# Il gruppo Avant-Garde

sweavantgarde@gmail.com

---

**SPECIFICA TECNICA**

---

Informazioni sul documento:

| | |
|---|---|
| **Versione** | 1.0.0 |
| **Approvazione** | Lorenzo Pasqualotto |
| **Redazione** | Zaccaria Marangon, Jessica Carretta |
| **Verifica** | Luca Securo, Zaccaria Marangon |
| **Uso** | Esterno |

# Registro delle Modifiche

| Versione | Data | Nominativo | Ruolo | Descrizione |
|---|---|---|---|---|
| v0.0.1 | 28-04-24 | Zaccaria Marangon | Amministratore | Prima redazione, scrittura delle sezioni 1.1,1.2. |
| v0.1.0 | 29-04-24 | Luca Securo | Responsabile | Verifica del documento alla versione v0.0.1. |
| v0.1.1 | 05-05-24 | Jessica Carretta | Progettista | Scrittura delle sezioni 2, 3, 4, 5 e 6. |
| v0.2.0 | 07-05-24 | Zaccaria Marangon | Verificatore | Revisione e correzione delle sezioni 2, 3, 4, 5 e 6. |
| v1.0.0 | 08-05-24 | Lorenzo Pasqualotto | Responsabile | Approvazione del documento. |

# Firma di approvazione esterna

| Versione | Data | Nominativo | Firma |
|---|---|---|---|
| v1.0.0 | 08-05-24 | Matteo Bassani |  |

# Indice

# Elenco delle figure

# Elenco delle tabelle

## Note

Si tenga presente che alcuni termini utilizzati nel documento riportano la lettera **G** in apice, allo scopo di evidenziare le parole che assumono uno specifico significato nell'ambito del progetto. Per comprenderle in maniera corretta, si rimanda il lettore al documento "Glossario", che contiene un elenco completo di tutte le terminologie utilizzate con relative definizioni, allo scopo di costruire un linguaggio uniforme che possa migliorare la comunicazione tra i componenti interni al gruppo e gli stakeholders$^{G}$ esterni.

# 1 Introduzione

## 1.1 Scopo del documento

Il presente documento ha lo scopo di descrivere nel dettaglio i componenti utilizzati e le scelte progettuali adottate per la realizzazione completa del progetto. Verranno quindi trattati in dettaglio gli aspetti fondamentali riguardanti i requisiti necessari per il prodotto, le tecnologie adottate, i design pattern applicati, l'architettura logica e quella di deployment del prodotto. Si intende fornire il ragionamento e le disposizioni per lo sviluppo del progetto, in modo da garantire coerenza con i requisiti specificati nel documento *Analisi dei requisiti v5.0.0*.

## 1.2 Scopo del progetto

Il progetto nasce nell'ambito dei **sistemi gestionali di magazzino**, meglio noti con il termine inglese di *Warehouse Management Systems* (WMS), con l'obiettivo di risolvere una serie di problematiche derivanti dalle soluzioni tradizionali tuttora presenti sul mercato.
Il focus principale sarà migliorare la user experience, tramite la realizzazione di un applicativo che proponga all'utente un'interazione con il magazzino in un ambiente di lavoro 3D.
Tale soluzione, rispetto ai tradizionali sistemi 2D, garantirebbe una maggiore comprensione degli spazi, proponendo una visualizzazione più intuitiva e completa degli spazi di magazzino. Permetterebbe quindi all'utente di prendere decisioni in modo più efficace ed efficiente, permettendo così di ottimizzare i processi di logistica.

Per raggiungere questo obiettivo, l'ambiente di lavoro non può essere una semplice visualizzazione del magazzino. L'utente dovrà infatti poter:

- Spostarsi all'interno dell'ambiente 3D;
- Progettare le scaffalature che sono presenti nel magazzino e modificarle nel tempo;
- Simulare i flussi di movimento di prodotti.

Il progetto deve concretizzarsi nella realizzazione di una web app fruibile agli impiegati d'ufficio ed incentrata sulla visualizzazione 3D del magazzino. Per visionare il capitolato[G] completo e la documentazione del gruppo, si veda la sezione Riferimenti Esterni del documento.

# 2 Tecnologie

In questa sezione si riportano gli strumenti e le tecnologie impiegati per lo sviluppo e l'implementazione del software relativo al progetto "WMS3". Si riporta dunque una descrizione delle tecnologie e del linguaggio di programmazione utilizzato, nonché delle librerie e dei framework[G] necessari. L'obiettivo principale è garantire che il software sia sviluppato utilizzando le tecnologie più appropriate in termini di efficienza, efficacia ed affidabilità.

## 2.1 Linguaggi e formato dati

Per lo sviluppo e l'implementazione del software viene utilizzato il seguente linguaggio di programmazione.

| **JavaScript (+ JSX)** | |
|---|---|
| **Descrizione generale:** | JS è un linguaggio di programmazione ad alto livello interpretato e orientato agli eventi. Viene comunemente utilizzato per aggiungere interattività e dinamicità alle pagine web, ma può essere utilizzato anche per lo sviluppo di applicazioni web, mobile e desktop. JSX, invece, è un'estensione di sintassi per JS che consente di scrivere codice simile all'HTML all'interno di JavaScript, semplificando la definizione dei componenti UI e della loro struttura. JSX viene comunque trasformato in JS regolare durante il processo di compilazione. |

| | |
|---|---|
| **Utilizzo nel progetto:** | JS è il linguaggio di programmazione utilizzato per implementare la logica, gestire le interazioni dell'utente, gestire i contenuti dinamici nel progetto, nonché per il testing. JSX, nello specifico, viene utilizzato per semplificare e rendere più leggibile la codifica delle componenti dell'UI. |

Table 1: Linguaggi di programmazione utilizzati

Per quanto riguarda il formato dei dati vengono utilizzati i seguenti.

| **JSON** | |
|---|---|
| **Descrizione generale:** | JSON è un formato di scrittura (basato su JS) che si caratterizza per la sua grande leggibilità, dovuta ad una sintassi semplice e facilmente interpretabile dai computer. Viene utilizzato in diversi contesti, tra cui lo sviluppo web, le API di servizi web e lo scambio di dati tra applicazioni. |
| **Utilizzo nel progetto:** | Nel progetto, JSON viene utilizzato per memorizzare la configurazione del magazzino per poter, successivamente, ricostruire il magazzino semplicemente caricando il file .json. Viene inoltre utilizzato come formato dei messaggi scambiati tra componenti ed API. |
| **SVG** | |
| **Descrizione generale:** | SVG è un formato di file basato su XML utilizzato per descrivere grafica vettoriale bidimensionale. In particolare, consente di creare immagini scalabili senza perdita di qualità, adatte per essere visualizzate su dispositivi di diversi formati e dimensioni. |
| **Utilizzo nel progetto:** | Nel progetto, SVG viene utilizzato per configurare una planimetria personalizzata del magazzino. |

Table 2: Formato dei dati utilizzati

## 2.2 Tecnologie implementative

| **Node.js (con npm)** | |
|---|---|
| **Versione:** | v21.4.0 |
| **Documentazione:** | • Node.js: `https://nodejs.org/en/learn/getting-started/introduction-to-nodejs` *(ultimo accesso 05-05-24)*<br>• npm: `https://docs.npmjs.com/` *(ultimo accesso 05-05-24)* |
| **Descrizione generale:** | Node.js è un framework JavaScript multipiattaforma e open-source tra i più utilizzati. In particolare, si tratta di un ambiente di esecuzione che permette di eseguire codice JavaScript come un qualsiasi linguaggio di programmazione, al di fuori dei browser. Invece, npm è il suo gestore di pacchetti predefinito. |
| **Utilizzo nel progetto:** | Viene utilizzato Node.js per la gestione back-end della web-app. Mentre, npm viene utilizzato come gestore di pacchetti per installare le dipendenze del progetto. |
| **React.js** | |

| | |
|---|---|
| **Versione:** | ^v18 |
| **Documentazione:** | `https://react.dev/learn` *(ultimo accesso 05-05-24)* |
| **Descrizione generale:** | Si tratta di una libreria open-source per la creazione di interfacce utente in JavaScript. Utilizza il concetto di componentizzazione e il Virtual DOM per migliorare le prestazioni. |
| **Utilizzo nel progetto:** | Viene utilizzata questa libreria per creare le varie componenti dell'UI in quanto particolarmente adatta allo sviluppo di componenti riutilizzabili e dinamiche, con dati variabili nel tempo. |
| **Next.js** | |
| **Versione:** | v14.1.3 |
| **Documentazione:** | `https://nextjs.org/docs` *(ultimo accesso 05-05-24)* |
| **Descrizione generale:** | Next.js è un framework open-source che permette di costruire e distribuire rapidamente applicazioni su larga scala e pronte per la produzione, renderizzandole lato server. In particolare, si basa ed estende la libreria JavaScript React e utilizza Node.js come ambiente run-time. |
| **Utilizzo nel progetto:** | Next.js viene utilizzato per migliorare le funzionalità di React fornendo una struttura e degli strumenti che ne migliorano le prestazioni. Nello specifico, per creare la struttura dell'applicazione, implementare le API e gestire il routing. A tal proposito, si precisa l'utilizzo dell'app router per gestire la navigazione tra le diverse viste all'interno dell'app. |
| **Three.js** | |
| **Versione:** | ^v0.162.0 |
| **Documentazione:** | `https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene` *(ultimo accesso 05-05-24)* |
| **Descrizione generale:** | Three.js è una libreria JavaScript open-source utilizzata per la realizzazione di contenuti 3D per il Web. In particolare, utilizza le API WebGL per integrare ambienti 3D a siti web usando un semplice canvas HTML. |
| **Utilizzo nel progetto:** | Fortemente consigliata dall'azienda proponente, questa libreria viene utilizzata per lo sviluppo della parte 3D del progetto. |
| **React-three/fiber** | |
| **Versione:** | ^v8.15.19 |
| **Documentazione:** | `https://docs.pmnd.rs/react-three-fiber/getting-started/introduction` *(ultimo accesso 05-05-24)* |
| **Descrizione generale:** | React Three Fiber (R3F) è un React renderer per Three.js. Esso permette di costruire componenti, basati sulla logica 3D di Three.js, che siano riutilizzabili e indipendenti. |
| **Utilizzo nel progetto:** | Viene utilizzato R3F per integrare al meglio Three.js con React. |
| **React-three/drei** | |
| **Versione:** | ^v9.102.6 |
| **Documentazione:** | `https://github.com/pmndrs/drei` *(ultimo accesso 05-05-24)* |

| | |
|---|---|
| **Descrizione generale:** | Drei è semplicemente una raccolta di "astrazioni già pronte all'uso", ovvero di componenti generici basati su R3F che sono già completamente funzionali. |
| **Utilizzo nel progetto:** | Viene utilizzato per integrare al meglio Three.js con React, semplificando e velocizzando lo sviluppo di alcune componenti 3D. |
| **Zustand** | |
| **Versione:** | ^v4.5.2 |
| **Documentazione:** | `https://docs.pmnd.rs/zustand/getting-started/introduction` *(ultimo accesso 05-05-24)* |
| **Descrizione generale:** | Zustand è una libreria di gestione dello stato per React. Con un'API semplice e intuitiva, permette di gestire lo stato dell'applicazione in modo efficiente, riducendo la complessità dello sviluppo e migliorando le prestazioni delle applicazioni React. |
| **Utilizzo nel progetto:** | Viene utilizzata per una gestione efficiente, ma comunque semplice, dello stato tra le diverse componenti dell'applicazione. In particolare, per memorizzare lo stato corrente del magazzino, per la conversione tra stato e file di configurazione esterni, oltre che per gestire eventuali errori nella modifica dello stato stesso. |
| **Ant Design** | |
| **Versione:** | ^v5.16.1 |
| **Documentazione:** | `https://ant.design/components/overview/` *(ultimo accesso 05-05-24)* |
| **Descrizione generale:** | Ant Design è una libreria di componenti UI React open-source. Offre una vasta gamma di componenti pronti all'uso per la creazione di interfacce utente moderne e intuitive. Semplifica così lo sviluppo di applicazioni web React e, al contempo, migliora l'aspetto estetico del prodotto finale. |
| **Utilizzo nel progetto:** | Vengono utilizzate le componenti di Ant Design per l'interfaccia grafica. Vengono anche utilizzate le icone fornite da Ant Design per migliorare l'intuitività delle funzionalità all'interno dell'UI. |

Table 3: Tecnologie implementative

## 2.3 Tecnologie per il testing

| | |
|---|---|
| **Jest** | |
| **Versione:** | ^v29.7.0 |
| **Documentazione:** | `https://jestjs.io/docs/getting-started` *(ultimo accesso 05-05-24)* |
| **Descrizione generale:** | Jest è un framework di testing JavaScript comunemente utilizzato per testare applicazioni React e Node.js. In particolare, offre funzionalità integrate per il mocking, l'esecuzione dei test in parallelo e la copertura del codice, migliorando l'affidabilità e la qualità del software. |
| **Utilizzo nel progetto:** | Viene utilizzato per l'implementazione di unit e integration test dell'applicazione, sia per la parte logica sia per l'interfaccia utente. |

| Testing-library/react | |
|---|---|
| **Versione:** | ^v15.0.2 |
| **Documentazione:** | `https://testing-library.com/docs/` *(ultimo accesso 05-05-24)* |
| **Descrizione generale:** | React Testing Library è una libreria di testing per applicazioni React che si concentra sulla validazione dell'interazione dell'utente con l'applicazione. |
| **Utilizzo nel progetto:** | Viene utilizzata assieme a Jest per l'implementazione di test che riguardano componenti di React.js. Inoltre, si precisa l'utilizzo dei pacchetti aggiuntivi jest-dom e user-event per il miglioramento dell'affidabilità e leggibilità dei test. |
| **React-three/test-renderer** | |
| **Versione:** | ^v8.2.1 |
| **Documentazione:** | `https://github.com/pmndrs/react-three-fiber/tree/master/packages/test-renderer` *(ultimo accesso 05-05-24)* |
| **Descrizione generale:** | È una libreria che semplifica il testing di componenti React che utilizzano Three.js per la grafica 3D. Consente di renderizzare e testare componenti 3D, simulare eventi utente e verificare lo stato e il comportamento dei componenti in risposta a interazioni simulate. |
| **Utilizzo nel progetto:** | Viene utilizzata assieme a Jest per l'implementazione di test che riguardano componenti 3D realizzate attraverso React-three/fiber e React-three/drei. |

Table 4: Tecnologie per il testing

## 2.4 Tecnologie per il deployment

| Vercel | |
|---|---|
| **Versione:** | ^v29.7.0 |
| **Documentazione:** | `https://vercel.com/docs` *(ultimo accesso 05-05-24)* |
| **Descrizione generale:** | Vercel è una piattaforma di hosting serverless per applicazioni web e siti statici. Integra anche funzionalità di deployment continuo e collaborazione in tempo reale. |
| **Utilizzo nel progetto:** | Viene utilizzato per una semplice distribuzione del software. In particolare, viene collegato il repository GitHub con il codice sorgente del progetto alla piattaforma. In tal modo, si ha un deployment automatico, garantendo dunque un aggiornamento continuo e senza problemi. |

Table 5: Tecnologie per il deployment

# 3 Architettura logica

Il gruppo ha dato particolare attenzione alla scelta di un'architettura che favorisse una adeguata "*separation of concerns*", specialmente tra la logica di business e la logica di presentazione. Una corretta applicazione di questo principio porta, infatti, ad una maggior coesione e ad una riduzione dell'accoppiamento nel software.

