Timebars Ltd
Agile PM | Classic PM | PPM

To do: have Claude work on adding this: Custom Standard Holiday Exceptions (Spreadsheet AdminPanel B6:C6 to B100:C100)

Data Model and Hierarchy Guide

Table of Contents

---

1. Introduction

The platform is a multi-product scheduling application available in three variants — **Agilebars (AB)**, **Timebars (TB)**, and **Costbars (CB)**. All three products share the same underlying data model and a single client-side IndexedDB database. The differences between products are driven by different scheduling engins and configuration, not by separate codebases.

At the heart of the platform is a **five-level parent child data hierarchy**, where every item — from a top-level portfolio down to an individual resource allocation — is a record in the `tbTimebars` store with a `tbType` field that identifies its level:

Level	`tbType`	Typical Purpose
L1	`Portfolio` or `Program`	Top-level grouping of related projects
L2	`Project`	A discrete deliverable or initiative
L3	`Sub-Project` or `Work-Package`	A smaller project, work package, or component within a project
L4	`Task` or `Milestone`	Individual work items or schedule control points
L5	`Allocation`	A specific resource's assignment to a task

Important Note: Agilebars is a Two Tier Hierarchy, Projects and Tasks, it has its own scheduling engine that skips the other levels.

Parent–child relationships are established via the `tbSelfKey2` field: every child record stores the `tbID` of its parent. This single field is what the scheduling engine, rollup functions, and hierarchy rendering all rely on to traverse the tree — upward to aggregate totals and downward to render the timescale and Kanban views.

Work and cost values are always entered at the leaf level (L5 Allocations, or L4 Tasks when allocations are not used) and **rolled up automatically** through the hierarchy. The `tbWork`, `tbAWork`, and `tbWorkRemaining` fields represent forecast hours, actual hours, and remaining hours respectively; `tbCost`, `tbACost`, and `tbCostRemaining` follow the same pattern for costs. Parent-level values are always derived — never entered directly — ensuring consistency from allocation to portfolio.

Agilebars is a simplified variant that omits L1 (Portfolio/Program), L3 (Sub-Project), and L5 (Allocation). Its hierarchy is a flat two-level structure: **Project (L2) → Task or Story (L4)**. The Kanban board and sprint burndown operate against this flatter model, and the scheduling engine for Agilebars uses lane-based progress steps rather than date-drag positioning.

Timebars supports the full five-level hierarchy with a gantt-style timescale. Resource allocations at L5 drive the scheduling engine: dragging a bar on the timescale recalculates actual start, actual hours/cost, remaining hours/cost, and percent complete in real time, then triggers a rollup of both hours and costs up the hierarchy chain.

Costbars also supports the full five-level hierarchy with the same cost and hours rollup model, and extends it with portfolio-level pipeline features — strategic value scoring, and demand vs. capacity analysis — making project selection decisions based on cost, hours, strategice value the primary analytical output.

This guide documents how data flows through that hierarchy — from the scheduling engine calculations at L5 up through each rollup stage to the portfolio total — and how the two stores (`tbTimebars` for core fields, `tbMetaData` for extended fields) relate to each other across all levels.

---

2. Data Hierarchy

The **tbTimebars** store in IndexedDB holds rows that represent different levels of work items. Each row is keyed by `tbID` (unique ID) and references its parent with `tbSelfKey2`. The `tbType` field determines its role in the hierarchy.

Timebars and Costbars

The full hierarchy is:

Portfolio → Project → Sub-Project → Task → Allocation

Example:

- **Portfolio**: "Digital Workplace - Transformation"
- **Project**: "Synchronisation Client"
- **Sub-Project**: "Private Cloud Phase 1"
- **Task**: "Consume requirements"
- **Allocations**: "Alloc 1" and "Alloc 2"

Each row stores cost, work, dates, durations, and scheduling metadata. Higher-level rows (Sub-Projects, Projects, Portfolios) never hold directly entered cost or work — their totals are always **rolled up** from their children.

