Premessa

Questa nota tecnica descrive come integrare la funzionalità di esportazione in Excel (SheetJS) nella tabella AgilaeTable mostrata nello screenshot allegato. L'obiettivo è consentire agli utenti di scaricare un report completo dei dati filtrati e ordinati secondo i criteri correnti della UI, mantenendo la coerenza con il design esistente (toolbar con icone Refresh e "Crea Documento").

1. Architettura attuale della tabella

• **Material React Table** (MRT) usato in modalità *manual* per filtro, paginazione e sorting.
• I dati vengono ottenuti tramite `searchEntity(pagination, sorting, columnFilters, globalFilter)` che chiama il backend.
• La toolbar superiore è renderizzata da `renderTopToolbarCustomActions` ed oggi contiene:
- `RefreshIcon` ⇒ refetch della query.
- `CreateButton` ⇒ apertura modale di creazione.
• Le righe azioni sono gestite da `renderRowActions` con vari permessi.
• Stato dei filtri / sorting / paginazione è salvato negli hook `columnFilters`, `globalFilter`, `sorting`, `pagination`.

2. Requisiti per l'export

1. **Rispettare i filtri correnti**: l'utente deve trovare in Excel esattamente il dataset che vede in tabella (non solo la pagina corrente).
2. **Formato unico**: file `.xlsx` generato client■side con SheetJS.
3. **UI coerente**: il comando di download deve stare vicino alle altre azioni di tabella.
4. **Gestione carica/errore**: utilizzare `useThemeContext` (`setLoading`, `setApiError`) per feedback.
5. **Scalabilità**: possibilità futura di aggiungere altri formati o diversi "tipi" di report dalla stessa modale.

3. Strategia dati: come recuperare *tutte* le righe filtrate

Con `manualPagination=true` MRT possiede **solo** la pagina visibile.
Soluzione consigliata:

```ts
async function fetchAllRows(filters, sorting) {
// pageSize molto grande o endpoint dedicato /export
const hugePagination = { pageIndex: 0, pageSize: 10000 };
return await searchEntity(hugePagination, sorting, filters, globalFilter);
}
```

• Vantaggi: logica lato client minimale, si ri■usa `searchEntity`.
• Alternative: endpoint REST `/export` che accetta gli stessi parametri e restituisce un blob precomposto ■ utile per dataset massivi.

4. Inserimento del bottone Download

```tsx
renderTopToolbarCustomActions: () => (
<div style={{ display:'flex', gap:'1rem' }}>
<RefreshIcon onClick={() => refetch()} />
{canCreate && <CreateButton ... />}
<ExportIcon onClick={() => setOpenExport(true)} />
</div>
)
```

• `ExportIcon` può essere `FileDownloadOutlined`.
• Manteniamo la coerenza allineando le icone a sinistra, con il testo solo dove già presente ("Crea Documento").
• Il FAB o menu hamburger NON sono necessari perché la toolbar è già visibile e scroll■sticky.

5. Flusso modale di export

```tsx
const [openExport, setOpenExport] = useState(false);

<Dialog open={openExport} onClose={()=>setOpenExport(false)}>
<DialogTitle>Scarica report</DialogTitle>
<List>
```

<ListItemButton onClick={handleExportExcel}>
Excel – dati filtrati
</ListItemButton>
</List>
</Dialog>
```
• In futuro basterà aggiungere altri `ListItemButton` per “Excel completo”, “Pivot”, ecc.
• Aggiungere backdrop `setLoading('export', true/false)` se `fetchAllRows` supera 200■300■ms.

## 6. Implementazione `handleExportExcel` (SheetJS)

```ts
import * as XLSX from 'xlsx';

async function handleExportExcel() {
try {
const key = setLoading('export', true);
const { data } = await fetchAllRows(columnFilters, sorting);

// 1. Filtra colonne visibili
const visibleCols = table.getAllLeafColumns().filter(c => c.getIsVisible());
const rows = data.map(r => {
const obj: any = {};
visibleCols.forEach(col => {
let val = (r as any)[col.id];
// Normalizza tipi complessi
if (val && typeof val === 'object') val = val.toString();
obj[col.columnDef.header as string] = val;
});
return obj;
});

// 2. Crea workbook
const ws = XLSX.utils.json_to_sheet(rows);
// Header bold + freeze
XLSX.utils.sheet_add_aoa(ws, [visibleCols.map(c=>c.columnDef.header)], { origin: 'A1' });
ws['!freeze'] = { rows:1, cols:0 };

const wb = XLSX.utils.book_new();
XLSX.utils.book_append_sheet(wb, ws, 'Report');

// 3. Salva
XLSX.writeFile(wb, `report_${new Date().toISOString().slice(0,10)}.xlsx`);
setLoading(key,false);
} catch(err) {
setApiError(err);
}
}
```
Note: `table.getAllLeafColumns()` appartiene all’istanza MRT già presente nello scope di `AgilaeTable`.

7. Trasformazione dati: casi particolari

• **`ExtraId`** ⇒ usare `.toId()`.
• **`FlowState` / `AgilaeEnum`** ⇒ esportare `button.label` o `label`.
• **Date** ⇒ formattare in stringa ISO locale (`localDateTime`).
• **Campi annidati** ⇒ appiattire con `key_subKey` o creare mappa dedicata.
• Con SheetJS lo styling avanzato richiede la versione Pro: se serve colore header usare ExcelJS in futuro.

8. Performance & limiti

Righe esportate	Strategie consigliate
```

< 20■000	Export client■side con SheetJS (≈ 200■KB bundle)
20■000 – 100■000	Valutare *streaming* ExcelJS oppure endpoint backend
> 100■000	Generazione server■side, invio link di download firmato

9. Integrazione con ThemeProvider

• **Loading**: `const key = setLoading('export',true)` ⇒ backdrop.
• **Toast info**: `setInfoText("Export completato (123 righe)")`.
• **Error**: `setApiError(err as ErrorResponse)` per mostrare dialog standard.
• Coerenza con gli altri componenti (`Kanban`, modali) garantita.

10. Checklist finale

✓ Bottone export aggiunto nel toolbar.
✓ Modale con elenco formati.
✓ Funzione `fetchAllRows` basata su `searchEntity`.
✓ Generazione `.xlsx` con SheetJS, header bold, freeze pane.
✓ Gestione loading / error centralizzata.
✓ Possibilità di estendere a ExcelJS o endpoint server senza refactor UI.

Riferimenti e risorse

• [SheetJS docs](https://docs.sheetjs.com/)
• [Material React Table – custom toolbar actions](https://www.material-react-table.com/docs/examples/custom_toolbar_actions)
• [ExcelJS per formattazioni avanzate](https://github.com/exceljs/exceljs)

Autore

Generato da ChatGPT – 26 giugno 2025