Questa parte del documento esplorerà dunque i pattern adottati per raggiungere tale obiettivo e descriverà i componenti che ne derivano.

## 3.1 Pattern

Questa sezione ha lo scopo di illustrare le motivazioni e i pattern sui quali il gruppo si è basato per la progettazione. L'obiettivo è di fornire una migliore comprensione del significato di ciascun elemento software nel contesto dell'architettura applicata al progetto.

### 3.1.1 Motivazioni e ispirazioni: studio delle tecnologie

Di fatto, le tecnologie scelte non pongono particolari limitazioni sul pattern architetturale da applicare al progetto, perciò la scelta e l'implementazione sono lasciate al team di sviluppo.

Tuttavia, alcuni pattern vengono considerati best practice e/o vengono suggeriti dalle risorse di riferimento delle tecnologie utilizzate. Ad esempio, per la libreria di gestione dello stato Zustand, utilizzata nel progetto, vengono raccomandati alcuni pattern ispirati alle architetture *Flux* e *Redux*. In realtà, è proprio la documentazione stessa dello strumento ad evidenziare la possibilità di utilizzare i principi *Flux* nell'applicazione. Mentre, per quanto riguarda la gestione dell'UI con React.js, anch'esso utilizzato nel progetto, si adotta un'architettura a componenti, come suggerito dalla struttura stessa di React. Inoltre, viene consigliata l'adozione del pattern *Container/Presentational*.

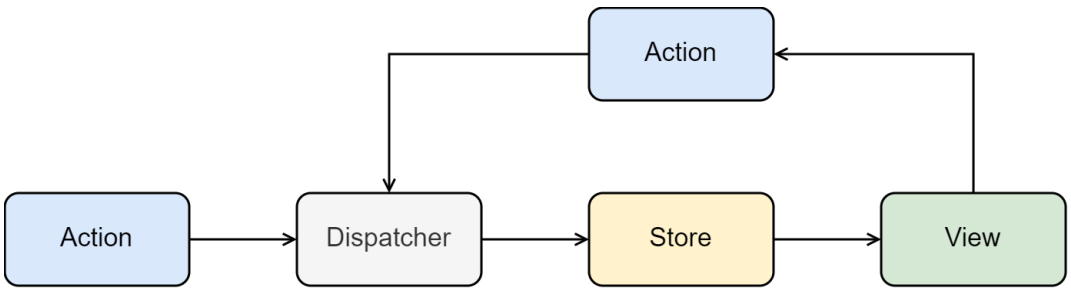
Per maggiori informazioni si rimanda alla sezione Riferimenti Esterni del documento.

### 3.1.2 Le architetture Flux/Redux

#### 3.1.2.1 Descrizione del funzionamento

L'architettura *Flux* nasce come proposta alternativa alle varianti del pattern MVC sviluppate nel corso del tempo per la progettazione di applicazioni web client-side, in particolare con React. L'esigenza alla base di *Flux* era far fronte ai problemi di dipendenze tra le componenti architetturali (legate attraverso un "two-way data binding"), che potevano crescere di complessità, man mano che l'applicazione web cresceva di dimensione, con l'aggiunta, ad esempio, di nuovi modelli e nuove view a quelli già presenti.

La principale differenza che presenta *Flux* è lo "*unidirectional data flow*", e cioè il flusso di dati segue un'unica direzione: nei pattern precedenti (derivati di MVC), i cambiamenti sulla view erano riportati al model e viceversa, ma con *Flux* una modifica segue un unico flusso rigorosamente definito, descritto dal diagramma sottostante.




Gli elementi dell'architettura sono:

- **Store**: contiene lo stato e la logica dell'applicazione. Nella sua implementazione originale, possono esistere diversi store, ciascuno contenente informazioni su uno specifico dominio nell'applicazione;
- **Dispatcher**: funziona da hub centrale per l'applicazione e gestisce la distribuzione delle azioni agli stores;

- **View**: rappresenta l'interfaccia utente, gestisce la visualizzazione dei dati e l'interazione con l'utente. Gli elementi della view rimangono in ascolto per cambiamenti sugli store ed effettuano il re-render se necessario;
- **Actions**: sono oggetti che contengono le informazioni necessarie per compiere un'azione sullo stato dell'applicazione. Possiedono una proprietà *type* che identifica il tipo di azione. Possono originarsi dalla view come risultato dell'interazione utente, ma anche dal server.

Come si può vedere, una modifica può essere effettuata solo tramite actions, le quali entrano nello store attraverso il dispatcher. Quando lo stato nello store cambia, la view effettua un re-render automatico. Questo pattern ben si adatta alle tecnologie scelte, perché rispecchia il funzionamento di React ed aiuta a mantenere il sistema più consistente.

Un'evoluzione di *Flux* è *Redux*, che implementa il pattern applicando alcuni elementi della programmazione funzionale, pur mantenendo gli stessi concetti che stanno alla base del suo precursore. In particolare, *Redux* si differenzia da *Flux* per:

- L'assenza del dispatcher, non necessario perché utilizza funzioni pure, che non hanno bisogno di ulteriori entità che le gestiscano;
- L'assunzione che le informazioni nello stato non vengano mai mutate, ma piuttosto che si crei un nuovo oggetto a partire dall'informazione iniziale;
- La presenza di un singolo store piuttosto che tanti, che quindi funge da "unique source of truth".

#### 3.1.2.2 L'implementazione con Zustand

Come accennato in precedenza, Zustand è per sua natura"unopinionated", tuttavia è possibile (e raccomandato) costruire una struttura che mantiene i concetti delle architetture *Flux* e *Redux*.

Zustand riesce anche a semplificare l'utilizzo di queste, in quanto sfrutta gli hooks di React per realizzare lo state management. Nello specifico, per come è implementata la libreria, non è necessario l'utilizzo di un dispatcher o dei reducer dell'architettura *Redux*. Infatti, tramite appunto l'utilizzo degli hooks, è possibile ottenere un comportamento "flux-like" senza l'utilizzo di ulteriori elementi nell'architettura.

Ed è proprio attraverso questa soluzione che il gruppo ha deciso di basare la progettazione del prodotto finale, utilizzando un'architettura che rispecchia gli standard moderni per le tecnologie scelte, pur mantenendo i vantaggi offerti dalla libreria Zustand. Infatti, oltre a tutto ciò, Zustand offre anche vantaggi in termini di leggibilità e semplicità della sintassi, leggerezza della gestione dello stato, nonché consente l'utilizzo di funzioni puramente JavaScript per definire le azioni di stato.

#### 3.1.2.3 Adozione del pattern nel progetto

Gli elementi che compongono il pattern adottato per l'architettura del progetto sono elencati di seguito, ciascuno con una breve spiegazione.

- **Store**: è la componente unica che contiene lo stato globale dell'applicazione, rappresentando una "*single source of truth*". Lo store contiene dati (sotto forma di tipi primitivi ed oggetti) ed azioni. Seguendo lo *slices pattern*, uno store può essere composto da più slices.
- **Slices**: sono unità individuali che rappresentano un sottoinsieme dello stato globale dell'applicazione, permettendo una migliore modularità nello state management. È bene che una slice contenga dati ed azioni che fanno riferimento ad uno specifico dominio dell'applicazione.
- **Actions**: sono funzioni contenute nelle slices e rappresentano l'unico modo di operare sui dati contenuti nello store. All'interno di una action, si utilizza la funzione *set* per aggiornare lo stato nello store, che effettua un merge in maniera automatica, mantenendo il principio di immutabilità.
- **View**: rappresenta l'interfaccia grafica del progetto e gestisce l'interazione con l'utente. È composta da diversi componenti React, ciascuno dei quali può effettuare l'operazione di "subscribe" ad un particolare sottoinsieme dello stato globale e, in questo caso, il re-render viene effettuato in maniera automatica solo al cambiamento di quella particolare pozione dello stato. Sempre attraverso la view, è possibile chiamare le actions delle slices per aggiornare lo stato dell'applicazione.

### 3.1.3 L'architettura a componenti e il pattern Container/Presentational

#### 3.1.3.1 Descrizione del funzionamento

Nel contesto dello sviluppo di interfacce utente moderne, l'architettura a componenti è diventata uno standard comune. Essa si basa sul concetto di suddividere l'interfaccia utente in componenti (i.e. unità autonome) riutilizzabili e modulari. Ogni componente è infatti responsabile di una specifica parte dell'interfaccia e può contenere sia la logica di presentazione che la logica di gestione dei dati.

Un approccio comune per organizzare i componenti è il pattern *Container/Presentational*. Questo pattern prevede la separazione dei componenti in due categorie principali:

- **Componenti Container:** questi componenti sono responsabili della gestione dei dati e della logica di business dell'applicazione. Essi, infatti, si occupano di recuperare i dati da una fonte esterna, elaborarli e passarli ai componenti Presentational. Sono spesso connessi allo stato dell'applicazione e possono utilizzare i metodi di gestione dello stato per aggiornare dinamicamente l'interfaccia utente.
- **Componenti Presentational:** questi componenti sono responsabili dell'aspetto visivo dell'applicazione e della visualizzazione dei dati forniti loro dai componenti Container. Essi, infatti, non contengono logica di business o di gestione dei dati, bensì si concentrano esclusivamente sulla presentazione dei dati all'utente in modo chiaro e comprensibile.

Riassumendo, in un'architettura basata sul pattern *Container/Presentational*, il componente container gestisce la logica di business, mentre il componente presentazionale si occupa solo dell'aspetto visivo dei dati.

Questo approccio promuove dunque la separazione dei concetti e aderisce al principio di singola responsabilità. Ciò porta ad una maggiore leggibilità, riutilizzabilità e modularità del codice, semplificando anche il processo di testing. Inoltre, questa netta separazione consente di aggiungere e rimuovere componenti presentazionali senza influenzare la logica dell'applicazione, rendendo il sistema più scalabile e facile da mantenere.

#### 3.1.3.2 Implementazione con React.js

React.js offre un set di strumenti e convenzioni che facilitano l'utilizzo di questa architettura.

- **Componenti React:** i componenti sono il fondamento dello sviluppo dell'interfaccia utente in React. In particolare, ogni componente React è una funzione o una classe che restituisce un'interfaccia utente sotto forma di elementi React. È possibile anche strutturare più componenti in un albero gerarchico per creare interfacce utente più complesse.
- **Props e State:** React utilizza props e state per consentire la comunicazione tra i componenti e per gestire lo stato dell'applicazione. Le props sono passate da un componente genitore a un componente figlio e vengono utilizzate per trasmettere dati e configurazioni. Lo state è un oggetto locale a ciascun componente e viene utilizzato per gestire le informazioni dinamiche all'interno di quel componente.
- **Hook di React:** gli hook di React consentono ai componenti funzionali di utilizzare lo state e altre funzionalità di React.

Dunque, con riferimento all'architettura a componenti, quest'ultime sono individuate dalle componenti di React stesse. Per quanto riguarda il pattern *Container/Presentational*, questo è implementato in React tramite:

- l'utilizzo degli hook di React nelle componenti Container per gestire lo stato dell'applicazione e gli effetti collaterali;
- l'utilizzo delle props (ricevuti dai componenti Container) nelle componenti Presentational per renderizzare l'UI in base a tali dati.

#### 3.1.3.3 Adozione del pattern nel progetto

Nel nostro progetto, abbiamo adottato l'architettura a componenti e il pattern *Container/Presentational* per organizzare la parte "View" del sistema. Nello specifico, i componenti Container accedono allo store di Zustand, ricevono dati dalla componente Presentational (tramite funzioni passati come props) e li aggiornano nello store. D'altra parte, i componenti Presentational mostrano i dati ricevuti come props dai componenti Container e trasmettono, a loro volta, i dati ricevuti dall'utente ai componenti Container.

Questo approccio ci ha dunque permesso di separare chiaramente la logica di gestione dei dati dalla logica di presentazione, migliorando la chiarezza e la manutenibilità del nostro codice.

## 3.2 Descrizione dell'architettura

Si riportano di seguito i diagrammi UML delle classi e delle componenti del sistema, corredati da una descrizione che ne chiarisce il significato e la funzionalità.

Per facilitare la comprensione dei diagrammi si utilizzano diversi colori per indicare un diverso tipo di entità:

- **Rosso:** store e slices. In particolare, con riferimento ad un singolo elemento di questi:
  - **Giallo:** per uno stato,
  - **Violetto:** per un'action;
- **Verde:** componenti React;
- **Azzurro**: classi di modello e oggetti.

Si noti inoltre che i metodi *setter* e *getter* per campi privati vengono omessi dai diagrammi per mantenere chiarezza e leggibilità degli stessi, concentrando l'attenzione su aspetti più rilevanti.

### 3.2.1 Store e model

Nel diagramma sottostante vengono mostrate dipendenze dello store, incluse tutte le slice e le componenti React che accedono a queste slice tramite lo store. Vedremo in dettaglio le componenti React nella sezioni 3.2.2, mentre in questa sezione approfondiremo, invece, la parte di modello e composizione dello store stesso.

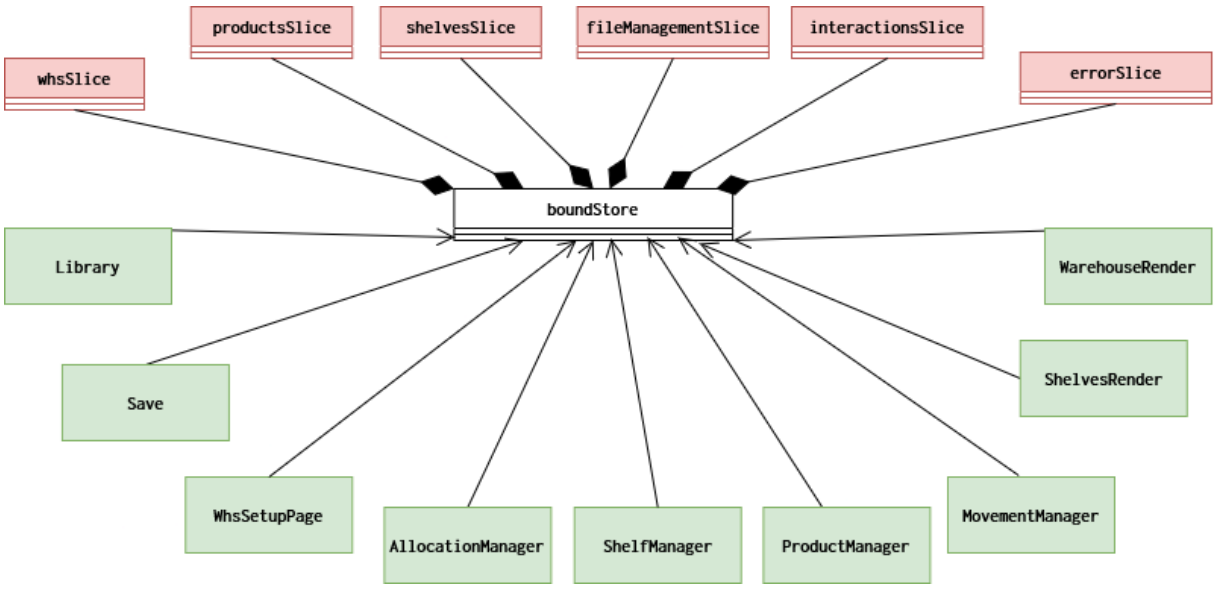



Figure 1: Diagramma UML delle dipendenze dello store

Lo store, denominato **boundStore**, agisce come un contenitore per lo stato globale dell'applicazione, il quale viene composto dalle varie slice definite. Infatti, ogni slice rappresenta una parte specifica dello stato che ha pertinenza su un certo dominio dell'applicazione.