At the task level, the rollup engine checks whether a Task has child Allocations. If Allocations exist, the Task's cost and work totals are summed from them. If no Allocations exist, the Task itself is treated as the leaf node and its own stored values are used directly in the rollup. This means allocations are optional — tasks without them still participate correctly in the hierarchy rollup.

Agilebars

Agilebars uses a simplified two-level hierarchy:

Project → Task (or Story)

L1 (Portfolio/Program), L3 (Sub-Project), and L5 (Allocation) are not supported. Example:

- **Project**: "Sprint 4 - Mobile App"
- **Tasks / Stories**: "Design login screen", "Write unit tests", "Fix auth bug"

Tasks are always the leaf nodes in Agilebars. Cost and work values are entered directly on each Task, and the Project rolls them up. There is no allocation layer — resource assignment in Agilebars is handled through the Kanban board lanes and sprint membership rather than discrete allocation records.

Let me grab a few more details before writing.

Now I have everything I need. Here are sections 3, 4, 5, and 6:

---

3. Parent–Child Relationships & WBS Ordering

Every row in `tbTimebars` links to its parent through a single field: **`tbSelfKey2`**, which stores the `tbID` of the parent row. Root nodes (Portfolios with no parent) have `tbSelfKey2` set to null or empty. This one field is what every traversal — rollup aggregation, hierarchy rendering, connector drawing — depends on.

Breadcrumb labels (`tbL1` through `tbL5`) are derived display fields, not structural ones. They are cleared and regenerated each time `updateL1ToL5()` runs. Each row gets its ancestor names written into these fields by walking up the `tbSelfKey2` chain, so that any row can show its full path without a live tree traversal at render time:

Field	Contains
`tbL1`	Portfolio or Program name
`tbL2`	Project name
`tbL3`	Sub-Project name
`tbL4`	Task name
`tbL5`	Allocation name

WBS ordering (`tbHierarchyOrder`) is a dot-separated position string, zero-padded to two digits per level, always five segments deep regardless of the actual depth of the row. Examples:

- Portfolio: `01.00.00.00.00`
- Project: `01.01.00.00.00`
- Sub-Project: `01.01.02.00.00`
- Task: `01.01.02.03.00`
- Allocation: `01.01.02.03.01`

This field is used for display ordering and WBS-style report sorting. Like `tbL1–tbL5`, it is a computed field regenerated on demand by `updateTbHierarchyOrder()` — it is not a source of truth for structure, only for ordering. This pre-processed data make managing project data in the spreadsheets and the reports easier for the users.

---

4. Rollup Rules & Field Aggregation

Costs and work flow upward through the hierarchy via the rollup engine (`tbschedulingengineRollups.js`). It builds an in-memory map of the entire hierarchy keyed by `tbID`, links children to parents via `tbSelfKey2`, then walks the tree bottom-up with `aggregateCostsAndWork()`.

The engine applies one rule at every node, regardless of type or level:

- **If a node has children**: its cost and work fields are overwritten with the sum of its children's values.
- **If a node has no children** (leaf node): its own stored values (`originalValues`) are written back as-is and passed upward unchanged.

The leaf-node rule is **not type-aware**. It does not distinguish between an Allocation, a Task, a Sub-Project, or a Project. Any node without children becomes the leaf at that branch and its own stored cost and work values participate in the rollup. This means it is valid to mix approaches within a single portfolio — some projects fully broken down to allocations, others with a cost and work figure entered directly at the project or sub-project level — and the rollup will aggregate all of them correctly into the portfolio total.

The general convention is that cost and work are entered at the Allocation level and that higher-level rows hold only derived totals. But this is a convention, not an enforcement. The engine simply sums whatever exists beneath each node and falls back to the node's own values when nothing is beneath it.

Running the rollup multiple times produces the same result — it is fully idempotent.

