamera al valore di default.
    - **getTarget()**: restituisce il valore corrente del target della telecamera.
  - Il componente utilizza **derived** per ottenere e gestire lo stato reattivo del target e degli altri dati necessari per il rendering.

4.2.1.3) Props e Variabili Reattive

- **Props:**
  - **resetTarget**: funzione passata a **SettingsPane.svelte** per consentire il reset del target della telecamera.
  - **target**: variabile passata a **Scene.svelte** per controllare la posizione della visualizzazione 3D.
- **Variabili reattive:**
  - **target**: stato reattivo che memorizza il valore corrente del target della telecamera, inizializzato con **utils.defaultTarget**.
  - **utils**: oggetto derivato che contiene le configurazioni e i calcoli necessari, come il valore di default per il target della telecamera.

4.2.1.4) Eventi e Comunicazione

- **SettingsPane.svelte** riceve la funzione **resetTarget** per permettere all'utente di ripristinare il target della telecamera.
- **Scene.svelte** riceve il prop **target** per impostare la posizione della visualizzazione 3D.

4.2.1.5) Stili e Layout

- Il layout è gestito tramite CSS, con il contenitore principale `<div>` che occupa tutta la finestra del browser e ha uno sfondo scuro (`rgb(14, 22, 37)`) per enfatizzare la scena 3D.
- La scena è renderizzata all'interno di un **Canvas**, e i pannelli di controllo sono disposti sopra di essa.

4.2.1.6) Esempi di Utilizzo

- Integrazione nell'applicazione: Il componente **App.svelte** viene utilizzato come punto di ingresso, gestendo l'intera interfaccia utente e la scena 3D.

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

<App />
```

4.2.1.7) Dipendenze Esterne

- **threlte/core**: fornisce il componente **Canvas** per il rendering 3D e l'integrazione con **Three.js**.
- **lib/index.svelte**: contiene la funzione **createUtils()** per ottenere la configurazione di default e altre utilità.
- **Svelte reactivity**: utilizza il sistema reattivo di Svelte per gestire e sincronizzare le variabili come `target` e **utils**.

4.2.2) Bar.svelte

4.2.2.1) Descrizione

Il componente **Bar.svelte** rappresenta una barra 3D interattiva all'interno della scena, la quale può essere selezionata e manipolata dall'utente. La barra è visualizzata con la sua altezza e posizione calcolata in base ai dati, ed è interattiva tramite il mouse, con funzionalità di selezione e di cambio dell'opacità al passaggio del puntatore. Viene anche visualizzato il valore dell'altezza sopra la barra.

4.2.2.2) Struttura e Funzionalità

- **Struttura**:
  - Il componente utilizza il **Mesh** di **threlte/core** per la rappresentazione della barra 3D, con una **BoxGeometry** e un **MeshStandardMaterial** per il rendering visivo.
  - Il componente **Text** di **threlte/extras** è usato per visualizzare il valore numerico dell'altezza sopra la barra.
  - Gestisce l'interazione con il mouse tramite **Raycaster** per rilevare le barre e i testi selezionati.
  - Viene usato un **Tween** per animare l'effetto di hover (cambiamento di opacità al passaggio del mouse).
- **Funzionalità**:
  - **Interattività**: al passaggio del mouse sulla barra, l'opacità cambia in base alla selezione. Quando l'utente clicca sulla barra, il filtro di selezione cambia a seconda che si tratti di un clic singolo o doppio.
  - **Hover e opacità**: la barra modifica l'opacità al passaggio del mouse, con un effetto di animazione morbido grazie all'uso di **Tween** e **cubicOut**.
  - **Selezione**: supporta la selezione singola e multipla delle barre, con un sistema di gestione dei filtri basato su **filter.selection**.
  - **Visualizzazione altezza barra**: il componente **Text** visualizza l'altezza della barra sopra di essa, mantenendo la posizione e l'orientamento corretti, anche quando la telecamera cambia angolo grazie all'utilizzo del quaternione della telecamera.

4.2.2.3) Props e Variabili Reattive

- **Props**:
  - **id**: identificativo unico per la barra.
  - **coordinates**: coordinate (x, y, z) che determinano la posizione della barra nella scena.
  - **height**: l'altezza della barra, che determina la dimensione della barra stessa e l'altezza visualizzata nel testo.
  - **currentCameraQuaternionArray**: quaternione della telecamera, utilizzato per mantenere l'orientamento del testo (che rappresenta l'altezza della barra).
- **Variabili reattive**:
  - **passesFilter**: variabile derivata che determina se la barra passa il filtro di selezione.
  - **opacity**: variabile che gestisce l'opacità della barra, che può cambiare in base al filtro di selezione e alle azioni dell'utente.
  - **mesh**: riferimento al **Mesh** della barra, utilizzato per gestire le interazioni di selezione.
  - **text**: riferimento al componente **Text** per l'altezza della barra, utile per la gestione delle interazioni col mouse.

- **hover**: variabile animata che gestisce l'effetto hover tramite un tween per il cambiamento di opacità.

4.2.2.4) Eventi e Comunicazione

- **onpointermove**: al passaggio del mouse sulla barra, la funzione cambia il valore di opacità tramite il **Tween** per il passaggio da 0 a 1 (hover attivo) e viceversa quando il puntatore esce dalla barra.
- **onclick (Mesh)**: quando si clicca sulla barra, viene modificata la selezione in base al tipo di clic (singolo o doppio). Un clic singolo attiva il toggle della selezione, mentre un doppio clic imposta la selezione esclusiva per quella barra.
- **onclick (Text)**: quando si clicca sul testo che visualizza l'altezza della barra, viene invocata la funzione **handleTextClick** per gestire l'interazione con la barra e il filtro applicato.

4.2.2.5) Stili e Layout

- La posizione della barra è determinata dalle **coordinates** passate come prop, mentre l'altezza della barra è configurata tramite la **scale** del **Mesh** (dimensione lungo l'asse y).
- L'effetto hover sulla barra è gestito tramite un'animazione che cambia l'opacità del materiale, creando un'animazione visiva fluida quando l'utente interagisce.
- Il testo dell'altezza della barra è centrato sopra la barra stessa, con un piccolo offset in y per renderlo visibile sopra la superficie.

4.2.2.6) Esempi di Utilizzo

- Integrazione nell'applicazione: Il componente **Bar.svelte** viene utilizzato per rappresentare le barre dati nella scena 3D. Ogni barra è interattiva e permette all'utente di selezionarla o visualizzarne l'altezza.

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

<Bar id="1" coordinates={[x, y, z]} height={barHeight}
currentCameraQuaternionArray={cameraQuaternion} />
```