Si esamina di seguito la composizione di ciascuna slice.

#### 3.2.1.1 Slice whsSlice

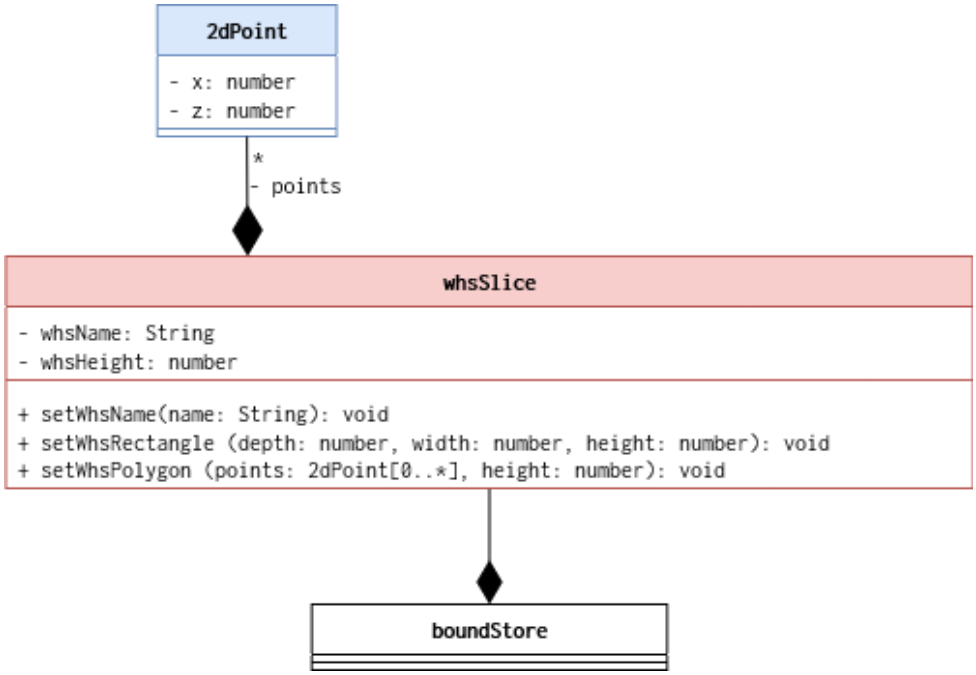



Figure 2: Diagramma UML della whsSlice

Questa slice contiene informazioni sullo stato del magazzino di per sé, in particolare contiene:

- il nome identificativo (**whsName**);
- le dimensioni, rappresentate attraverso un semplice numero (**whsHeight**) per l'altezza e attraverso un array (**points**) di oggetti nel formato $\{x, z\}$ per indicare i vari punti di angolo del magazzino.

Le action contenute in questa slice sono invece:

- **setWhsName** utilizzata per impostare il nome del magazzino;
- due funzioni per impostare le dimensioni del magazzino:
  - **setWhsRectangle**, utilizzata nel caso in cui il magazzino sia creato di default, i.e. con planimetria rettangolare,
  - **setWhsPolygon**, utilizzata nel caso in cui il magazzino sia creato con planimetria personalizzata (poligonale).

Nel caso di successo di un'action, viene aggiornato correttamente lo stato, altrimenti viene impostato un errore nella errorSlice.

#### 3.2.1.2 Slice shelvesSlice

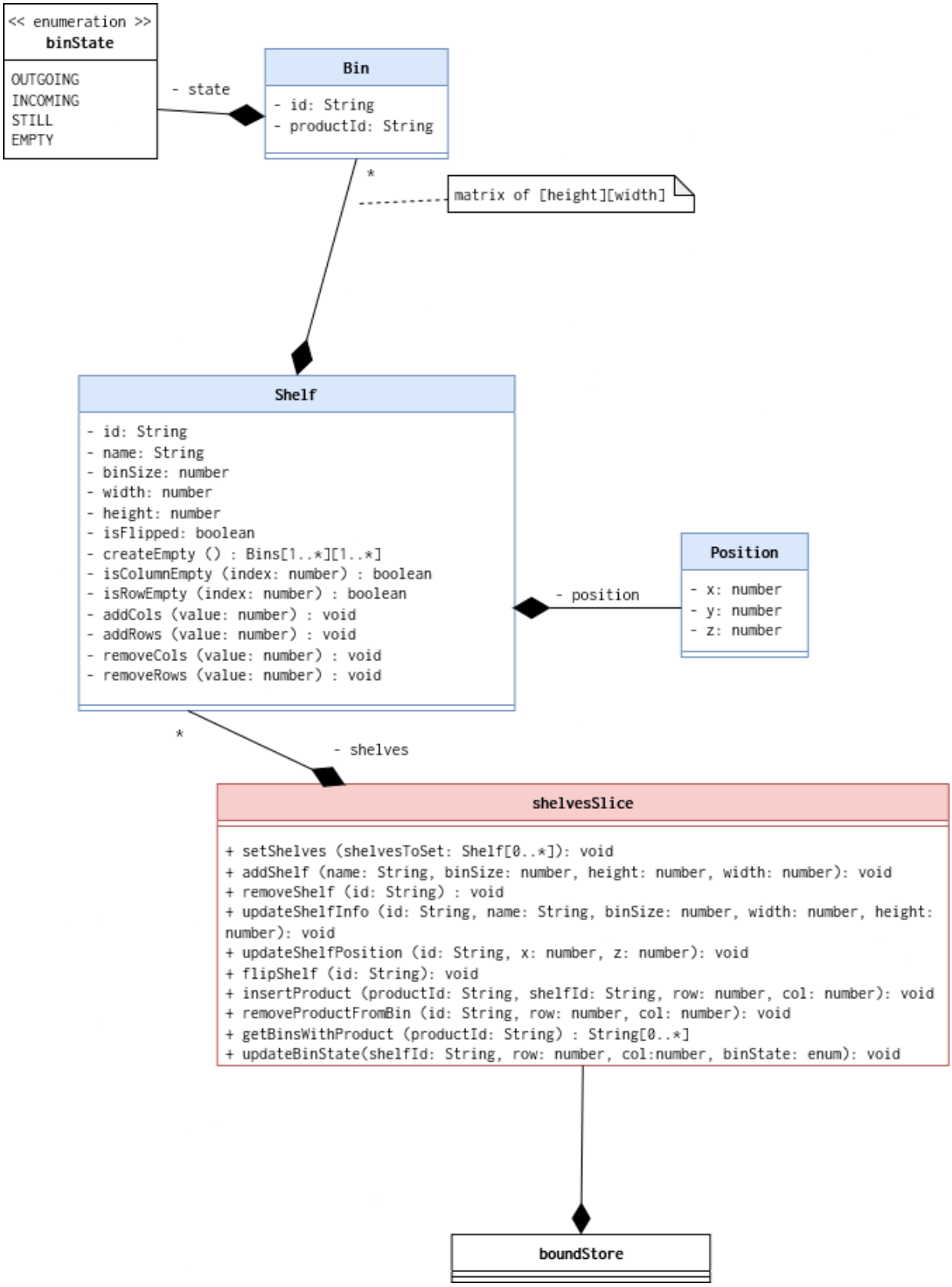



Figure 3: Diagramma UML della shelvesSlice

Questa slice contiene le informazioni sullo stato delle scaffalature, in particolare possiede un elenco delle scaffalature (**shelves**) contenute nel magazzino.

Le scaffalature sono rappresentate individualmente come oggetti della classe "Shelf", la quale raggruppa in sé tutte le informazioni di una scaffalatura, ovvero:

- elementi identificativi, quali **id** univoco (nella forma s_[uuid]) e **nome**;
- dimensioni, ovvero misura dei bin$^{G}$ (**binSize**) e numero di questi in altezza (**height**) e larghezza (**width**);
- disposizione della scaffalatura nel magazzino, i.e. la sua posizione (**position**, rappresentata da un oggetto del tipo $\{x,y,z\}$) e orientamento (**isFlipped**);
- occupazione della scaffalatura, attraverso una matrice di oggetti "Bin". Ciascuno di questi indica lo stato del prodotto contenuto nel bin (**state**) e, se presente, l'id del prodotto (**productId**) contenuto in essi. Questi "Bin" vengono identificati da un **id** univoco formato da [id di Shelf]+[row]+[col], dove "row" e "col" sono gli indici della matrice relativi al bin in questione.

Questa classe "Shelf" fornisce anche tutta una serie di funzioni (visibili nel diagramma sovrastante) che servono a modificare le dimensioni della matrice di oggetti "Bin" ed, eventualmente, generare errori in caso ciò non sia possibile.

Le action contenute in shelvesSlice sono invece:

- **setShelves**, utilizzato per impostare direttamente shelves. In particolare, viene impiegato durante la configurazione del magazzino da file, quando possono essere presenti più scaffalature;
- **addShelf** e **removeShelf**, utilizzate rispettivamente per aggiungere e rimuovere una scaffalatura dallo stato;
- **updateShelfInfo**, **updateShelfPosition** e **flipShelf** utilizzate per aggiornare i diversi dati relativi alla configurazione di una scaffalatura;
- **insertProduct**, **removeProductFromBin** e **updateBinState** per gestire l'inserimento e rimozione di prodotti dai bin della scaffalatura;
- **getBinsWithProduct**, per ottenere una panoramica di tutti i bin presenti nel magazzino che contengono un determinato prodotto.

Nel caso di successo di un'action, viene aggiornato correttamente lo stato, altrimenti viene impostato un errore nella errorSlice.

#### 3.2.1.3 Slice productsSlice

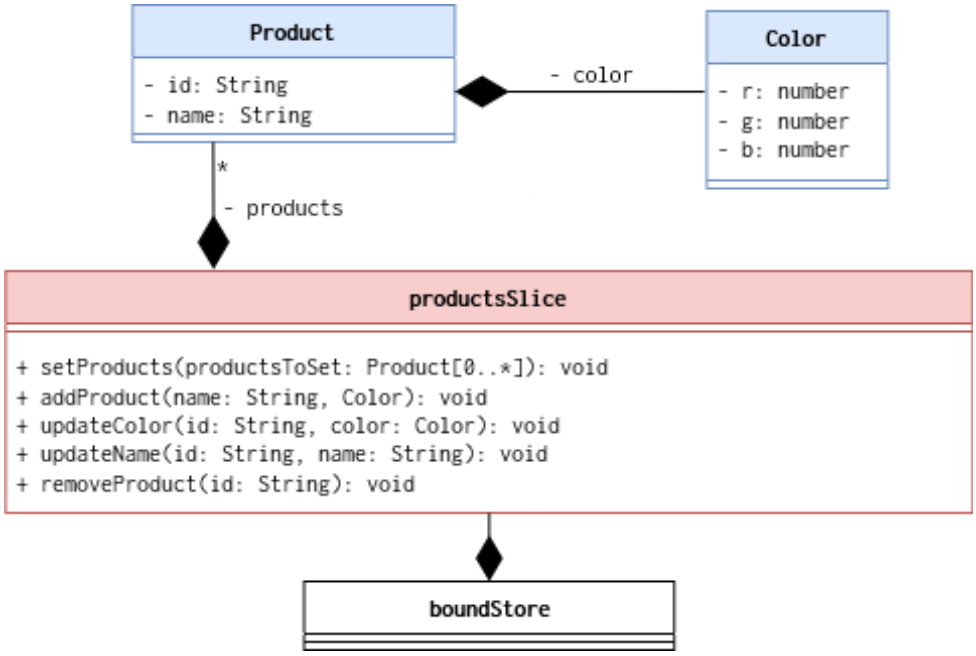



Figure 4: Diagramma UML della productsSlice

Questa slice contiene le informazioni sullo stato dei prodotti, in particolare possiede un elenco dei prodotti (**products**) che sono distribuiti nel magazzino.

I prodotti sono modellati come oggetti della classe "Product", la quale raggruppa in sé tutte le informazioni di un prodotto, ovvero:

- elementi identificativi, quali **id** univoco (nella forma p_[uuid]) e **nome**;
- colore del prodotto, rappresentato da un oggetto nel formato $\{r,g,b\}$.

Si tenga presente che questi oggetti rappresentano "etichette" più che prodotti fisici. Essi infatti possono esistere senza essere fisicamente nel magazzino, oppure posso essere presenti nel magazzino in una o più copie. Il singolo prodotto fisico è dunque simboleggiato dal contenuto di un determinato bin.

Le action contenute in productsSlice sono invece:

- **setProducts**, utilizzato per impostare direttamente products. In particolare, viene impiegato durante la configurazione del magazzino da file, quando possono essere presenti più prodotti;
- **addProduct** e **removeProduct**, utilizzate rispettivamente per aggiungere e rimuovere completamente un prodotto dallo stato;
- **updateName** e **updateColor** utilizzate per aggiornare i diversi dati relativi ad un prodotto;

Nel caso di successo di un'action, viene aggiornato correttamente lo stato, altrimenti viene impostato un errore nella errorSlice.