Fields that roll up at every level:

Field	Meaning
`tbWork`	Forecast hours
`tbAWork`	Actual hours to date
`tbWorkRemaining`	Remaining hours
`tbCost`	Forecast cost

Field	Meaning
`tbACost`	Actual cost to date
`tbCostRemaining`	Remaining cost

The full bottom-up flow:

```
Allocation (or any childless node)
  → summed into Task (or nearest parent with children)
    → summed into Sub-Project
      → summed into Project
        → summed into Portfolio
```

The Portfolio row is always the root node (`tbSelfKey2` = null). Its totals equal the sum of every leaf node beneath it, wherever in the tree those leaves sit.

5. Scheduling Engine & Scenarios

The scheduling engine (`tbschedulingengine2.js`) is triggered when a user drags or resizes an allocation bar on the timescale. It only recalculates **allocation-level fields** — tasks, sub-projects, projects, and portfolios are updated afterwards by the rollup engine, not by the scheduling engine directly.

> **Important — Manual Cost & Work at L1–L4:** The scheduling engine only recalculates cost and work fields for **Allocation (L5)** rows. If cost and work have been entered manually on a Task, Sub-Project, Project, or Portfolio row (i.e. the row has no children and is acting as a leaf node), those values will **not** be recalculated when the bar is dragged or resized on the timescale. The engine will still update the bar's dates and duration, but `tbWork`, `tbCost`, and all related actual and remaining fields will remain unchanged. To have cost and work recalculate automatically on drag, the values must be held at the Allocation level.

Input: Pixels to Days

The entry point is `prepForDateHoursCostCalculator()`, which converts the pixel offset of the drag (`leftDiff` for a move, `resizeDelta` for a resize) into days using the current timescale pixel factor (`tsPxFactor`, set per the weekly or monthly view). This produces `daysdiffS` (start shift) and `daysdiffF` (finish shift), which are passed to `dateHoursCostCalculator()`.

Scenario Detection

The engine reads the allocation's `tbStart` and `tbFinish` dates, converts them to JavaScript Date objects, then compares against `apStatusDate` (the user's status/report date) to determine which of three scenarios applies:

Condition	Scenario
`dStart < statusDate` AND `dFinish > statusDate`	**In Progress**

Condition	Scenario
`dStart < statusDate` AND `dFinish < statusDate`	**Completed**
`dStart >= statusDate`	**Not Started**

Scenario 1 — In Progress

The allocation has started but not yet finished. The actual start is locked to the bar's start date; actual finish remains empty.

Standard (no override):

```
tbDuration         = working days (dStart → dFinish)
tbADuration        = working days (dStart → statusDate)
tbRemainingDuration = tbDuration - tbADuration
tbWork             = tbCalendar × (tbPercentTimeOn / 100) × tbDuration
tbAWork            = tbCalendar × (tbPercentTimeOn / 100) × tbADuration
tbWorkRemaining    = tbCalendar × (tbPercentTimeOn / 100) ×
tbRemainingDuration
tbCost             = tbWork × tbPayRate
tbACost            = tbAWork × tbPayRate
tbCostRemaining    = tbWorkRemaining × tbPayRate
tbPercentComplete  = (tbWork - tbWorkRemaining) / tbWork × 100
tbAStart           = dStart (formatted DD-MMM-YYYY)
tbAFinish          = "" (not yet finished)
```

Override path (`tbSchEngOverride = "Overridden"`): The user has manually fixed `tbWorkRemaining`. The engine respects that stored value instead of recalculating it from duration:

```
tbAWork            = tbCalendar × (tbPercentTimeOn / 100) × tbADuration
tbWorkRemaining    = stored value (not recalculated)
tbWork             = tbWorkRemaining + tbAWork
tbCost/tbACost/tbCostRemaining recalculated from the above
tbPercentComplete  = (tbWork - tbAWork) / tbWork × 100
```