4.2.2.7) Dipendenze Esterne

- **threlte/core**: fornisce il componente **Mesh** per il rendering della barra 3D e l'integrazione con Three.js.
- **threlte/extras**: fornisce il componente **Text** per la visualizzazione dell'altezza della barra e **interactivity** per gestire le interazioni con il mouse.
- **three**: utilizzato per il raycasting e la gestione delle geometrie e dei materiali nel rendering 3D.
- **svelte/motion**: utilizzato per gestire le animazioni (Tween) dell'opacità durante l'effetto hover.
- **lib/index.svelte**: contiene le funzioni di utilità come **passesBarFilter**, **getBarColor**, **isFirstIntersected**, e **handleTextClick**.

4.2.3) BarPane.svelte

4.2.3.1) Descrizione

Il componente **BarPane.svelte** è un pannello di controllo dedicato alla gestione e al filtraggio delle barre presenti nella scena 3D. Fornisce informazioni di dettaglio sulla barra selezionata e strumenti

interattivi per applicare filtri basati sui valori delle barre. Il pannello consente inoltre di resettare le selezioni e di nascondere il filtro visivo, migliorando l'esperienza utente nell'analisi dei dati visualizzati.

4.2.3.2) Struttura e Funzionalità

- **Struttura:**
  - Il componente utilizza i componenti forniti dalla libreria **svelte-tweakpane-ui** per creare un'interfaccia utente modulare e intuitiva.
  - La struttura è organizzata in un elemento **Pane** che funge da contenitore principale, con al suo interno due **Folder**: uno per le informazioni della barra selezionata e uno per gli strumenti di filtraggio.
  - Viene utilizzata una condizione `{#if filter.displayBarFilter}` per mostrare o nascondere il pannello di filtro in base alle impostazioni correnti.
  - Gli elementi interni includono componenti **Text** per la visualizzazione dei dati, **Button** per attivare azioni, **Slider** per regolare l'opacità e **Checkbox** per abilitare/disabilitare la visualizzazione di piani medi per righe e colonne.
- **Funzionalità:**
  - Visualizza informazioni dettagliate della barra selezionata, come riga, colonna, altezza e valori medi calcolati.
  - Permette di applicare filtri alle barre tramite pulsanti che selezionano barre con valori specifici (solo barre selezionate, valori superiori o inferiori all'ultima barra selezionata, oppure resettare il filtro).
  - Consente di modificare l'opacità delle barre selezionate mediante uno slider.
  - Include pulsanti per resettare la selezione e per chiudere il pannello di controllo.

4.2.3.3) Props e Variabili Reattive

- **Props:**
  - Il componente non riceve props direttamente tramite parametri, ma utilizza funzioni e variabili derivate importate da `$lib/index.svelte` per gestire lo stato e la logica delle barre.
- **Variabili reattive:**
  - **selectedBarInfo**: variabile derivata che contiene le informazioni della barra attualmente selezionata.
  - **data**: variabile derivata che raccoglie i dati da `fetchedData.values`.
  - **utils**: variabile derivata ottenuta da `createUtils()`, utile per calcolare valori medi e altre statistiche globali.

4.2.3.4) Eventi e Comunicazione

- Il componente comunica con il resto dell'applicazione attraverso l'utilizzo di funzioni importate che aggiornano lo stato globale:
  - I pulsanti all'interno dei folder invocano funzioni per impostare il filtro sulle barre (`setBarFilterSelection`), resettare la selezione (`resetBarSelection`) e chiudere il pannello (`hideBarFilterPane`).

- I componenti **Slider** e **Checkbox** sono legati a variabili globali (ad esempio, `filter.selectedOpacity`, `filter.showRowAvgPlane`, e `filter.showColAvgPlane`) che aggiornano in tempo reale l'aspetto della scena 3D.