#### 3.2.1.4 Slice interactionsSlice

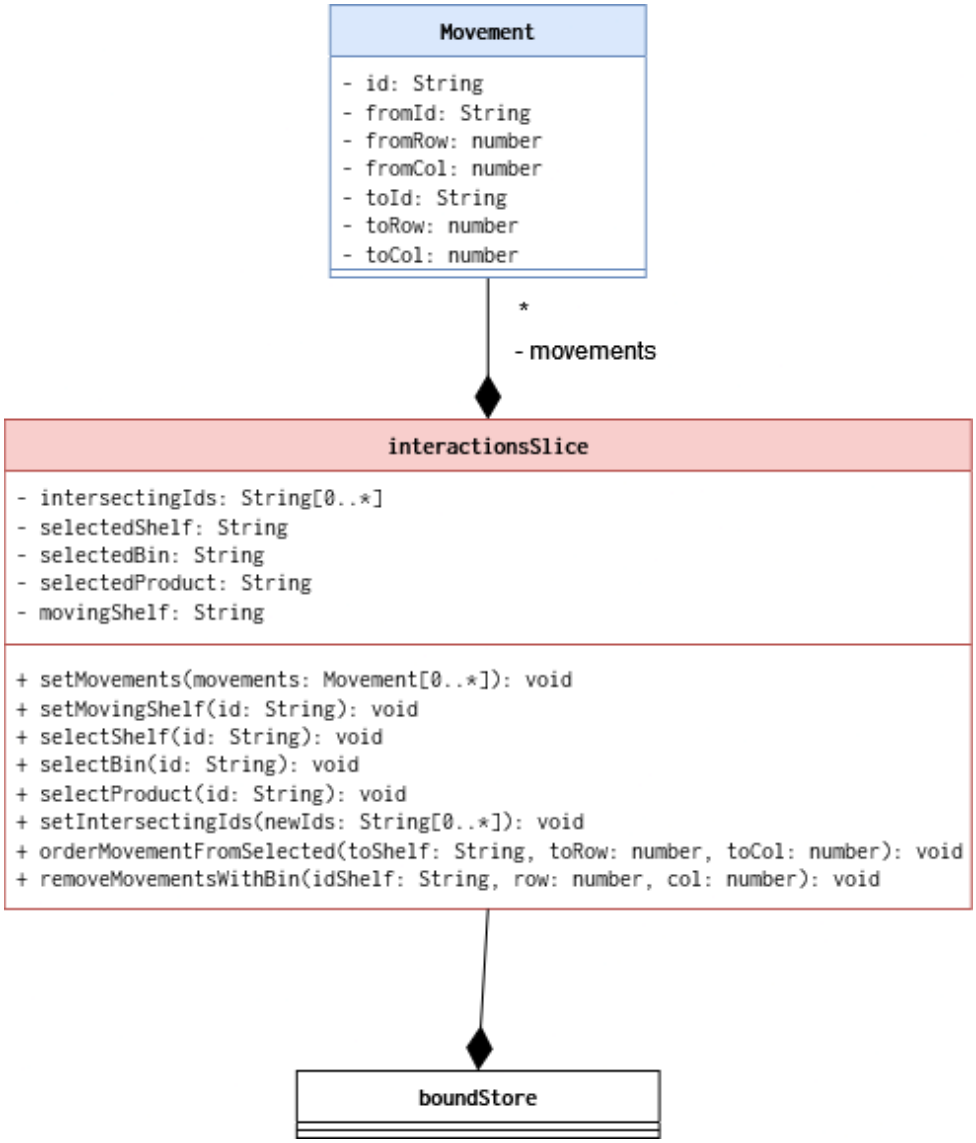



Figure 5: Diagramma UML dell'interactionsSlice

Questa slice contiene le informazioni sulle attuali interazioni dell'utente con il magazzino. In particolare contiene:

- un array (**intersectingIds**) degli elementi del magazzino tra cui vi è una collisione causata, ad esempio, dal movimento di una scaffalatura;
- l'id dell'elemento attualmente selezionato (esclusivamente uno tra **selectedShelf**, **selectedBin** e **selectedProduct**, gli altri sono invece posti a *null*);
- l'id della scaffalatura (**movingShelf**) che è in movimento al momento (se presente);
- un array (**movements**) delle richieste di movimento (di prodotti) pendenti.
  - Una singola di queste richieste è rappresentata da un oggetto "Movement", identificato da un **id** (nel formato m_[uuid]). Esso contiene le informazioni sull'attuale posizione del prodotto da movimentare (**fromId** , **fromRow** e **fromCol**, che nella forma [fromId]+[fromRow]+[fromCol]

rappresentano il bin di origine) e sulla destinazione dopo il movimento, se completato con successo (**toId** , **toRow** e **toCol**, che nella forma [toId]+[toRow]+[toCol] rappresentano il bin di arrivo).

Le action contenute in interactionsSlice sono invece:

- **setMovements**, utilizzato per impostare direttamente movements. In particolare, viene impiegato durante la configurazione del magazzino da file, quando possono essere presenti più movimentazioni pendenti;
- **selectShelf**, **selectProduct** e **selectBin**, per impostare rispettivamente una scaffalatura, un prodotto o un bin come elemento correntemente selezionato e, al contempo, deselezionare qualsiasi altro elemento eventualmente già selezionato;
- **setMovingShelf**, per impostare una scaffalatura come "in movimento";
- **setIntersectingIds**, per segnalare tutti gli elementi correntemente interessati da una collisione;
- **orderMovementFromSelected** e **removeMovementsWithBin**, utilizzate rispettivamente per aggiungere e rimuovere un movimento di un prodotto dallo stato.

Nel caso di successo di un'action, viene aggiornato correttamente lo stato, altrimenti viene impostato un errore nella errorSlice.

#### 3.2.1.5 Slice fileManagementSlice

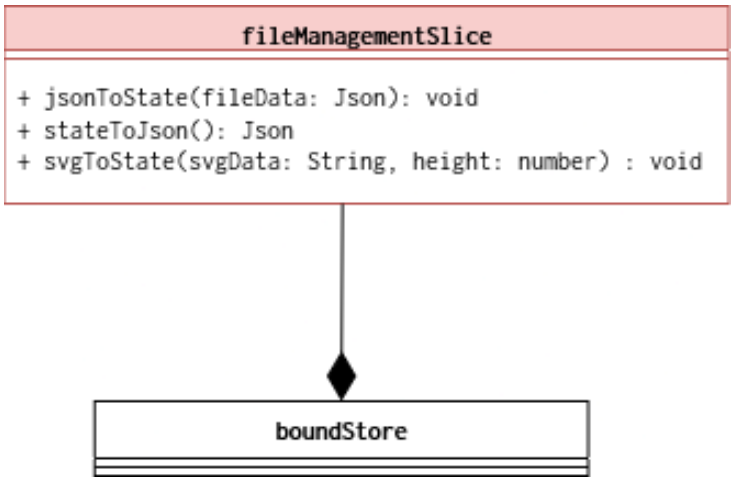



Figure 6: Diagramma UML della fileManagementSlice

Questa slice funge da tramite tra le funzioni dello stato e i dati di configurazione in formato JSON ed SVG. In particolare, essa si occupa di ricevere i dati JSON e chiamare le funzioni appropriate dello stato per aggiornare le informazioni (attraverso l'action **jsonToState**). Viceversa, quando vi è la necessità di trasferire dati dallo stato al formato JSON, per salvare la configurazione del magazzino, questa slice si occupa di tale conversione (attraverso l'action **stateToJson**). Analogamente, gestisce anche la conversione dei dati dal formato SVG, utilizzato per planimetrie personalizzate del magazzino, allo stato (attraverso l'action **svgToState**).

Si noti che le funzioni all'interno di questa slice si occupano esclusivamente della conversione dei dati e dell'aggiornamento dello stato in base ad essi. I controlli di conformità di questi dati non sono effettuati da queste funzioni, ma vengono gestiti tramite chiamate a API effettuate direttamente nelle componenti.

#### 3.2.1.6 Slice errorSlice

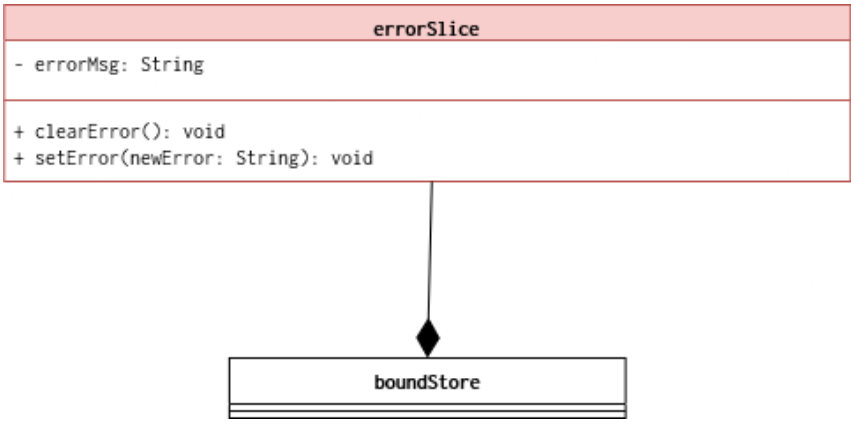



Figure 7: Diagramma UML dell'errorSlice

Questa slice contiene un eventuale messaggio di errore (**errorMsg**) e le corrispettive action per impostarlo o eliminarlo (**setError** e **clearError**). L'errore viene impostato da altre slice dello stato in caso di errori nell'aggiornamento dello stato. Le componenti, successivamente, controllano errorMsg per verificare se l'operazione richiesta è andata a buon fine.

### 3.2.2 Componenti React

Per facilitare la lettura dei diagrammi delle componenti è stato scelto di organizzarli per feature in modo che ogni diagramma rappresenti le componenti che permettono di implementare aspetti chiave dell'applicazione.

I diagrammi prodotti che rappresentano funzionalità specifiche sono:

- **Diagramma feature/setup** (3.2.2.1): include le macro-componenti che costituiscono la pagina iniziale di configurazione del magazzino.
- **Diagramma feature/wmsLayout** (3.2.2.2): include le macro-componenti di layout presenti nella pagina principale, i.e. le componenti che costituiscono header e sidebar.
- **Diagramma feature/actions** (3.2.2.3): include tutte le macro-componenti che permettono all'utente di interagire con l'applicativo attraverso input testuali o pulsanti, nonché i dati informativi che si modificano in risposta alle azioni dell'utente sul magazzino.
- **Diagramma feature/render** (3.2.2.4): include tutte le macro-componenti che costituiscono gli elementi dell'ambiente 3D.

#### 3.2.2.1 Feature/setup

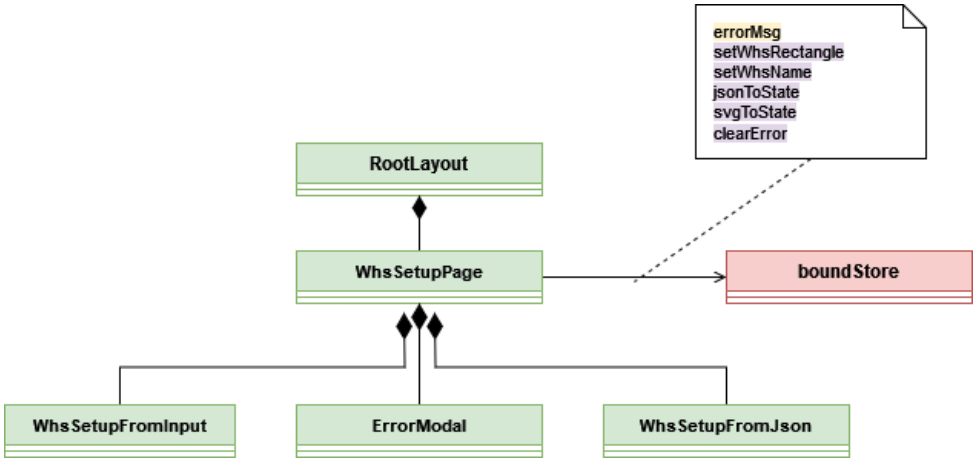



Figure 8: Diagramma UML feature/setup

**RootLayout**

**Tipo:** Presentational
**Interazioni con utente:**
Nessuna.
**Composizione**:
Contiene tutte le componenti di tutte le pagine dell'applicativo.
**Descrizione e funzionamento:**
Funge da contenitore principale per le informazioni di base e i tag HTML (<*html*> e <*body*>) utilizzati in tutte le pagine dell'applicazione. In altre parole, *RootLayout* contiene i metadati e gli elementi HTML comuni a tutte le pagine, fornendo una struttura di base uniforme per l'intera applicazione.

**WhsSetupPage**

**Tipo:** Container
**Accesso allo store:**
Chiama le action di configurare del magazzino (*setWhsRectangle*, *setWhsName*, *svgToState* o *jsonToState* a seconda dell'input fornito). Inoltre, rimane in ascolto per eventuali errori nell'esecuzione di tali operazioni, controllando il valore di *errorMsg* nello store. Utilizza anche l'action *clearError* per resettare l'errore e lasciare all'utente la possibilità di modificare quanto inserito con dati corretti.
**Composizione:**
*WhsSetupPage* è costituita dalla componente *Tabs* di Ant Design, dove ogni tab contiene uno dei moduli di configurazione del magazzino, i.e. *WhsSetupFromInput* e *WhsSetupFromJson*. Questo layout permette di visualizzare un solo form alla volta. Quando un form viene inviato, viene disabilitata l'altra tab. È inoltre presente una componente *ErrorModal* per la visualizzazione di eventuali errori.
**Descrizione e funzionamento:**
Costituisce la pagina iniziale di configurazione del magazzino. Aggiorna lo stato del magazzino nello store con le informazioni ricevute da *WhsSetupFromInput* o *WhsSetupFromJson*. In caso di errori in tale aggiornamento, apre invece una finestra modale *ErrorModal*.

#### WhsSetupFromInput

**Tipo:** Presentational *
**Interazione con l'utente:**
L'utente inserisce nome e dimensioni del magazzino nelle apposite caselle di input. Di default, le dimensioni includono altezza, larghezza e profondità, affinché sia possibile creare un magazzino quadrato/rettangolare. Tuttavia, gli utenti hanno anche la possibilità di caricare una planimetria personalizzata tramite un file SVG. Quando viene caricato tale file, le caselle di input per la larghezza e la profondità del magazzino vengono disabilitate. L'utente deve però inserire in ogni caso nome e altezza, poiché solo le dimensioni del pavimento del magazzino sono definite dalla planimetria caricata.
**Composizione:**
Si tratta di una componente *Form* di Ant Design che ha come *Form.item* una componente *Input* per il nome del magazzino, delle componenti *InputNumber* per le dimensioni e una *Upload* per il caricamento di un solo file SVG.
**Descrizione e funzionamento:**
Consente la configurazione del solo spazio del magazzino e dunque si utilizza nel caso in cui non si abbia ancora creato e salvato alcun magazzino. L'utente compila il modulo seguendo le indicazioni fornite. Al momento poi dell'invio del modulo (se compilato correttamente e completamente), viene richiamata una funzione di *WhsSetupPage* con i dati caricati e un valore indicante il caricamento o meno di planimetria personalizzata. Quest'ultimo segnala alla componente quale action (*setWhsRectangle* o *svgToState*) richiamare per la configurazione del pavimento del magazzino.

#### WhsSetupFromJson

**Tipo:** Presentational *
**Interazione con l'utente:**
L'utente carica un file .json che include la configurazione del magazzino, i dati relativi agli elementi presenti al suo interno, nonché le richieste di movimentazione pendenti.
**Composizione:**
Si tratta di una componente *Form* di Ant Design che ha come *Form.item* una componente *Upload* per il caricamento di un solo file JSON.
**Descrizione e funzionamento:**
Consente la configurazione dell'intero stato del magazzino e dunque si utilizza nel caso in cui si abbia già creato e salvato un magazzino in precedenza. L'utente compila il modulo seguendo le indicazioni fornite. Al momento dell'invio del modulo (se compilato correttamente e completamente), viene richiamata una funzione di *WhsSetupPage* con i dati caricati. Quest'ultima si occuperà di aggiornare lo stato tramite l'action *jsonToState*.