Scenario 2 — Completed

The entire bar sits before the status date. The allocation is 100% done.

```
tbAStart           = dStart (formatted DD-MMM-YYYY)
tbAFinish          = dFinish (formatted DD-MMM-YYYY)
tbDuration         = working days (dStart → dFinish)
tbADuration        = tbDuration
tbRemainingDuration = 0
tbWork             = tbCalendar × (tbPercentTimeOn / 100) × tbDuration
tbAWork            = tbWork
tbWorkRemaining    = 0
```

```
tbCost              = tbWork × tbPayRate
tbACost             = tbCost
tbCostRemaining     = 0
tbPercentComplete   = 100
```

Scenario 3 — Not Started

The bar's start date is on or after the status date. No actual progress exists.

```
tbAStart            = "" (not started)
tbAFinish           = "" (not started)
tbDuration          = working days (dStart → dFinish)
tbRemainingDuration = tbDuration
tbWork              = tbCalendar × (tbPercentTimeOn / 100) × tbDuration
tbAWork             = 0
tbWorkRemaining     = tbWork
tbCost              = tbWork × tbPayRate
tbACost             = 0
tbCostRemaining     = tbCost
tbPercentComplete   = 0
```

Key Allocation Fields Used by the Engine

Field	Role
`tbCalendar`	Work hours per day for this resource
`tbPercentTimeOn`	Percentage of time allocated to this task (0–100)
`tbPayRate`	Hourly cost rate — used to derive all cost fields from work hours
`tbSchEngOverride`	`"Overridden"` locks `tbWorkRemaining` against recalculation

After the scenario calculation completes, results are written back to IndexedDB via `initSchEngForm()` → `UpdateProgressHoursCalculator()`, and the rollup engine then propagates the updated totals up the hierarchy chain.

Good context from the code — the Kanban board even has an explicit toggle (`apABuseStatusDate`) letting Agilebars teams opt into using today's date instead, which makes the contrast between the two approaches even clearer. Here is the note:

> **Important — Progress is Calculated Against the Status Date, Not Today's Date:** The engine always compares bar positions against `apStatusDate` — the status or report date set by the user in the admin panel — not against the current calendar date. This is a deliberate project management discipline, not a technical limitation.
>
> In formal project management, schedules are updated on a defined reporting cycle: weekly, fortnightly, or monthly. The status date is the agreed "as of" date for that cycle — the point in time at which the team has collectively assessed progress and recorded actuals. Using today's date instead

> would mean that the same schedule opened on a Monday shows different percent complete, actual hours, and remaining work than it does on a Friday, even if no update has been made and no real work has been recorded. Reports sent to a steering committee or PMO would not match the numbers the PM sees in the tool, and trend analysis across reporting periods would become unreliable because the reference point keeps drifting.
>
> By anchoring all calculations to a fixed status date, the platform ensures that every report, chart, and rollup total reflects a consistent snapshot: what the project looked like as of that date. The PM advances the status date at the start of each new reporting period, updates the schedule, and the next snapshot is produced cleanly.
>
> Note: The Agilebars Kanban board offers an optional setting (`apABuseStatusDate`) that allows teams to use today's date instead of the status date, for teams that prefer to update progress continuously rather than on a fixed reporting cycle.

---

6. Predecessor & Constraint Logic

Predecessor (`tbPredecessor`)

A row can reference one other row as its predecessor by storing that row's `tbID` in the `tbPredecessor` field. This establishes a **Finish-to-Start** relationship in intent: the successor is expected to start when its predecessor finishes. The field stores the predecessor's `tbID` as a plain string and defaults to `""` when no predecessor exists.

Predecessors are supported on **Projects (L2)** and **Tasks, Milestones, and Gates (L4)**. Sub-Projects (L3), Allocations (L5), and Portfolios (L1) do not carry a predecessor reference in their rendered bar HTML and will not display connector lines even if a value exists in `tbPredecessor` on the underlying record.