4.2.3.5) Stili e Layout

- Il pannello viene posizionato in modo fisso sullo schermo (position "fixed") con coordinate specifiche, garantendo una visibilità costante durante l'interazione con la scena 3D.
- La struttura a folder consente di organizzare in maniera chiara le informazioni e i controlli, suddividendo il pannello in sezioni distinte per le informazioni e per il filtro.

4.2.3.6) Esempi di Utilizzo

- Integrazione nell'applicazione: Il componente **BarPane.svelte** viene utilizzato per mostrare il pannello di filtraggio quando il flag `filter.displayBarFilter` è attivo.

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

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

4.2.3.7) Dipendenze Esterne

- **svelte-tweakpane-ui**: fornisce i componenti per costruire l'interfaccia utente, inclusi **Pane**, **Button**, **Text**, **Separator**, **Folder**, **Slider** e **Checkbox**.
- **lib/index.svelte**: contiene le funzioni e variabili che gestiscono il filtraggio, il recupero dei dati e la creazione delle utilità necessarie per il funzionamento del pannello.
- **Svelte reactivity**: il componente sfrutta i sistemi di variabili derivate e state management di Svelte per aggiornare in tempo reale i valori mostrati e l'interazione con l'utente.

4.2.4) CameraSettings.svelte

4.2.4.1) Descrizione

Il componente **CameraSettings.svelte** fornisce un'interfaccia utente per la gestione della telecamera all'interno dell'applicazione 3D. Consente agli utenti di resettare la posizione della camera e di controllarne lo zoom, garantendo una visione ottimale della scena 3D.

4.2.4.2) Struttura e Funzionalità

- **Struttura:**
  - Il componente sfrutta il framework **Threlte** per accedere agli elementi della scena 3D, in particolare la camera.
  - Sono presenti tre pulsanti, ciascuno implementato tramite il componente **Button** della libreria **svelte-tweakpane-ui**.
- **Funzionalità:**
  - Il primo pulsante permette il reset della posizione della camera, invocando la funzione `resetCamera` dalla utility `cameraUtils` e ricevendo il nuovo target tramite prop.

- Il secondo e il terzo pulsante gestiscono, rispettivamente, l'azione di zoom in e zoom out, chiamando le funzioni `zoomIn` e `zoomOut` di `cameraUtils`.

4.2.4.3) Props e Variabili Reattive

- **Props**:
  - **resetTarget**: valore passato al componente per reimpostare il target della telecamera.
- **Variabili reattive**:
  - Non sono presenti variabili reattive interne, se non l'utilizzo diretto dei metodi forniti dalle librerie esterne per gestire la camera.

4.2.4.4) Eventi e Comunicazione

- Gli eventi di click sui pulsanti attivano le funzioni di controllo della telecamera:
  - **Reset**: invoca `cameraUtils.resetCamera(camera, resetTarget)` per resettare la posizione della camera.
  - **Zoom In**: invoca `cameraUtils.zoomIn(camera)` per avvicinare la visualizzazione.
  - **Zoom Out**: invoca `cameraUtils.zoomOut(camera)` per allontanare la visualizzazione.

4.2.4.5) Stili e Layout

- Il componente si basa sui controlli forniti da **svelte-tweakpane-ui**, garantendo uno stile coerente con il resto dell'interfaccia utente.
- I pulsanti sono disposti in sequenza e progettati per essere facilmente accessibili, ottimizzando l'interazione con l'utente.

4.2.4.6) Esempi di Utilizzo

- Integrazione nell'applicazione: Il componente **CameraSettings.svelte** viene utilizzato per fornire controlli rapidi per la gestione della telecamera nella UI dell'applicazione 3D.

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

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

4.2.4.7) Dipendenze Esterne

- **threlte/core**: utilizzato per accedere al contesto della scena 3D e ottenere il riferimento alla camera.
- **svelte-tweakpane-ui**: fornisce i componenti UI come **Button** per i controlli interattivi.
- **lib/index.svelte**: contiene la utility `cameraUtils`, che espone le funzioni `resetCamera`, `zoomIn` e `zoomOut` per la gestione della telecamera.

4.2.5) Chart.svelte

4.2.5.1) Descrizione

Il componente **Chart.svelte** si occupa di costruire e gestire la visualizzazione 3D dei dati mediante l'integrazione di grafici a barre e piani di riferimento. Coordina la disposizione di elementi quali la griglia, i piani medi (globali, per riga e per colonna), le etichette degli assi e le barre 3D, assicurando un aggiornamento continuo della visualizzazione in base allo stato della telecamera.

4.2.5.2) Struttura e Funzionalità

- **Struttura**:
  - Il componente è racchiuso in un gruppo (**T.Group**) che contiene tutti gli elementi 3D della scena.
  - Include uno **T.Mesh** che rappresenta la griglia di base, calcolata in base al numero di righe e colonne, con un **PlaneGeometry** e un materiale di tipo **MeshBasicMaterial**.
  - Mostra opzionalmente piani medi:
    - Un piano medio globale, se abilitato, realizzato con **T.Mesh** che usa **PlaneGeometry** e **MeshBasicMaterial**.
    - Un piano medio per la riga selezionata, se abilitato, che evidenzia i dati medi relativi ad una specifica riga, con **T.MeshStandardMaterial** in tonalità rosata.
    - Un piano medio per la colonna selezionata, se abilitato, evidenziato con un materiale di colore blu chiaro.
  - Visualizza etichette per righe e colonne utilizzando il componente **Text** di **threlte/extras**, posizionate con un offset per indicare chiaramente i nomi degli assi.
  - Crea la visualizzazione delle barre 3D iterando su una struttura dati bidimensionale e istanziando il componente **Bar.svelte** per ciascuna barra.
- **Funzionalità**:
  - Recupera e gestisce dati e configurazioni tramite variabili derivate: dati da `fetchedData.values`, etichette da `fetchedData.xLabels` e `fetchedData.zLabels`, e configurazioni di layout e spazio da `filter` e `createUtils()`.
  - Aggiorna in tempo reale l'orientamento della telecamera memorizzando il quaternione corrente in `currentCameraQuaternionArray`, sfruttando un ciclo di animazione basato su `requestAnimationFrame`.
  - Mostra o nasconde componenti (come i piani medi) in base alle impostazioni dei filtri e alle selezioni correnti.