#### ErrorModal

**Tipo:** Presentational
**Interazione con l'utente:**
L'utente può chiudere la finestra attraverso un apposito bottone.
**Composizione:**
È costituita dalla componente *Modal* di Ant Design.
**Descrizione e funzionamento:**
Riceve il contenuto dello stato *errorMsg* dalla componente Container padre e lo mostra come messaggio della finestra modale. Una volta chiusa la finestra viene richiamata una funzione della componente Container che si occupa di cancellare l'errore dallo stato (tramite *clearError*) e riabilitare alla modifica la parte che ha causato l'errore.

*Non accede allo store, tuttavia si occupa anche di una validazione preliminare di dati forniti da file esterni tramite il richiamo di un'API. Ques'API controlla che la struttura del file sia corretta, non occupandosi invece della correttezza dei dati stessi. Si è deciso di mantenere qui questa parte per minimizzare la complessità del componente padre *WhsSetupPage*, mantenendo quest'ultimo più snello e focalizzato sulla gestione dello stato globale dell'applicazione e sul coordinamento delle diverse sue componenti figlie. Inoltre, chiamare l'API direttamente in questa componente consente di fornire un feedback più immediato all'utente tramite messaggi di errore (*message.error* di Ant Design).

#### 3.2.2.2 Feature/wmsLayout

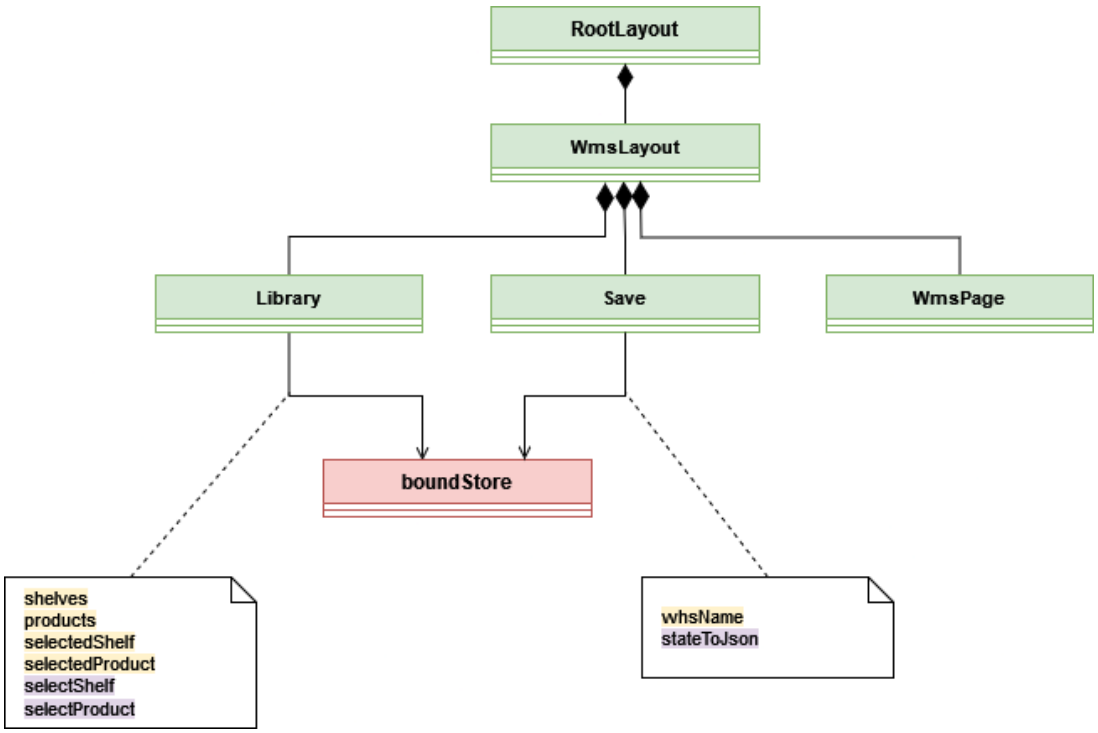



Figure 9: Diagramma UML feature/wmsLayout

**RootLayout**

Si veda in 3.2.2.1 per le specifiche.

**WmsLayout**

**Tipo:** Presentational
**Interazione con l'utente:**
L'utente può cliccare un bottone per aprire e chiudere la sidebar.
**Composizione:**
È costituito come componente *Layout* di Ant Design, il quale a sua volta è costituito da *Header*, *Sider* e *Content*. La parte di *Header* contiene un bottone di apertura e chiusura della sidebar, il logo e la componente *Save*. La parte *Sider*, i.e. la sidebar, contiene la componente *Library*. Infine, la parte di *Content* contiene la pagina principale *WmsPage*.
**Descrizione e funzionamento:**
Funge da contenitore secondario per la pagina principale dell'applicazione (*WmsPage*) ed eventuali sue future pagine figlie. Si innesta infatti a RootLayout.

**Library**

**Tipo:** Container **
**Accesso allo store:**
*Library* rimane in ascolto dello store per ottenere la lista aggiornata dei prodotti e scaffalature presenti nel magazzino (tramite gli stati *products* e *shelves*). *Library* rimane anche in ascolto di *selectedProduct* e *selectedShelf* per avere sempre l'ultimo elemento selezionato e poterlo così evidenziare nell'elenco. Richiama inoltre *selectProduct* o *selectShelf* se l'utente clicca nella lista, rispettivamente, un prodotto o una scaffalatura.
**Composizione:**
È costituita principalmente da una componente *Tree* di Ant Design che comprende la lista di tutte le

**In questo caso la parte Presentational è delegata alle componenti di Ant Design sue figlie.

scaffalature e prodotti presenti nel magazzino. Contiene inoltre una componente *Search*, sempre di Ant Design, che consente di ricercare per nome un prodotto o scaffalatura in questa lista, filtrando l'albero stesso.
**Descrizione e funzionamento:**
Visualizza l'elenco di tutte le scaffalature e prodotti nel magazzino. L'utente può anche ricercare uno specifico elemento tramite nome attraverso la barra di ricerca fornita. Inoltre, se uno degli elementi della lista è selezionato, o tramite click o tramite selezione da render (quest'ultima solo per scaffalature, i prodotti in sé infatti non possono essere selezionati da render), questo viene evidenziato nella lista.

#### Save

**Tipo:** Container **
**Accesso allo store:**
Se il bottone viene cliccato, viene chiamata l'action *stateToJson*. *Save* utilizza anche lo stato *whsName* per denominare il file .json.
**Composizione:**
È costituita principalmente da una componente *Button* di Ant Design.
**Descrizione e funzionamento:**
Viene utilizzato per salvare lo stato del magazzino in un file .json. Ricevuti i dati dallo store, viene creato un file .json con questi dati (denominato [*whsName*].json). L'utente può dunque salvare in locale la configurazione attuale del magazzino, compresi tutti gli elementi al suo interno e le movimentazioni pendenti.

#### WmsPage

**Tipo:** Presentational
**Interazione con l'utente:**
Nessuna.
**Composizione:**
Contiene il *Render* e tutte le altre componenti con cui l'utente può interagire per apportare modifiche al magazzino o semplicemente visualizzarne lo stato. In particolare, contiene anche le componenti *Tools*, *ShelfManager*, *ProductManager*, *AllocationManager* e *MovementManager*.
**Descrizione e funzionamento:**
Costituisce la pagina principale dell'applicazione. Il suo ruolo principale è infatti quello di contenitore e coordinatore della visibilità delle sue componenti figlie. Per far ciò utilizza gli hooks di React.

**In questo caso la parte Presentational è delegata alle componenti di Ant Design sue figlie.

#### 3.2.2.3 Feature/actions

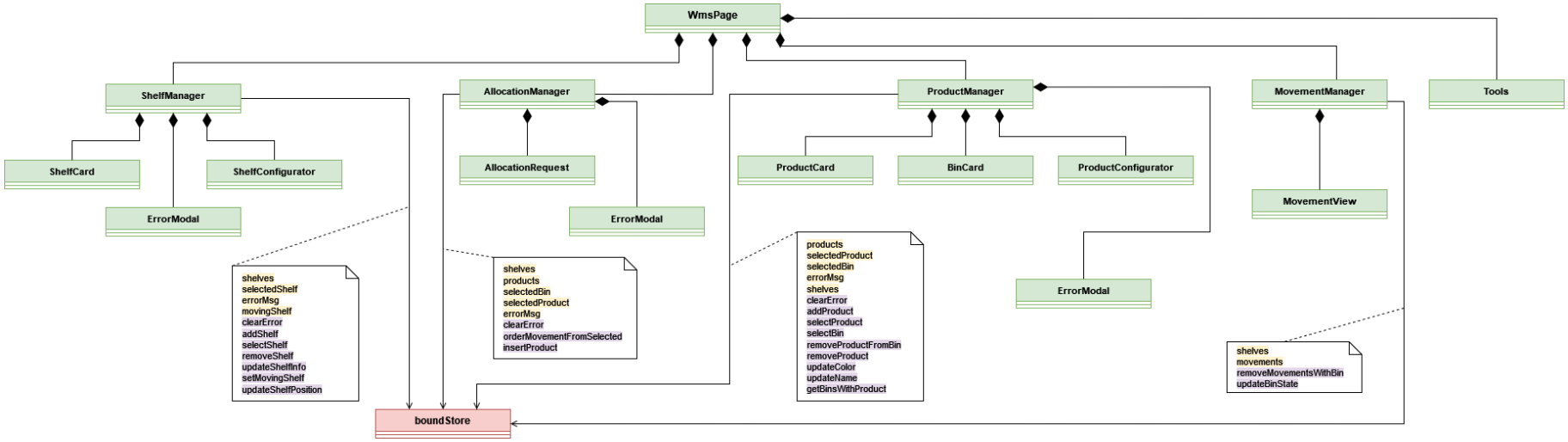



Figure 10: Diagramma UML feature/actions

#### WmsPage

Si veda in 3.2.2.2 per le specifiche.

#### ShelfManager

**Tipo:** Container
**Accesso allo store:**
Accede allo store per avere lo stato aggiornato di tutte le scaffalature (*shelves*) e/o per modificarlo in risposta alle azioni compiute dall'utente attraverso le componenti figlie (a seconda dell'azione attraverso *addShelf*, *removeShelf*, *updateShelfInfo* o *updateShelfPosition*). Oltre a ciò, utilizza *selectedShelf*, *movingShelf*, *selectShelf* e *setMovingShelf* per coordinare le operazioni con il render. Utilizza *selectedShelf* anche per aprire la *ShelfCard* corrispondente. Inoltre, rimane in ascolto per eventuali errori nell'esecuzione di tali operazioni, controllando il valore di *errorMsg* nello store. Utilizza anche l'action *clearError* per resettare l'errore e lasciare all'utente la possibilità di modificare quanto inserito con dati corretti.
**Composizione:**
Contiene le componenti *ShelfCard* per la visualizzazione dei dati di una scaffalatura, *ShelfConfigurator* per l'aggiunta/modifica di una scaffalatura e *ErrorModal* per la visualizzazione di eventuali errori.
**Descrizione e funzionamento:**
È responsabile della gestione delle scaffalature nel magazzino, inclusa l'aggiunta, la modifica e la semplice visualizzazione dei dati a essa relativi. Nello specifico, se è presente una scaffalatura selezionata (*selectedShelf*) ne ricava le informazioni e li visualizza tramite la componente figlia *ShelfCard*. Altrimenti, se riceve indicazione da *WmsPage* o dalla *ShelfCard* stessa, apre *ShelfConfigurator* rispettivamente per l'aggiunta o per la modifica di una scaffalatura. Inoltre, se viene segnalata l'eliminazione di una scaffalatura da parte di *ShelfCard*, *ShelfManager* provvede alla sua rimozione. In caso di errori nell'aggiornamento dello stato, apre anche una componente *ErrorModal*.

#### ShelfConfigurator

**Tipo:** Presentational
**Interazione con l'utente:**
In caso di aggiunta di una scaffalatura, l'utente inserisce nome e dimensioni della scaffalature nelle apposite caselle di input. In caso di modifica, si limita a modificare le parti desiderate. Si procede poi con l'eventuale modifica della posizione da render.
**Composizione:**
È costituito da un componente *Drawer* di Ant Design, contenente a sua volta un componente *Steps*, sempre di Ant Design. Quest'ultimo divide il processo di creazione/modifica in due parti: una prima parte dedicata al nome e alle misure, disponibile tramite una componente *Form*, e una seconda parte per il posizionamento. Quest'ultimo può avvenire solamente tramite render quindi qui viene riportata solo una semplice descrizione di come eseguirlo. Per quanto riguarda il *Form*, invece, questo contiene i *Form.item* per il nome (tipo *Input*), per la dimensione del bin (tipo *InputNumber*), per la capacità in altezza (tipo *InputNumber*) e larghezza (tipo *InputNumber*). Sono presenti anche alcuni pulsanti per chiudere il *Drawer* annullando le modifiche oppure per procedere con il step successivo/confermare le modifiche.
**Descrizione e funzionamento:**
Viene utilizzato per aggiungere una scaffalatura al magazzino o modificarne una già presente. In questo secondo caso, il *Form* fornito è già completato con i dati correnti della scaffalatura selezionata (*selectedShelf*). L'utente compila/modifica il modulo seguendo le indicazioni date. Al momento poi dell'invio del modulo (se compilato correttamente e completamente), viene richiamata una funzione di *ShelfManager* con i dati caricati. Quest'ultima controlla attraverso un hook di React, precedentemente impostato, se i dati inviati riguardano una scaffalatura esistente e procede a richiamare l'action dello store corrispondente (*addShelf* o *updateShelfInfo*). Alla conferma finale della creazione, dopo il posizionamento da render, *ShelfConfigurator* richiama un'altra funzione di *ShelfManager* che aggiorna la posizione della scaffalatura nello store (tramite *updateShelfPosition*).

### ShelfCard

**Tipo:** Presentational
**Interazione con l'utente:**
L'utente può cliccare dei pulsanti per modificare oppure per eliminare la scaffalatura.
**Composizione:**
È costituita da una componente *Card* di Ant Design, il cui titolo è dato dal nome della scaffalatura e la descrizione ne tutte contiene le dimensioni. Inoltre, contiene due pulsanti utilizzati per eliminare la scaffalatura oppure modificarla.
**Descrizione e funzionamento:**
Viene utilizzata principalmente per visualizzare i dati relativi alla scaffalatura correntemente selezionata. Quest'ultimi sono ricevuti come props da *ShelfManager*. L'utente può anche decidere di eliminare o modificare la scaffalatura tramite alcuni pulsanti. Questi richiamano delle funzioni di *ShelfManager* che aggiornano lo stato con l'eliminazione della scaffalatura nel primo caso (tramite *removeShelf*) e aprono *ShelfConfigurator* per la modifica nel secondo.