Constraint Type (`tbConstraintType`)

The `tbConstraintType` field controls whether a bar's position is locked against movement caused by a predecessor cascade. The meaningful value is:

Value	Effect
`"Pinned"`	The bar is locked. A predecessor cascade will not move it. A pin icon is rendered on the bar.
`""` (empty)	No constraint. The bar moves freely and participates in predecessor cascades normally.

Pinning is useful when a task has a hard deadline or a fixed contractual date that must not drift when upstream bars are rescheduled.

Connector Rendering

When the timescale draws bars, each bar element receives `data-pred` and `data-tbid` HTML attributes. After rendering, `tbCanvas.js` iterates the bar elements, reads `data-pred` on each successor, looks up the corresponding predecessor element by `id="key{tbID}"`, and draws a connector line between the two. If the predecessor element is not found in the current view (filtered out or off-screen), the connector is silently skipped.

Important — Simplified Dependency Relationships (Scheduling Light)

Traditional project scheduling tools support four dependency relationship types: **Finish-to-Start (FS)**, **Start-to-Start (SS)**, **Finish-to-Finish (FF)**, and **Start-to-Finish (SF)**, each optionally combined with a positive or negative lag value. This platform intentionally implements a simplified subset of that model.

Only **Finish-to-Start** relationships are supported, with the option to apply a **positive lag** (a gap between the predecessor finishing and the successor starting) or a **negative lag** (which effectively approximates a Start-to-Start with overlap, allowing the successor to begin before its predecessor finishes). Finish-to-Finish and Start-to-Finish relationships are not supported, as Finish-to-Finish is rarely practical in day-to-day scheduling and Start-to-Finish is almost never used in practice.

When a **Task** is dragged on the timescale, the engine automatically cascades the move to any successor Tasks linked to it via `tbPredecessor`, recursively cascading through the full chain. A successor Task that has been **Pinned** will not move regardless of what its predecessor does — the pin takes priority over the cascade. This cascade is **limited to Tasks only** — dragging a Project bar does not trigger a cascade to linked successor Projects. The predecessor connector line will display on Project bars as a visual reference, but Project-level predecessors must be manually rescheduled.

This approach is intentionally described as **Scheduling Light** — enough dependency structure to communicate sequence and intent clearly on the timescale and in reports, without requiring teams to manage the complexity of a fully automated predecessor network across all levels of the hierarchy.

---

7 Timebar and MetaData Field Reference

The following are the associated Fields for each of the stores mandated through out the applications.

{ "tbTimebars": [{ "tbID": "system generated", "tbSelfKey2": null, "tbName": "My New L2 Project", "tbType": "Project", "tbSubType": "", "tbCoordTop": "76", "tbCoordLeft": "366", "tbStart": "18-Oct-2025", "tbFinish": "15-Feb-2026", "tbAStart": "", "tbAFinish": "", "tbDuration": 85, "tbRemainingDuration": 85, "tbBudgetHours": 0, "tbWork": 0, "tbAWork": 0, "tbWorkRemaining": 0, "tbPercentComplete": 0, "tbExpHoursPerWeek": null, "tbBudgetCost": 0, "tbCost": 0, "tbCostRemaining": 0, "tbACost": 0, "tbResID": 0, "tbCostID": null, "tbCustomerID": "44", "tbOwner": "", "tbBarHeight": null, "tbBLID": "", "tbPredecessor": "", "tbConstraintType": "", "tbConstraintDate": "", "tbFloat": 0, "tbFreeFloat": 0, "tbL1": "system generated", "tbL2": "system generated", "tbL3": "system generated", "tbL4": "system generated", "tbL5": "system generated", "tbWbsDescription": null, "tbHierarchyOrder": "system generated", "canvasNo": 1, "kbCoordTop": 167, "kbCoordLeft": "0", "tbStatus": "Active", "tbState": "", "tbStep": null, "tbStepStatus": null, "tbWfReasonNote": null, "tbStage": null, "tbPriority": null, "tbPhase": null, "tbBarColor": null, "tbTextColor": null, "focdItemCoordLeft": null, "focdItemCoordTop": null,

```
}
```