4.2.5.3) Props e Variabili Reattive

- **Props**:
  - Il componente non riceve props esterni direttamente, ma si basa sulle variabili derivate e sugli store importati per gestire lo stato globale.
- **Variabili reattive**:
  - **currentCameraQuaternionArray**: array che memorizza l'orientamento corrente della telecamera, aggiornato in continuo tramite animazione.
  - **data**: dati bidimensionali derivati da `fetchedData.values` che definiscono le altezze delle barre.

- **utils**: oggetto derivato da `createUtils()` per la gestione di configurazioni e calcoli medi (come il valore medio globale, per riga e per colonna).
- **xLabels** e **zLabels**: etichette per gli assi X e Z, rispettivamente, derivate da `fetchedData`.
- **rows** e **cols**: numero di righe e colonne, utili per il calcolo delle dimensioni della griglia.
- **spacing**: distanza fra gli elementi della griglia, derivato dalla configurazione del filtro.
- **selectedBarInfo**: informazioni relative alla barra selezionata, utili per evidenziare e calcolare i piani medi relativi a riga e colonna.

4.2.5.4) Eventi e Comunicazione

- Il componente sfrutta i lifecycle methods di Svelte:
  - **onMount**: inizializza un ciclo di animazione per aggiornare continuamente la variabile `currentCameraQuaternionArray` in base al quaternione della telecamera.
  - **onDestroy**: cancella il ciclo di animazione per evitare memory leak quando il componente viene rimosso.
- La comunicazione con altri componenti avviene mediante:
  - L'importazione e l'utilizzo di funzioni e variabili dallo store globale (`filter`, `fetchedData`, `createUtils`, ecc.), che permettono a **Chart.svelte** di sincronizzarsi con le interazioni utente gestite da altri componenti come **Bar.svelte**.

4.2.5.5) Stili e Layout

- Il layout 3D è definito principalmente tramite le proprietà di posizione e rotazione:
  - La griglia e i piani sono posizionati e orientati per ricreare una rappresentazione spaziale ordinata dei dati.
  - Gli offset per le etichette e i piani medi sono calcolati in relazione alle dimensioni della griglia (rows, cols e spacing), garantendo una disposizione coerente e centrata.
  - L'uso di materiali trasparenti e colori differenti permette di evidenziare le differenze tra gli elementi (ad esempio, il piano medio della riga con una tonalità rossastra e quello della colonna con una tonalità bluastra).

4.2.5.6) Esempi di Utilizzo

- Integrazione nell'applicazione: Il componente **Chart.svelte** viene importato e utilizzato per visualizzare l'intera scena 3D, integrando le barre, la griglia e le etichette.

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

<Chart />
```

4.2.5.7) Dipendenze Esterne

- **threlte/core**: fornisce il supporto per il rendering 3D e l'accesso agli elementi della scena, come la telecamera e i gruppi di oggetti.
- **threlte/extras**: utilizza il componente **Text** per la visualizzazione delle etichette.
- **Three.js**: impiegata per le operazioni matematiche e la definizione delle geometrie (es. **Vector3** e le trasformazioni relative a **Mesh**).
- **Svelte**: sfrutta i lifecycle hooks (`onMount`, `onDestroy`) e il sistema reattivo per aggiornare in tempo reale le proprietà della scena.

- **lib/index.svelte**: fornisce gli store e le utility necessarie per la gestione dei dati, dei filtri e del layout della scena, inclusa la funzione `createUtils()` e lo store `fetchedData`.

4.2.6) Color.svelte

4.2.6.1) Descrizione

Il componente **Color.svelte** offre un'interfaccia di selezione per il tipo di colore, utilizzando il componente **List** fornito dalla libreria **svelte-tweakpane-ui**. Il componente permette all'utente di scegliere tra differenti opzioni, aggiornando lo stato globale relativo alla selezione del colore.

4.2.6.2) Struttura e Funzionalità

- **Struttura**:
  - Il file importa il componente **List** e il tipo **ListOptions** da **svelte-tweakpane-ui**.
  - Viene definito un oggetto `options` che specifica le proprietà della lista (numero di colonne, righe e valori).
  - Il componente espone il controllo **List** che viene renderizzato con un'etichetta «Color type» e le opzioni definite.
- **Funzionalità**:
  - Permette di selezionare un valore da una lista numerica, rappresentante il tipo di colore.
  - La selezione effettuata viene collegata direttamente alla proprietà `filter.colorSelection` dello store globale, assicurando che la scelta dell'utente venga propagata all'intera applicazione.

4.2.6.3) Props e Variabili Reattive

- **Props**:
  - Il componente non riceve props esterne, ma utilizza il binding per collegare lo stato del componente ad una variabile globale (`filter.colorSelection`).
- **Variabili reattive**:
  - **options**: oggetto che definisce la configurazione della lista (1 colonna, 2 righe, 3 valori disponibili).
  - **filter.colorSelection**: variabile dello store globale, legata al valore selezionato dall'utente tramite il componente **List**.

4.2.6.4) Eventi e Comunicazione

- Il componente utilizza il binding bidirezionale (`bind:value`) sul componente **List** per aggiornare in tempo reale lo stato globale `filter.colorSelection` quando l'utente effettua una selezione.
- Non sono definiti eventi custom, poiché la comunicazione avviene direttamente tramite il meccanismo di binding reattivo di Svelte.

4.2.6.5) Stili e Layout

- Il layout è gestito internamente dal componente **List** della libreria **svelte-tweakpane-ui**, che assicura una presentazione coerente ed integrata con il resto dell'interfaccia utente.
- La configurazione delle opzioni (colonne, righe, valori) permette di personalizzare la colorazione in base agli elementi nella lista.

4.2.6.6) Esempi di Utilizzo

- Integrazione nell'applicazione: Il componente **Color.svelte** viene usato per permettere agli utenti di impostare il tipo di colore preferito per il filtraggio o la visualizzazione, aggiornando automaticamente lo stato globale.

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

<Pane title="Settings" position="fixed">
  <TabGroup>
    <TabPage title="Color">
      <Color />
    </TabPage>
  </TabGroup>
</Pane>
```