### ProductManager

**Tipo:** Container
**Accesso allo store:**
Accede allo store per avere lo stato aggiornato di tutti i prodotti (*products*) e bin (attraverso *shelves*) e/o per modificarlo in risposta alle azioni compiute dall'utente attraverso le componenti figlie (a seconda dell'azione attraverso *addProduct*, *removeProduct*, *updateName*, *updateColor* o *removeProductFromBin*). Oltre a ciò, utilizza *selectedProduct*, *selectedBin*, *selectProduct* e *selectBin* per coordinare la visualizzazione e le operazioni tra le sue componenti figlie. Inoltre, utilizza *getBinsWithProduct* per ottenere una lista di tutti i bin in cui il prodotto è presente. Infine, rimane in ascolto per eventuali errori nell'esecuzione delle operazioni sopra-elencate, controllando il valore di *errorMsg* nello store. Utilizza anche l'action *clearError* per resettare l'errore e lasciare all'utente la possibilità di modificare quanto inserito con dati corretti.
**Composizione:**
Contiene le componenti *ProductCard* per la visualizzazione dei dati generali del prodotto, *BinCard* per la visualizzazione dei dati del prodotto in un specifico bin, *ProductConfigurator* per l'aggiunta/modifica di un prodotto e *ErrorModal* per la visualizzazione di eventuali errori.
**Descrizione e funzionamento:**
È responsabile della gestione dei prodotti nel magazzino, inclusa l'aggiunta, la modifica e la semplice visualizzazione dei dati a esso relativi. Nello specifico, se è presente un bin selezionato (*selectedBin*) ne ricava le informazioni e le visualizza tramite *BinCard*. Analogamente, se è presente un prodotto selezionato (*selectedProduct*), ne ricava le informazioni e le visualizza tramite la componente figlia *ProductCard*. Altrimenti, se riceve indicazione da *WmsPage* o dalla *ProductCard* stessa, apre *ShelfConfigurator* rispettivamente per l'aggiunta o per la modifica di un prodotto. Inoltre, se viene segnalata l'eliminazione di un prodotto da parte di *ProductCard* o da *BinCard*, *ProductManager* provvede alla sua rimozione (rispettivamente tramite *removeProduct*, che rimuove completamente il prodotto dal magazzino, e *removeProductFromBin*, che lo rimuove solamente dal bin selezionato). Alla stessa maniera, se viene segnalato il posizionamento di un prodotto da parte di *ProductCard* o *BinCard*, *ProductManager* provvede a tale operazione segnalando a *WmsPage* la volontà di aprire l'*AllocationManager* (nel primo caso per posizionamento di un prodotto e nel secondo per una richiesta di movimentazione). In caso di errori nell'aggiornamento dello stato, apre anche una componente *ErrorModal*.

### ProductConfigurator

**Tipo:** Presentational
**Interazione con l'utente:**
In caso di aggiunta di un prodotto, l'utente inserisce nome e sceglie il colore del prodotto negli appositi spazi di input. In caso di modifica, si limita a modificare le parti desiderate.
**Composizione:**
È costituito da un componente *Drawer* di Ant Design, contenente a sua volta una componente *Form*, Quest'ultimo contiene i *Form.item* per il nome (tipo *Input*), e per il colore (tipo *ColorPicker*). Sono presenti anche due pulsanti per chiudere il *Drawer* annullando le modifiche oppure per confermare la creazione.

**Descrizione e funzionamento:**
Viene utilizzato per aggiungere un prodotto al magazzino o modificarne uno già presente. In questo secondo caso, il *Form* fornito è già completato con i dati correnti del prodotto selezionato (*selectedProduct*). L'utente compila/modifica il modulo seguendo le indicazioni date. Al momento poi dell'invio del modulo (se compilato correttamente e completamente), viene richiamata una funzione di *ProductManager* con i dati caricati. Quest'ultima controlla attraverso un hook di React, precedentemente impostato, se i dati inviati riguardano un prodotto esistente e procede a richiamare l'action dello store corrispondente (*addProduct* oppure *updateName*+*updateColor*).

#### ProductCard

**Tipo:** Presentational
**Interazione con l'utente:**
L'utente può cliccare dei pulsanti per modificare, per eliminare oppure per posizionare un prodotto.
**Composizione:**
È costituita da una componente *Card* di Ant Design, il cui titolo è dato dal nome del prodotto e la descrizione ne contiene il colore e una componente *Table* di Ant Design con la lista di tutti i bin in cui il prodotto è allocato. Inoltre, contiene tre pulsanti utilizzati per eliminare, modificare e allocare il prodotto.
**Descrizione e funzionamento:**
Viene utilizzata principalmente per visualizzare i dati relativi al prodotto correntemente selezionato nella sua interezza. Questi dati sono ricevuti come props da *ProductManager*. L'utente può anche decidere di eliminare, modificare oppure allocare un prodotto tramite alcuni pulsanti. Questi richiamano delle funzioni di *ProductManager* che aggiornano lo stato con l'eliminazione della scaffalatura nel primo caso (tramite *removeProduct*), aprono *ProductConfigurator* per la modifica nel secondo oppure segnalano ad *AllocationManager* la volontà di allocare un prodotto nel terzo. Inoltre, se l'utente clicca su uno specifico bin in cui è allocato il prodotto nella *Table* della *ProductCard*, sarà possibile selezionare tale bin tramite *ProductManager* e aprire dunque la *BinCard* a esso relativa.

#### BinCard

**Tipo:** Presentational
**Interazione con l'utente:**
L'utente può cliccare dei pulsanti per eliminare il prodotto presente nel bin oppure per richiederne lo spostamento.
**Composizione:**
È costituita da una componente *Card* di Ant Design, il cui titolo è dato dall'id del bin e la descrizione ne tutte contiene lo stato e il prodotto inserito (se presente). Inoltre, contiene due pulsanti utilizzati per eliminare il prodotto contenuto (se presente) oppure per richiederne lo spostamento (sempre se è presente un prodotto).
**Descrizione e funzionamento:**
Viene utilizzata principalmente per visualizzare i dati relativi ad un prodotto specifico correntemente selezionato (*selectedBin*). Quest'ultimi sono ricevuti come props da *ProductManager*. L'utente può anche decidere di eliminare o richiederne lo spostamento tramite alcuni pulsanti. Questi richiamano delle funzioni di *ProductManager* che aggiornano lo stato con l'eliminazione del prodotto dal bin nel primo caso (tramite *removeProductFromBin*) e segnalano ad *AllocationManager* la volontà di movimentare un prodotto nel secondo. Nel caso in cui il bin sia vuoto questi pulsanti richiamano un *message.error* di Ant Design che informa l'utente dell'impossibilità di eseguire tali operazioni.

#### AllocationManager

**Tipo:** Container
**Accesso allo store:**
Accede allo store per avere lo stato aggiornato di tutte le scaffalature (*shelves*) e prodotti (*products*), necessario per poter fornire tutti i dati ad *AllocationRequest*. Accede allo store anche per modificare lo stato in risposta alle azioni compiute dall'utente attraverso *AllocationRequest* (a seconda dell'azione attraverso *orderMovementFromSelected* o *insertProduct*). Oltre a ciò, rimane in ascolto di *selectedBin* e *selectedProduct* per poter aprire l'*AllocationRequest* correttamente. Infine, rimane in ascolto anche per

eventuali errori nell'esecuzione di tali operazioni, controllando il valore di *errorMsg* nello store. Utilizza anche l'action *clearError* per resettare l'errore e lasciare all'utente la possibilità di modificare quanto inserito con dati corretti.
**Composizione:**
Contiene le componenti *AllocationRequest* e *ErrorModal* per la visualizzazione di eventuali errori.
**Descrizione e funzionamento:**
È responsabile della gestione del posizionamento dei prodotti nelle scaffalature, sia per quanto riguarda l'allocazione sia per la richiesta di movimentazione. Nello specifico, se riceve indicazione da *WmsPage* ed è presente un prodotto selezionato (*selectedProduct*) o un bin selezionato (*selectedBin*) apre *AllocationRequest*. *AllocationRequest* sarà dunque dedicato all'allocazione se a essere selezionato è il prodotto in sé, mentre sarà dedicato alla movimentazione se a essere selezionato è il bin. Si occupa poi di aggiornare lo stato con i dati ricevuti da *AllocationRequest*. In caso di errori durante tale aggiornamento, *AllocationManager* apre anche una componente *ErrorModal*.

#### AllocationRequest

**Tipo:** Presentational
**Interazione con l'utente:**
L'utente seleziona la posizione (scaffalatura e ripiano+colonna della scaffalatura stessa) in cui inserire il prodotto.
**Composizione:**
È costituito da un componente *Drawer* di Ant Design, contenente a sua volta una componente *Form*, Quest'ultimo contiene i *Form.item* per la scaffalatura di destinazione (selezionabile tramite *Select*), per il ripiano della scaffalatura di destinazione (selezionabile tramite *Select*) e per la colonna della scaffalatura di destinazione (selezionabile tramite *Select*). Sono presenti anche due pulsanti per chiudere il *Drawer* annullando le modifiche oppure per confermare il posizionamento.
**Descrizione e funzionamento:**
Viene utilizzato per allocare un prodotto nel magazzino o per richiedere la movimentazione di uno già allocato. In questo secondo caso, il *Form* fornito contiene anche l'indicazione del bin di origine (*selectedBin*). L'utente compila/modifica il modulo seguendo le indicazioni date. Al momento poi dell'invio del modulo (se compilato correttamente e completamente), viene richiamata una funzione di *AllocationManager* con i dati caricati. Quest'ultima controlla attraverso un hook di React, precedentemente impostato, se i dati inviati riguardano un prodotto già allocato e procede a richiamare l'action dello store corrispondente (*orderMovementFromSelected* o *insertProduct*).

#### MovementManager

**Tipo:** Container
**Accesso allo store:**
Accede allo store per avere lo stato aggiornato di tutte le scaffalature (*shelves*) e movimentazioni pendenti (*movements*), necessario per poter fornire tutti i dati a *MovementView*. Accede allo store anche per modificare lo stato in risposta alle azioni compiute dall'utente attraverso *MovementView* (attraverso *removeMovementsWithBin* e *updateBinState*).
**Composizione:**
È composto dalla componente *MovementView*.
**Descrizione e funzionamento:**
È responsabile della gestione delle richieste di movimento pendenti. Nello specifico, se riceve indicazione da *WmsPage* apre *MovementView*. Attraverso *MovementView* sarà possibile vedere le richieste pendenti e, per ciascuna, richiedere un sollecito. Il sollecito viene effettuato sotto forma di chiamata API ad un meccanismo esterno che ritorna "True" se la movimentazione è stata accettata o "False" se rifiutata. In base a questo valore *MovementManager* aggiornerà lo stato tramite *updateBinState* e *removeMovementsWithBin*.

#### MovementView

**Tipo:** Presentational
**Interazione con l'utente:**
L'utente può cliccare dei pulsanti di sollecito, presenti per ciascuna richiesta di movimentazione pendente.
**Composizione:**
È costituito da un componente *Drawer* di Ant Design, contenente a sua volta una componente *List*. Ogni *List.item* ha come descrizione scaffalatura e bin di origine e destinazione della movimentazione e come action un pulsante di sollecito della movimentazione.
**Descrizione e funzionamento:**
Viene utilizzato principalmente per visualizzare tutte le movimentazioni ancora pendenti e richiederne il disbrigo tramite apposito pulsante. Quest'ultima funzionalità è resa possibile tramite chiamata di una funzione di *MovementManager*.

#### Tools

**Tipo:** Presentational
**Interazione con l'utente:**
L'utente può cliccare dei pulsanti per aggiungere nuovi prodotti o scaffalature e visualizzare/sollecitare le richieste di movimentazione di prodotti.
**Composizione:**
È costituita principalmente da componenti *FloatButton* di Ant Design, accompagnate da *ToolTip* che ne chiariscono l'utilizzo. In particolare, include un singolo *FloatButton* impiegato per aprire la lista delle movimentazioni pendenti, e un *FloatButton.Group* che contiene i pulsanti *FloatButton* utilizzati per aggiungere elementi (scaffalature e prodotti) al magazzino.
**Descrizione e funzionamento:**
Funge da piattaforma centralizzata per l'accesso a varie funzionalità legate al magazzino, tra cui l'aggiunta di scaffalature e prodotti, nonché la gestione delle movimentazioni pendenti. Se cliccati i bottoni di aggiunta scaffalatura o prodotto vengono visualizzati i form di configurazione presenti in *ShelfManager* e *ProductManager*. Se invece viene cliccato il pulsante di apertura della lista dei movimenti viene visualizzato tramite *MovementManager* un elenco di tutte le richieste pendenti. L'apertura di tutte queste componenti è gestita attraverso il passaggio dello stato dei click di questi pulsanti a *WmsPage*. Come accennato, quest'ultima si occuperà poi di aprire le giuste componenti figlie tramite hooks di React.

#### ErrorModal

Si veda in 3.2.2.1 per le specifiche.

#### 3.2.2.4 Feature/render

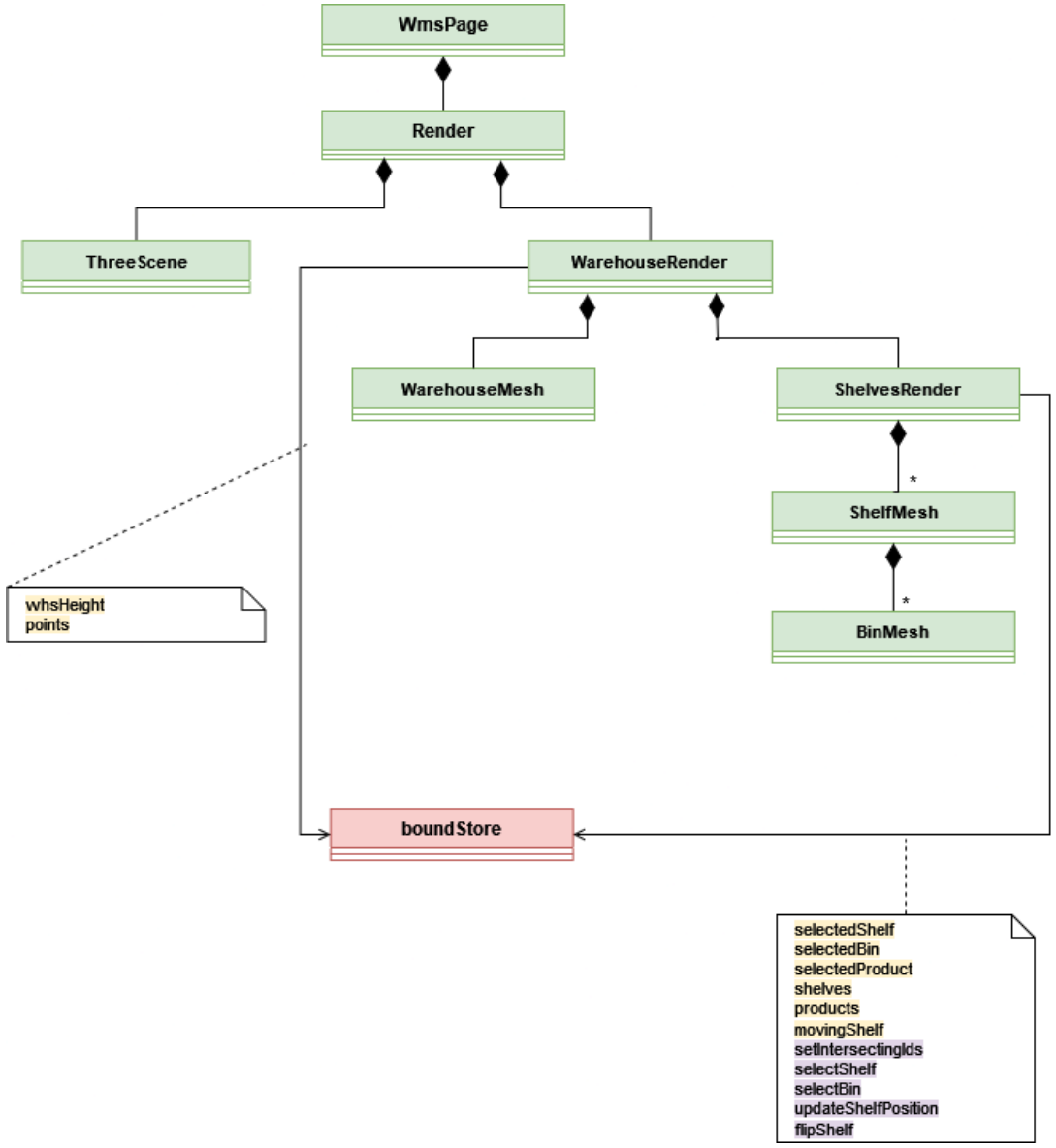



Figure 11: Diagramma UML feature/render

**WmsPage**

Si veda in 3.2.2.2 per le specifiche.