], "tbMetaData": [{ "tbMDID": "2753951", "tbMDName": "Auto Created tbID:2753951, Matching tbName: My New L2 Project, Optional "tbMDDecisionStrategic": null, "tbMDDecisionNotes": null, "tbMDFinancialScore": null, "tbMDCustomerID": "44", "tbMDGate": null, "tbMDHealthOverall": null, "tbMDHealthScope": null, "tbMDHealthCost": null, "tbMDHealthIssues": null, "tbMDHealthRisk": null, "tbMDHealthSchedule": null,

"tbMDHealthHours": null, "tbMDInvestmentCategory": null, "tbMDInvestmentInitiative": null, "tbMDInvestmentObjective": null, "tbMDInvestmentStrategy": null, "tbMDROMEstimate": null, "tbMDPortfolio": null, "tbMDProgram": null, "tbMDProgActivityAlignment": null, "tbMDSize": null, "tbMDStageApprover": null, "tbMDWrittenBy": null, "tbMDBusinessAdvisor": null, "tbMDBusinessOwner": null, "tbMDOrgManager": null, "tbMDPriorityStrategic": null, "tbMDSponsoringDepartment": null, "tbMDPrimaryContact": null, "tbMDBenefitCostRatio": null, "tbMDContactNumber": null, "tbMDResponsibleTeam": null, "tbMDRiskVsSizeAndComplexity": null, "tbMDEcnomicValueAdded": null, "tbMDEstimationClass": null, "tbMDInternalRateOfReturn": null, "tbMDSprintName": null, "tbMDSunkCosts": null, "tbMDSyncNotes": null, "tbMDNotesProject": null, "tbMDContractNumber": null, "tbMDNetPresentValue": null, "tbMDOpportunityCost": null, "tbMDPaybackPeriod": null, "tbMDPrimaryLineOfBusiness": null, "tbMDBackgroundInfo": null, "tbMDCapabilitiesNeeded": null, "tbMDConsequence": null, "tbMDExpectedBenefits": null, "tbMDProblemOpportunity": null, "tbMDConstraintsAssumptions": null, "tbMDCostBenefitAnalysis": null, "tbMDSeniorLevelCommitment": null, "tbMDStakeholderDescription": null, "tbMDNotesWorkflow": null, "tbMDOther2": null, "tbMDOther3": null, "tbMDOther4": null, "tbMDOther5": null, "tbMDOther6": null, "tbMDOther7": null, "tbMDOther8": null, "tbMDOther9": null, "tbMDOther10": null, "tbMDOther11": null, "tbMDOther12": null, "tbMDAzureID": null, "tbMDProjectNumber": null, "tbMDNotes": null, "tbMDExtLink1": null, "tbMDExtSystemID1": null, "tbMDSortOrder": null, "tbMDtbLastModified": null, "tbMDPriority": null, "tbMDStatus": null, "tbMDState": null, "tbMDSeverity": null, "tbMDStage": null, "tbMDPhase": null, "tbMDCategory": null, "tbMDHealth": null, "tbMDExSponsor": null, "tbMDPM": null, "tbMDProjectType": null, "tbMDShowIn": null, "tbMDYesNoSelector": null, "tbMDProduct": null, "tbMDWBS": null, "tbMDWeighting": null, "canvasNo": 1, "tbMDContingency": null, "tbMDMitigationPlan": null, "tbMDMitigationStatus": null, "tbMDProbability": null, "tbMDImpact": null, "tbMDScore": null, "tbMDRiskResponseStrategy": null, "tbMDContingencyPlan": null, "tbMDEscalationLevel": null, "tbMDTriggerEvent": null, "tbMDEarlyWarningIndicators": null, "tbMDCostbarsScore": 0, "tbMDObjectivesAndScope": null, "tbMDOptionsAnalysis": null, "tbMDImplementationApproach": null, "tbMDNextSteps": null, "tbMDVersion": null, "tbMDBCaseDateApproved": null, "tbMDPrerequisitesChecklist": null, "tbMDScheduleCommentary": null, "tbMDBudgetCommentary": null, "tbMDScopeCommentary": null, "tbMDResourceCommentary": null, "tbMDFinancialCommentary": null, "tbMDKeyDependencies": null, "tbMDValueProposition": null, "tbMDSuccessCriteria": null, "tbMDDecisionsRequired": null, "tbMDKeyRecommendations": null, "tbMDMarketAnalysis": null, "tbMDPortfolioImpactAnalysis": null, "tbMDKeyProjectMetrics": null, "tbMDRiskCategory": null, "tbMDIssueCategory": null, "tbPASTeamSize": null, "tbPASStakeholderCount": null, "tbPASTechnologyNovelty": null, "tbPASTeamTechExperience": null, "tbPASProjectSimilarity": null, "tbPASDomainExperience": null, "tbPASExternalIntegrations": null, "tbPASVendorDependencies": null, "tbPASCrossTeamCollaboration": null, "tbPASRegulatoryApprovals": null, "tbPASMarketTiming": null, "tbPASProjectType": null, "tbPASOverallFeasibilityScore": null, "tbPASPoliticalRiskScore": null, "tbPASComplexityScore": null, "tbPASRiskScore": null, "tbPASScaleScore": null, "tbPASTechnologyScore": null, "tbPASFamiliarityScore": null, "tbPASDependencyScore": null, "tbPASRegulatoryScore": null, "tbPASMarketScore": null, "tbPASAssessmentVersion": null, "tbPASStatus": null, "tbPASApprovalRequired": null, "tbPASAssessmentDate": null, "tbPASAssessorName": null }] }