4.2.6.7) Dipendenze Esterne

- **svelte-tweakpane-ui**: fornisce il componente **List** e il tipo **ListOptions**, utilizzati per creare l'interfaccia di selezione.
- **lib/index.svelte**: contiene lo store `filter` che include la variabile `colorSelection` per la gestione della scelta del colore nell'applicazione.

4.2.7) DataSource.svelte

4.2.7.1) Descrizione

Il componente **DataSource.svelte** gestisce il recupero e l'upload dei dati da diverse origini, interagendo con API esterne e un database interno. Inoltre, integra funzionalità per il reset della telecamera, garantendo che la vista 3D si adatti in modo appropriato dopo ogni operazione di caricamento o invio file.

4.2.7.2) Struttura e Funzionalità

- **Struttura:**
  - Il componente importa e utilizza vari controlli dalla libreria **svelte-tweakpane-ui**, come **Button** e **File**, per fornire un'interfaccia utente interattiva.
  - Viene utilizzato **useThrelte** per ottenere il riferimento alla camera dalla scena 3D, permettendo l'integrazione con le funzionalità di reset della visualizzazione.
  - Il componente riceve il prop **resetTarget** per riapplicare la configurazione di default della telecamera.
  - La gestione dello stato per il file da inviare è realizzata attraverso una variabile reattiva, **file**, definita come **FileValue**.
- **Funzionalità:**
  - Il pulsante «Select API» invoca la funzione **fetchExternal()** per recuperare dati da un'API esterna e successivamente resetta la telecamera tramite **cameraUtils.resetCamera()**.
  - Il pulsante «Slect DB1» consente di ottenere dati dal database interno chiamando **fetchDb()**, seguito dal reset della telecamera per sincronizzare la visualizzazione.
  - Un controllo **File** permette all'utente di selezionare un file, mentre il pulsante «Select CSV» invia il file selezionato al server tramite la funzione **uploadFile()**.

4.2.7.3) Props e Variabili Reattive

- **Props**:
  - **resetTarget**: viene passato al componente per definire il target di reset della telecamera, essenziale per il corretto riposizionamento della visualizzazione 3D.
- **Variabili reattive**:
  - **file**: variabile di stato tipizzata come **FileValue**, inizialmente non definita, che memorizza il file selezionato dall'utente per l'upload tramite interfaccia.

4.2.7.4) Eventi e Comunicazione

- Il componente comunica con il resto dell'applicazione attraverso:
  - **Eventi click**: i pulsanti «External API» e «DB1» attivano, rispettivamente, le funzioni **fetchExternal()** e **fetchDb()**, seguite dall'invocazione del reset della telecamera tramite **cameraUtils.resetCamera(camera, resetTarget)**.
  - **File Binding**: il controllo **File** utilizza il binding bidirezionale per collegare il valore selezionato alla variabile **file**, consentendo successivamente il caricamento del file tramite il pulsante «Send file CSV».

4.2.7.5) Stili e Layout

- Il layout del componente è gestito tramite i controlli forniti da **svelte-tweakpane-ui**, garantendo un'interfaccia pulita e coerente.
- I pulsanti sono distribuiti per offrire scelte chiare all'utente, mentre il controllo per la selezione del file permette una facile navigazione e caricamento dei dati.

4.2.7.6) Esempi di Utilizzo

- Integrazione nell'applicazione: Il componente **DataSource.svelte** viene utilizzato per selezionare la fonte dati desiderata (API esterna, DB interno o file CSV), e per assicurare che la telecamera si resetti in seguito a tali operazioni.