**Render**

**Tipo:** Presentational
**Interazione con l'utente:**
L'utente può abilitare il cambiamento della vista tramite click del tasto centrale del mouse.
**Composizione:**
È costituita da una componente *KeyboardControls* di react-three/drei e una *Canvas* di react-three/drei. Quest'ultima contiene, a sua volta, le componenti *ThreeScene* e *WarehouseRender*.
**Descrizione e funzionamento:**
Serve principalmente come contenitore della vista 3D del magazzino. Tuttavia, viene utilizzata anche per coordinare l'utilizzo della tastiera tra gli usi esterni al render 3D e quelli interni, i.e. di cambiamento della

vista. In particolare, viene utilizzato un hook di React, passato come prop a *ThreeScene*, per abilitare o meno questa proprietà.

#### ThreeScene

**Tipo:** Presentational
**Interazione con l'utente:**
L'utente può utilizzare il mouse o la tastiera per cambiare la vista del magazzino 3D.
**Composizione:**
È composta dalle componenti *CameraControls*, *ambientLight* e *gridHelper*.
**Descrizione e funzionamento:**
Serve a configurare la scena di Three.js. In particolare, aggiunge una luce e una griglia utile a dare delle coordinate all'utente per quanto riguarda la pavimentazione del magazzino. Sono presenti anche dei controlli per la camera per consentire il cambiamento della vista da tastiera e da mouse.

#### WarehouseRender

**Tipo:** Container
**Accesso allo store:**
Accede allo store per *whsHeight* e *points* per fornire i dati del magazzino a *WarehouseMesh*.
**Composizione:**
È composta dalle componenti *ShelvesRender* e *WarehouseMesh*.
**Descrizione e funzionamento:**
Serve da coordinatore tra le mesh delle scaffalature presenti nel magazzino (create da *ShelvesRender*) e la mesh del magazzino stesso (*WarehouseMesh*). In particolare, memorizza i riferimenti di queste mesh.

#### WarehouseMesh

**Tipo:** Presentational
**Interazione con l'utente:**
Nessuna.
**Composizione:**
Si tratta di una componente *group*, che raccoglie in sé delle componenti *Extrude* di react-three/drei per il pavimento e per ciascun muro.
**Descrizione e funzionamento:**
Serve alla costruzione della mesh del magazzino. Per ogni coppia di punti in *points*, ricevuti dalla componente *WarehouseRender*, viene creata una linea che viene estrusa con profondità *whsHeight*, misura anch'essa ricevuta come prop da *WarehouseRender*. Sempre attraverso *points*, viene creata una *Shape* di Three.js per il pavimento del magazzino.

#### ShelvesRender

**Tipo:** Container
**Accesso allo store:**
Accede allo store per avere lo stato aggiornato di tutte le scaffalature (*shelves*), necessario per poter visualizzare in 3D le scaffalature aggiornate all'ultima modifica. Rimane in ascolto anche per *selectedProduct*, *selectedBin* e *selectedShelf* per poter evidenziare nel render l'elemento selezionato. Nel caso la selezione avvenga da render, si utilizza *selectBin* e *selectShelf* per aggiornare lo stato. Utilizza inoltre *movingShelf* se è presenta una scaffalatura in movimentazione. In quest'ultimo caso, inoltre, accede allo store anche per modificare lo stato in risposta alla movimentazione (attraverso *updateShelfPosition*, *flipShelf* e *setIntersectingIds*).
**Composizione:**
Contiene una *ShelfMesh* per ogni scaffalatura presente in *shelves*. Contiene inoltre delle componenti *TransformControls* di react-three/drei attivati per la movimentazione e rotazione delle scaffalature in movimento (*movingShelf*).
**Descrizione e funzionamento:**
Si occupa della gestione delle mesh delle scaffalature. In particolare, si occupa di aggiornare lo store

quando scaffalature o bin vengono selezionati dal render (attraverso rispettivamente *selectShelf* e *selectBin*). Si occupa anche di controllare le collisioni tra scaffalature e mesh del magazzino e, di conseguenza, aggiornare la posizione nello store (attraverso *updateShelfPosition*). Inoltre, si occupa anche della movimentazione stessa delle scaffalature in movimento (*movingShelf*) attraverso i *TransformControls*.

**ShelfMesh**

**Tipo:** Presentational
**Interazione con l'utente:**
Nessuna.
**Composizione:**
Si tratta di un *group*, che raccoglie in sé componenti *BinMesh* per ogni bin della scaffalatura.
**Descrizione e funzionamento:**
Serve alla costruzione della mesh di una scaffalatura. Vengono create delle *BinMesh* per ogni bin della scaffalatura passata come prop. Queste vengono poi posizionati in modo tale da avere una matrice di mesh [*height* della scaffalatura][*width* della scaffalatura] con posizione centrale dell'intera matrice uguale alla posizione della scaffalatura passata come prop.

**BinMesh**

**Tipo:** Presentational
**Interazione con l'utente:**
L'utente può selezionare la scaffalatura cliccando sulla *BinMesh*. Se la scaffalatura è già selezionata, si seleziona il bin invece. Se invece è il bin a essere già selezionato, si seleziona la scaffalatura.
**Composizione:**
È composta da una *mesh* con geometria *boxGeometry* rappresentante il bin e un'altra interna più piccola rappresentare il prodotto contenuto, se esistente. È presente anche una componente *Edges* di react-three/drei utilizzata in caso di selezione di un bin o di un prodotto.
**Descrizione e funzionamento:**
Costruisce fisicamente in 3D il bin e l'eventuale prodotto contenuto. Il bin viene rappresentato come un cubo di dimensione fornita da *ShelfMesh* (dato dalla *binSize* memorizzata nello store per tale scaffalatura). Sono poi applicati diversi materiali ai lati di questo cubo per farlo somigliare ad una scaffalatura. Viene anche utilizzata una texture per segnalare colonna e ripiano dei bin. Per quanto riguarda il prodotto, anche questo viene rappresentato come un cubo del colore fornito da *ShelfMesh* (dato dal colore memorizzato nello store per tale prodotto). In caso di stati particolari del bin, vengono applicate delle texture al prodotto per segnalare ciò. Sia per prodotto che per bin, infine, in caso di selezione viene modificato il colore del bin e/o utilizzata una componente *Edges* per il prodotto.

### 3.2.3 API

Questa sezione fornisce una panoramica delle varie API implementate nel progetto, delineandone lo scopo, le funzionalità e l'utilizzo. Si precisa l'utilizzo dell'**API routes** di Next.js per l'implementazione di tali API. Questa soluzione permette di instradare e gestire le richieste dei client in modo efficiente, consentendo un'interazione fluida e affidabile con il server, oltre che facilitare lo sviluppo, il debug e il testing.

#### 3.2.3.1 jsonParser

**Scopo:**
Si occupa della validazione iniziale dei dati JSON. In particolare, verifica che i dati JSON ricevuti siano conformi alla struttura e al formato richiesti dall'applicazione.

**Funzionalità:**
L'API richiama una funzione *isValidJson(jsonData)* che prende un oggetto JSON come input ed esegue vari controlli per validarne la struttura. Verifica la presenza di chiavi essenziali come *products*, *shelves*, *whsName*, *whsHeight*, *whsPoints* e *movements*. Inoltre, garantisce che queste chiavi contengano i tipi e i formati di dati previsti:

- Per *products*, ogni prodotto nei dati JSON deve avere una proprietà *id* (tipo stringa), *name* (tipo stringa) e *color* (tipo oggetto).
- Per *shelves*: ogni scaffalatura deve avere una proprietà *id* (tipo stringa), *name* (tipo stringa), *binSize* (tipo numero), *width* (tipo numero), *height* (tipo numero), *isFlipped* (tipo boolean) e *bins* (tipo array di oggetti bin).
- Per *bins*: ogni oggetto bin deve avere una proprietà *id* (tipo stringa), *state* (tipo stringa) e una proprietà facoltativa *productId* (tipo stringa oppure null).

La funzione restituisce "True" se i dati JSON sono validi secondo i criteri sopra specificati; in caso contrario, restituisce "False".
Dopo aver ricevuto indicazione da questa funzione, l'API genera dunque risposte JSON contenenti messaggi di stato e di errore in base al risultato ottenuto.

**Utilizzo nel progetto:**
Questa API viene utilizzata da *WhsSetupFromJson* per gestire richieste POST contenenti file JSON. Dopo la ricezione dei dati, l'API li sottopone a una serie di controlli per convalidarli prima che siano elaborati ulteriormente. In particolare, questo test serve ad assicurarsi che il formato e struttura del file sia quello generato dal salvataggio del magazzino tramite l'applicazione. Se la convalida dei dati fallisce, al client viene restituita una risposta di errore appropriata. Si noti che non si controlla la validità dei dati stessi, che sarà invece controllata al momento dell'inserimento nello store. Tuttavia, questo approccio impedisce l'accesso a quest'ultimo per l'aggiornamento con dati sicuramente errati o incompleti.

#### 3.2.3.2 svgParser

**Scopo:**
Si occupa della validazione iniziale dei file SVG. Essa verifica la presenza di elementi specifici all'interno del file SVG per garantirne la conformità alle richieste dell'applicazione.

**Funzionalità:**
L'API richiama una funzione *isValidSVG(svgData)* che utilizza la libreria JSDOM per analizzare e manipolare il contenuto SVG ricevuto. In particolare, questa funzione crea un documento DOM virtuale, estrae gli elementi SVG pertinenti e applica le regole di validazione per determinare la conformità del file SVG. In particolare, gli elementi SVG che controlla sono:

- La presenza di elementi *polygon*;
- La presenza di almeno uno di questi con proprietà *points* definita.

Dunque, se almeno un poligono contiene punti, il file SVG viene considerato valido e viene restituito "True". Altrimenti, viene ritornato "False".
Dopo aver ricevuto indicazione da questa funzione, l'API genera dunque risposte JSON contenenti messaggi di stato e di errore in base al risultato ottenuto.

**Utilizzo nel progetto:**
Questa API viene utilizzata da *WhsSetupFromInput* per gestire richieste POST contenenti file SVG. Dopo la ricezione dei dati, l'API li sottopone a una serie di controlli per convalidarli prima che siano elaborati ulteriormente. In particolare, questo test serve ad assicurarsi che il formato e struttura del file renda possibile la creazione di un pavimento personalizzato a forma poligonale. Se la convalida dei dati fallisce, al client viene restituita una risposta di errore appropriata. Si noti che non si controlla la validità dei dati stessi, che sarà invece controllata al momento dell'inserimento nello store. Tuttavia, questo approccio impedisce l'accesso a quest'ultimo per l'aggiornamento con dati sicuramente errati o incompleti.

#### 3.2.3.3 movementRequest

**Scopo:**
Questa API è progettata per ricevere richieste relative ai movimenti all'interno dell'applicazione e determinare se devono essere accettate o rifiutate. Il suo scopo principale è emulare il processo di decisione automatizzato per le richieste di movimento.

**Funzionalità:**
L'API accetta richieste contenenti l'identificatore del movimento (*movementId*) e utilizza un algoritmo di simulazione per determinare lo stato della richiesta (accettata o rifiutata) in base a una probabilità casuale generata. L'API genera poi risposte JSON contenenti il messaggio relativo allo stato della richiesta.

**Utilizzo nel progetto:**
Questa API viene utilizzata da *movementManager* per il disbrigo delle richieste di movimento di prodotti all'interno del magazzino. Va notato che l'implementazione di questa API non è inclusa nel capitolato, bensì è sviluppata internamente dal proponente. Pertanto, questa API viene attualmente impiegata esclusivamente a fini di simulazione. Infatti, il suo utilizzo serve unicamente a fornire una rappresentazione pratica del processo decisionale automatico nel contesto operativo.

# 4 Architettura di deployment

Il gruppo ha deciso di adottare una architettura *Serverless* per il deployment. Essa consente agli sviluppatori di creare e gestire applicazioni senza doversi preoccupare dell'infrastruttura sottostante. Infatti, con "serverless" non si intende l'assenza totale di server, bensì l'assenza di questi e della loro gestione nell'esperienza di sviluppo. Infatti, tutte le attività di gestione dei server ricadono su un fornitore di servizi cloud.

In particolare, il gruppo ha deciso di utilizzare **Vercel** come piattaforma di hosting che supporta questo tipo di architettura. La scelta di questo strumento rispetto ad altri con le stesse funzionalità è dettata dalla sua alta integrazione con Next.js (utilizzato anch'esso nello sviluppo del progetto).

Un'architettura *Serverless* porta a vantaggi quali:

- **Automazione:** le soluzioni serverless eliminano infatti la fatica di gestire i server automatizzando le attività;
- **Scalabilità:** le soluzioni serverless aumentano e diminuiscono automaticamente in risposta al traffico senza la necessità di ottimizzazioni o altre configurazioni manuali;
- **Produttività:** le soluzioni serverless consentono agli sviluppatori di concentrarsi sulla scrittura del codice e sull'ottimizzazione dello stesso anziché dedicare tempo alla gestione dei server;
- **Semplicità di gestione:** e.g. con Vercel, è sufficiente collegare la cartella GitHub al servizio ed ogni volta che vengono apportate modifiche al repository GitHub, l'applicazione viene automaticamente rilasciata e aggiornata sulla piattaforma di hosting senza ulteriori interventi manuali.

# 5 Stato dei requisiti funzionali

Di seguito vengono riportati i requisiti funzionali corredati dal loro stato: "Soddisfatto" o "Non soddisfatto". Per una visione più completa sui requisiti si rimanda al documento *Analisi dei Requisiti v5.0.0*.

| Codice | Descrizione | Stato |
|---|---|---|
| ROF_1 | L'utente può creare un ambiente di magazzino tridimensionale | Soddisfatto |
| ROF_1.1 | L'utente può creare un ambiente di magazzino tridimensionale da zero | Soddisfatto |
| ROF_1.1.1 | L'utente può creare una planimetria personalizzata | Soddisfatto |

| Codice | Descrizione | Stato |
|---|---|---|
| RDF_1.2 | L'utente può caricare un layout memorizzato in database per inizializzare l'ambiente | Non soddisfatto |
| RFF_1.3 | L'utente può caricare un file in formato svg per inizializzare l'ambiente | Soddisfatto |
| RDF_2 | L'utente può salvare i dati del magazzino creato in un database | Non soddisfatto |
| RDF_2.1 | L'utente salva i dati dello spazio del magazzino | Soddisfatto |
| RDF_2.2 | L'utente salva i dati delle scaffalature presenti nel magazzino | Soddisfatto |
| RDF_2.3 | L'utente salva i dati dei prodotti presenti del magazzino | Soddisfatto |
| ROF_3 | L'utente può visualizzare tutto il magazzino in 3D | Soddisfatto |
| ROF_3.1 | L'utente può visualizzare lo spazio del magazzino in 3D | Soddisfatto |
| ROF_3.2 | L'utente può visualizzare le scaffalature posizionate all'interno del magazzino in 3D | Soddisfatto |
| ROF_3.3 | L'utente può visualizzare i prodotti posizionati all'interno del magazzino in 3D | Soddisfatto |
| RFF_3.3.1 | L'utente può visualizzare i prodotti creati (non posizionati) in 3D | Non soddisfatto |
| ROF_4 | L'utente può navigare attraverso lo spazio tridimensionale | Soddisfatto |
| ROF_4.1 | L'utente deve poter ingrandire l'area di visione a cui è interessato | Soddisfatto |
| ROF_4.2 | L'utente deve poter rimpicciolire l'area di visione a cui è interessato | Soddisfatto |
| ROF_4.3 | L'utente deve poter ruotare orizzontalmente la camera | Soddisfatto |
| ROF_4.4 | L'utente deve poter ruotare verticalmente la camera | Soddisfatto |

| Codice | Descrizione | Stato |
|---|---|---|
| ROF_4.5 | L'utente può navigare nello spazio tridimensionale attraverso il mouse | Soddisfatto |
| ROF_4.6 | L'utente può navigare nello spazio tridimensionale attraverso la tastiera | Soddisfatto |
| ROF_5 | L'utente può visualizzare in un'area gestionale separata (la libreria) l'elenco degli oggetti creati | Soddisfatto |
| ROF_5.1 | L'utente può visualizzare l'elenco delle scaffalature create | Soddisfatto |
| ROF_5.1.1 | L'utente per ogni scaffalatura deve poterne visualizzare il codice | Soddisfatto |
| ROF_5.1.2 | L'utente per ogni scaffalatura deve poterne visualizzare le dimensioni | Soddisfatto |
| ROF_5.1.3 | L'utente per ogni scaffalatura deve poterne visualizzare le dimensioni dei bin | Soddisfatto |
| ROF_5.2 | L'utente può visualizzare l'elenco dei prodotti creati | Soddisfatto |
| ROF_5.2.1 | L'utente per ogni prodotto deve poterne visualizzare il nome | Soddisfatto |
| ROF_5.2.2 | L'utente per ogni prodotto posizionato all'interno del magazzino deve poterne visualizzare la posizione | Soddisfatto |
| ROF_6 | L'utente può creare delle scaffalature | Soddisfatto |
| ROF_6.1 | L'utente può scegliere un codice univoco da dare alla scaffalatura | Soddisfatto |
| ROF_6.2 | L'utente può scegliere la dimensione delle scaffalature | Soddisfatto |
| ROF_6.2.1 | Le scaffalature devono essere divise in bin codificabili con coordinate | Soddisfatto |
| ROF_6.2.2 | L'utente può scegliere la dimensione del bin per la scaffalatura | Soddisfatto |
| ROF_7 | L'utente può inserire le scaffalature nello spazio 3D | Soddisfatto |
| ROF_8 | L'utente può selezionare una scaffalatura | Soddisfatto |

| Codice | Descrizione | Stato |
|---|---|---|
| ROF_8.1 | L'utente può selezionare una scaffalatura dalla libreria | Soddisfatto |
| RFF_8.1.1 | La scaffalatura è evidenziata in libreria quando viene selezionata | Soddisfatto |
| ROF_8.2 | L'utente può selezionare una scaffalatura dal render 3D | Soddisfatto |
| RFF_8.2.1 | La scaffalatura è evidenziata nel render 3D quando viene selezionata | Soddisfatto |
| ROF_9 | L'utente può modificare una scaffalatura creata | Soddisfatto |
| ROF_9.1 | L'utente può modificare la capacità della scaffalatura | Soddisfatto |
| RDF_9.2 | L'utente può modificare il codice della scaffalatura | Soddisfatto |
| RDF_9.3 | L'utente può modificare la posizione della scaffalatura | Soddisfatto |
| RDF_10 | L'utente può ricercare per codice una scaffalatura | Soddisfatto |
| ROF_11 | L'utente può eliminare una scaffalatura creata | Soddisfatto |
| ROF_11.1 | L'utente può cancellare la scaffalatura dalla libreria | Soddisfatto |
| ROF_11.2 | L'utente può cancellare la scaffalatura dal render 3D | Soddisfatto |
| ROF_12 | L'utente può creare un prodotto di forma parallelepipeda | Soddisfatto |
| ROF_12.1 | L'utente può scegliere un nome univoco da dare al prodotto | Soddisfatto |
| ROF_13 | L'utente può inserire i prodotti in un bin di una scaffalatura all'interno dello spazio 3D | Soddisfatto |
| ROF_14 | L'utente può selezionare un prodotto creato | Soddisfatto |
| ROF_14.1 | L'utente può selezionare un prodotto dalla libreria | Soddisfatto |
| RFF_14.1.1 | Il prodotto è evidenziato in libreria quando viene selezionato | Soddisfatto |
| ROF_14.2 | L'utente può selezionare un prodotto posizionato dal render 3D | Soddisfatto |

| Codice | Descrizione | Stato |
|---|---|---|
| RFF_14.2.1 | Il prodotto è evidenziato nel render 3D quando viene selezionato | Soddisfatto |
| RDF_15 | L'utente può ricercare per nome un prodotto | Soddisfatto |
| ROF_16 | L'utente può eliminare un prodotto creato | Soddisfatto |
| ROF_16.1 | L'utente può eliminare un prodotto creato dalla libreria | Soddisfatto |
| ROF_16.2 | L'utente può eliminare un prodotto posizionato dal render 3D | Soddisfatto |
| ROF_17 | L'utente può richiedere lo spostamento di un oggetto | Soddisfatto |
| RDF_17.1 | L'utente può richiedere lo spostamento tramite trascinamento | Non soddisfatto |
| ROF_17.2 | L'utente può richiedere lo spostamento tramite click del mouse | Soddisfatto |
| ROF_18 | Il sistema deve verificare la disponibilità della scaffalatura target alla richiesta di uno spostamento | Soddisfatto |
| ROF_19 | Il sistema delega ad un meccanismo terzo la decisione finale sull'accettazione di un movimento | Soddisfatto |
| RFF_20 | L'utente può codificare il magazzino in aree specifiche per uno scopo | Non soddisfatto |

Table 6: Stato dei requisiti funzionali

## 5.1 Grafici riassuntivi di copertura

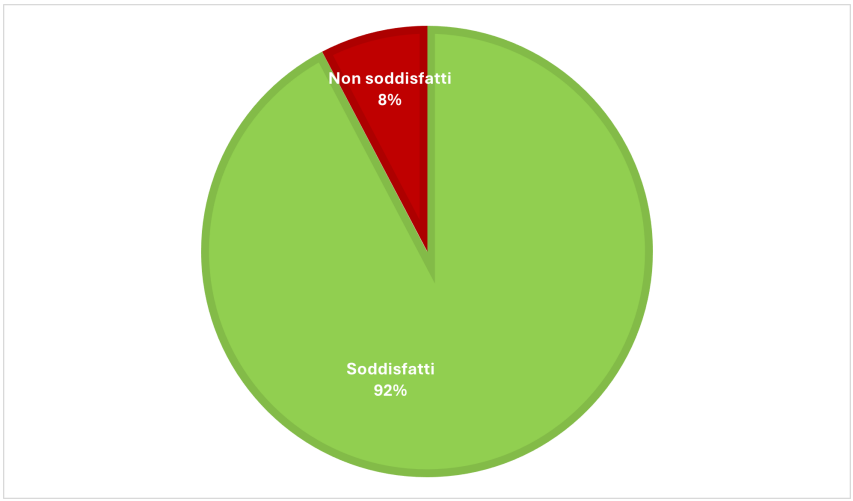



Figure 12: Grafico dello stato di copertura dei requisiti totali

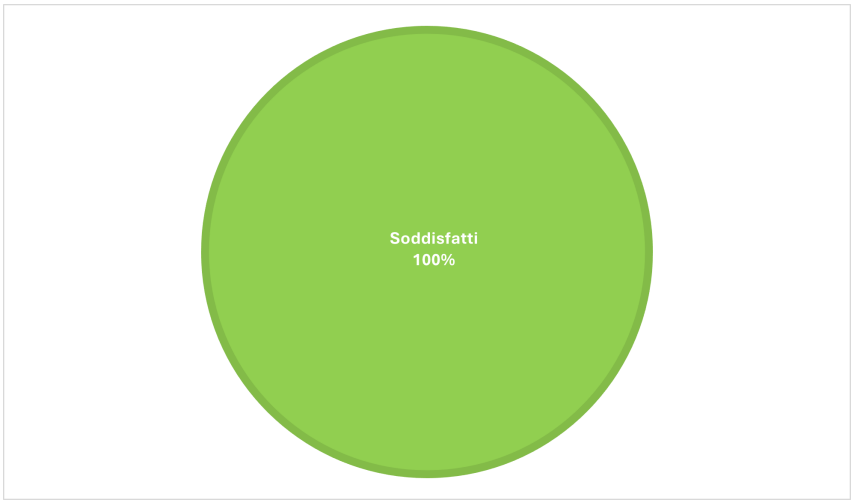



Figure 13: Grafico dello stato di copertura dei requisiti funzionali obbligatori

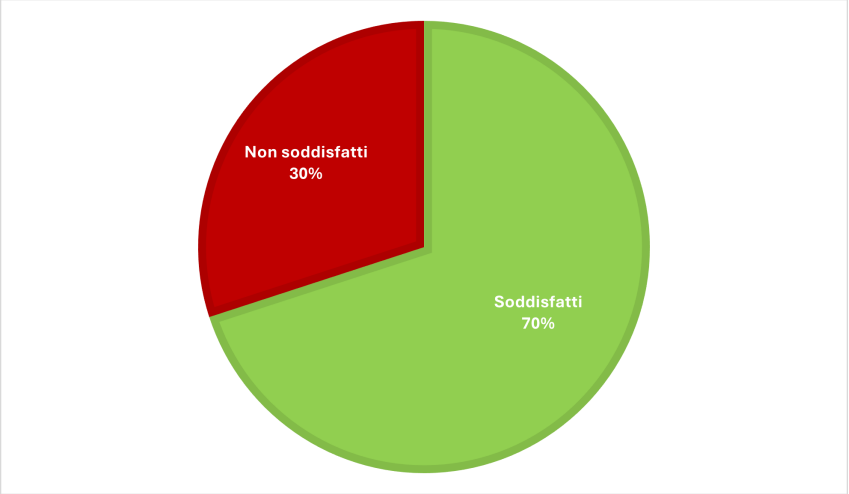



Figure 14: Grafico dello stato di copertura dei requisiti funzionali desiderabili

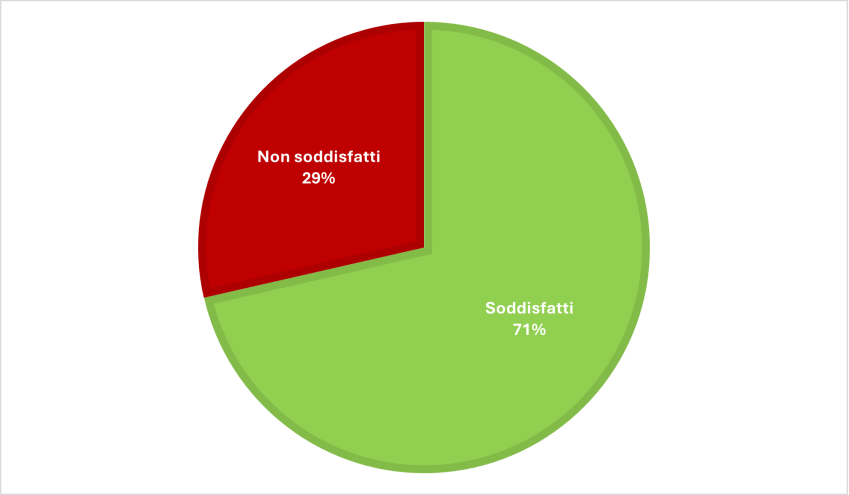



Figure 15: Grafico dello stato di copertura dei requisiti funzionali facoltativi

# 6 Riferimenti esterni

Per ulteriori chiarimenti sugli argomenti discussi nel documento, si possono consultare i seguenti link esterni:

- **Glossario v1.0.0**:
  `https://github.com/Avant-Garde-Software-Engineering/WMS3D/blob/main/Documentazione/PB/Esterna/glossario_v1.0.0.pdf`
- **Analisi dei requisiti v5.0.0**:
  `https://github.com/Avant-Garde-Software-Engineering/WMS3D/blob/main/Documentazione/PB/Esterna/analisi_dei_requisiti_v5.0.0.pdf`
- Capitolato **Warehouse Management 3D**:
  `https://www.math.unipd.it/~tullio/IS-1/2023/Progetto/C5.pdf`
- Link alla **documentazione del gruppo**:
  `https://avant-garde-software-engineering.github.io/documentazione.html` *(ultimo accesso 05-05-24)*

Per riferimenti all'architettura e ai pattern adottati si possono consultare i link:

- Link alla documentazione Zustand sulla **Flux inspired practice**:
  `https://docs.pmnd.rs/zustand/guides/flux-inspired-practice` *(ultimo accesso 05-05-24)*
- Link alla documentazione Zustand sullo **Slices pattern**:
  `https://docs.pmnd.rs/zustand/guides/slices-pattern` *(ultimo accesso 05-05-24)*
- Link alla documentazione Zustand su **Immutable state**:
  `https://docs.pmnd.rs/zustand/guides/immutable-state-and-merging` *(ultimo accesso 05-05-24)*
- Documentazione sull'**architettura a componenti** di React:
  `https://handsonreact.com/docs/component-architecture` *(ultimo accesso 05-05-24)*
- Documentazione sul **pattern Container/Presentational**:
  `https://www.patterns.dev/react/presentational-container-pattern/` *(ultimo accesso 05-05-24)*