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

 <Pane title="Settings" position="fixed">
  <TabGroup>
    <TabPage title="Source">
      <DataSource {resetTarget} />
    </TabPage>
  </TabGroup>
</Pane>
```

4.2.7.7) Dipendenze Esterne

- **svelte-tweakpane-ui**: fornisce i componenti **Button** e **File** per la gestione dell'interfaccia utente interattiva.
- **threlte/core**: utilizzato per ottenere il riferimento alla camera della scena 3D tramite **useThrelte**, integrandosi con le funzionalità di reset della visualizzazione.
- **lib/index.svelte**: contiene le funzioni **fetchDb()**, **fetchExternal()**, **uploadFile()** e la utility **cameraUtils**, che gestiscono il recupero e l'invio dei dati e il controllo della telecamera.

4.2.8) Export.svelte

4.2.8.1) Descrizione

Il componente **Export.svelte** consente di catturare un'istantanea della scena 3D e salvarla come immagine. È pensato per offrire agli utenti una semplice funzionalità di «export» della visualizzazione corrente, integrata con la libreria **Threlte** per il rendering 3D.

4.2.8.2) Struttura e Funzionalità

- **Struttura:**
  - Il componente importa il **Button** dalla libreria **svelte-tweakpane-ui** per creare un'interfaccia utente semplice e intuitiva.
  - Utilizza **useThrelte** per ottenere i riferimenti a renderer, scena e camera, necessari per catturare lo screenshot della scena 3D.
  - Definisce una funzione locale, **handleScreenshot**, che invoca la funzione **takeScreenshot** importata dallo store globale per eseguire l'export dell'immagine.
- **Funzionalità:**
  - Quando l'utente clicca sul pulsante, viene attivata la funzione **handleScreenshot()**, che cattura lo stato corrente della scena 3D e lo salva come immagine.
  - La funzione **takeScreenshot** utilizza i riferimenti al renderer, alla scena e alla camera per generare lo screenshot, integrando le capacità di rendering offerte da **Threlte**.

4.2.8.3) Props e Variabili Reattive

- **Props:**
  - Il componente **Export.svelte** non riceve props esterni; invece, accede ai riferimenti necessari tramite **useThrelte** e alle funzioni globali importate da **lib/index.svelte**.
- **Variabili reattive:**
  - **renderer**, **scene**, **camera**: variabili ottenute da **useThrelte** che rappresentano rispettivamente il renderer, la scena e la camera della visualizzazione 3D, aggiornate in tempo reale.

4.2.8.4) Eventi e Comunicazione

- Il componente espone un singolo evento di click sul pulsante:
  - **on:click**: quando il pulsante viene cliccato, viene eseguita la funzione **handleScreenshot()** che comunica con la funzionalità di esportazione presente nello store globale, catturando lo stato attuale della scena e salvandolo come immagine.

4.2.8.5) Stili e Layout

- Il layout è definito in modo semplice, affidandosi allo stile predefinito del componente **Button** della libreria **svelte-tweakpane-ui**, che assicura un aspetto coerente con il resto dell'interfaccia utente.
- Il pulsante esplicita chiaramente la sua funzione con l'etichetta «Export as image», rendendo chiara l'intenzione del controllo.

4.2.8.6) Esempi di Utilizzo

- Integrazione nell'applicazione: Il componente **Export.svelte** può essere integrato in qualsiasi parte dell'applicazione dove sia necessario fornire agli utenti la possibilità di esportare una visualizzazione 3D.

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

<Pane title="Settings" position="fixed">
  <TabGroup>
    <TabPage title="Export">
      <Export />
    </TabPage>
  </TabGroup>
</Pane>
```

4.2.8.7) Dipendenze Esterne

- **svelte-tweakpane-ui**: fornisce il componente **Button** utilizzato per attivare l'azione di screenshot.
- **threlte/core**: utilizzato per ottenere i riferimenti a renderer, scena e camera tramite **useThrelte**, che sono essenziali per generare lo screenshot della scena 3D.
- **lib/index.svelte**: contiene la funzione **takeScreenshot**, che implementa la logica per la cattura e l'export dell'immagine dalla scena corrente.

4.2.9) Scene.svelte

4.2.9.1) Descrizione

Il componente **Scene.svelte** si occupa di definire e gestire la scena 3D dell'applicazione. Include la configurazione della fotocamera con controlli di navigazione, l'illuminazione e la visualizzazione del grafico interattivo, in modo da fornire una visualizzazione immersiva e dinamica dei dati.

4.2.9.2) Struttura e Funzionalità

- **Struttura:**
  - Il componente imposta una **PerspectiveCamera** con una posizione predefinita per inquadrare l'intera scena.
  - All'interno della camera vengono utilizzati gli **OrbitControls** per consentire all'utente di interagire con la scena, abilitando funzioni come il damping e il limite sull'angolo polare.
  - Viene integrato un componente **Gizmo** per facilitare la navigazione e fornire riferimenti visivi alla direzione e all'orientamento della fotocamera.
  - La scena include diverse sorgenti di luce, con un'**AmbientLight** per un'illuminazione diffusa e due **DirectionalLight** per simulare luci direzionali con ombre.
  - Il componente **Chart** è incorporato per visualizzare il grafico 3D, che rappresenta i dati attraverso barre e griglie.
- **Funzionalità:**
  - Configura la fotocamera per rendere predefinita la vista 3D e applica controlli interattivi per la navigazione (ruotare, zoomare e muoversi).
  - Gestisce l'orientamento della scena tramite **OrbitControls**, che permettono un'esperienza utente fluida, grazie anche alla possibilità di attivare o meno l'auto-rotazione.
  - Integra elementi di illuminazione per garantire una resa visiva ottimale della scena e per evidenziare il modello 3D.
  - Incorpora il componente **Chart** per renderizzare dinamicamente i dati in un ambiente 3D.

4.2.9.3) Props e Variabili Reattive

- **Props**:
  - **target** - viene passato al componente per definire il punto centrale della visualizzazione, fondamentale per il corretto funzionamento dei controlli di orbiting della fotocamera.
- **Variabili reattive**:
  - **autoRotate** - variabile locale impostata a `false`, che controlla se l'auto-rotazione della scena è abilitata nei controlli della camera.

4.2.9.4) Eventi e Comunicazione

- Il componente non gestisce direttamente eventi custom, ma integra i seguenti meccanismi di comunicazione:
  - **OrbitControls**: gestiscono eventi di interazione dell'utente (movimenti del mouse, zoom, ecc.) per aggiornare in tempo reale la posizione e l'orientamento della telecamera.
  - **Props Binding**: il prop **target** viene passato ai controlli per garantire che la fotocamera si focalizzi sul punto desiderato, sincronizzando la visualizzazione con le modifiche apportate da altri componenti.

4.2.9.5) Stili e Layout

- La disposizione della scena è definita dalle proprietà di posizione e rotazione:
  - La **PerspectiveCamera** viene posizionata in modo strategico (`[15, 7.1, 15]`) per garantire una visualizzazione equilibrata del contenuto 3D.
  - I controlli degli **OrbitControls** offrono un'interazione fluida e reattiva, mentre la presenza del **Gizmo** agevola l'orientamento.
  - Le luci (ambientale e direzionali) sono posizionate per illuminare in maniera omogenea e per creare ombre realistiche.

4.2.9.6) Esempi di Utilizzo

- Integrazione nell'applicazione: Il componente **Scene.svelte** viene utilizzato come contenitore della scena 3D, includendo fotocamera, controlli, luci ed il grafico interattivo, e rappresenta il cuore della visualizzazione 3D.

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

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

4.2.9.7) Dipendenze Esterne

- **threlte/core**: fornisce componenti essenziali come **PerspectiveCamera** e **T.Group** per il rendering della scena 3D.
- **threlte/extras**: include componenti aggiuntivi come **OrbitControls** per l'interazione con la fotocamera e **Gizmo** per facilitare l'orientamento visivo.

- **Chart.svelte**: componente integrato che gestisce la visualizzazione dettagliata dei dati in forma di grafico 3D.
- **Svelte**: sfrutta il meccanismo di binding dei props per sincronizzare la visualizzazione della scena con il target passato dall'esterno.

4.2.10) SettingsPane.svelte

4.2.10.1) Descrizione

Il componente **SettingsPane.svelte** funge da pannello principale per la configurazione e il controllo dell'applicazione 3D. Offre una serie di schede (tab) per regolare la telecamera, la fonte dei dati, i filtri, la selezione dei colori e l'export della visualizzazione, centralizzando tutte le impostazioni in un'unica interfaccia accessibile.

4.2.10.2) Struttura e Funzionalità

- **Struttura**:
  - Il componente importa e utilizza vari controlli da **svelte-tweakpane-ui**, quali **Pane**, **TabGroup** e **TabPage**, per organizzare le impostazioni in schede distinte.
  - All'interno del **Pane** principale, è integrato un **TabGroup** che contiene diverse **TabPage**:
    - **Camera**: ospita il componente **CameraSettings.svelte** per il controllo della fotocamera.
    - **Source**: contiene il componente **DataSource.svelte** per la selezione della fonte dati (API, DB o file CSV).
    - **Filter**: include il componente **DataFilter.svelte** per applicare filtri sui dati visualizzati.
    - **Color**: integra il componente **Color.svelte** per la scelta del tipo di colore.
    - **Export**: comprende il componente **Export.svelte** per esportare la scena corrente come immagine.
  - Viene impostato il tema globale predefinito attraverso la funzione **ThemeUtils.setGlobalDefaultTheme**, che utilizza i preset standard.
- **Funzionalità**:
  - Consente all'utente di navigare tra diverse schede per accedere rapidamente alle impostazioni desiderate.
  - Permette di interagire con componenti specifici che gestiscono aspetti chiave dell'interfaccia 3D, come il reset della fotocamera, la scelta della fonte dati, l'applicazione dei filtri, la selezione del colore e l'export della visualizzazione.
  - Viene passato il prop **resetTarget** ai componenti **CameraSettings** e **DataSource** per sincronizzare il comportamento della telecamera con le azioni di configurazione effettuate.

4.2.10.3) Props e Variabili Reattive

- **Props**:
  - **resetTarget**: valore ricevuto dal componente genitore, utilizzato per reimpostare il target della telecamera in **CameraSettings** e **DataSource**.
- **Variabili reattive**:
  - Non sono presenti variabili reattive specifiche interne, poiché lo stato è principalmente gestito attraverso i binding e le impostazioni nei componenti importati.

4.2.10.4) Eventi e Comunicazione

- Il componente facilita la comunicazione tra le impostazioni e il resto dell'applicazione tramite l'inclusione di componenti figlio che aggiornano lo stato globale:
  - **CameraSettings** e **DataSource** ricevono **resetTarget** per il reset della telecamera.
  - Le schede **Filter**, **Color** ed **Export** comunicano direttamente con i rispettivi store e funzioni globali per aggiornare l'interfaccia 3D e la visualizzazione dei dati.
  - La struttura a tab permette agli utenti di passare agevolmente da una sezione all'altra, garantendo un flusso di interazione intuitivo.

4.2.10.5) Stili e Layout

- Il pannello è posizionato in modo fisso sullo schermo, assicurando la visibilità delle impostazioni indipendentemente dalla navigazione nella scena 3D.
- L'organizzazione in schede tramite **TabGroup** e **TabPage** offre una chiara separazione delle funzionalità, mantenendo un layout ordinato e facilmente navigabile.
- L'utilizzo del tema globale impostato con **ThemeUtils.presets.standard** garantisce uno stile coerente e professionale in tutta l'interfaccia utente.

4.2.10.6) Esempi di Utilizzo

- Integrazione nell'applicazione: Il componente **SettingsPane.svelte** viene incluso come strumento principale per la configurazione dell'interfaccia, consentendo agli utenti di modificare le impostazioni di fotocamera, fonte dati, filtri, colori ed export.

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

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

4.2.10.7) Dipendenze Esterne

- **svelte-tweakpane-ui**: fornisce i componenti per creare l'interfaccia utente (Pane, TabGroup, TabPage, Button, ecc.) e per impostare il tema globale.
- **lib/index.svelte**: contiene le funzioni e gli store che gestiscono le impostazioni globali, integrandosi con i componenti specifici di configurazione (CameraSettings, DataSource, DataFilter, Color, Export).
- **Threlte**: indirettamente coinvolto nella gestione e visualizzazione della scena 3D, influenzato dalle impostazioni fornite tramite questo pannello.

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

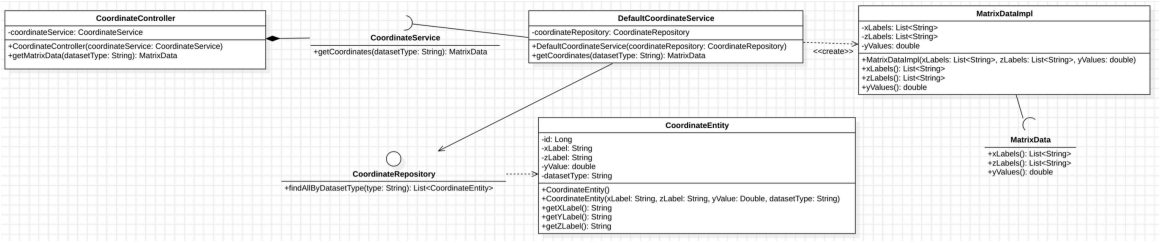

Figura 3: Modulo di interazione con il database

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

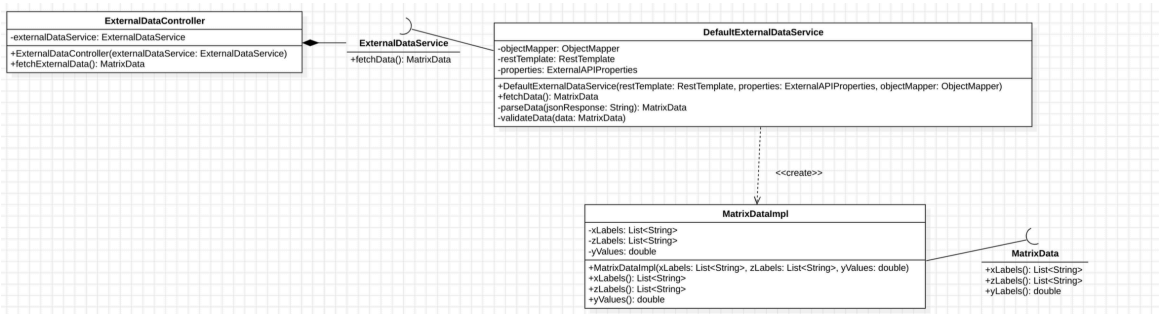

Figura 4: Modulo di interazione con le API esterne

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

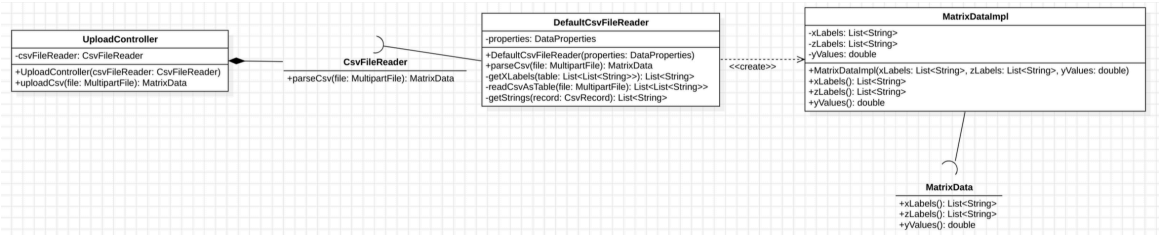

Figura 5: Modulo di interazione con il file CSV

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.