---

tbBaseline Field Reference

The `tbBaseline` store is a point-in-time snapshot and mirrors the `tbTimebars` field structure exactly. Refer to the `tbTimebars` JSON above for all field names and types. The baseline is typically set from within the application (Right-click Canvas > Set Baseline) after the initial schedule is approved. For migration

purposes, if the source system carries an approved baseline plan, the same fields used to populate `tbTimebars` can be used to populate `tbBaseline` — the two structures are identical.

---

tbResources Field Reference

The `tbResources` store holds the resource pool. Each row represents one named person or generic role. L5 Allocation rows in `tbTimebars` link to a resource via the `tbResID` field.


```
{
  "tbResources": [
    {
      "tbResID": "700",
      "tbResResourceCalendar": "8",
      "tbResQuantity": "1",
      "tbResPayRate": "50",
      "tbResName": "Joe Invent",
      "tbResPrimaryRole": "R&D",
      "tbResNameShort": "Joe I",
      "tbResTeam": "Product & Innovation",
      "tbResPrimarySkill": "Testing",
      "tbResLabourType": "Human",
      "tbResPercentGeneralAvailability": "50",
      "tbResDaysNotAvailableByMonth": "Jan25-2, Feb25-2",
      "tbResStart": "1/Sep/25",
      "tbResFinish": "30/Nov/25",
      "tbResMonth1": "1.00",
      "tbResMonth2": "1.00",
      "tbResMonth3": "1.00",
      "tbResMonth4": "0.00",
      "tbResMonth5": "0.00",
      "tbResMonth6": "0.00",
      "tbResMonth7": "0.00",
      "tbResMonth8": "0.00",
      "tbResMonth9": "0.00",
      "tbResMonth10": "0.00",
      "tbResMonth11": "0.00",
      "tbResMonth12": "0.00",
      "tbResMonth13": "0.00",
      "tbResMonth14": "0.00",
      "tbResMonth15": "0.00",
      "tbResMonth16": "0.00",
      "tbResMonth17": "0.00",
      "tbResMonth18": "0.00",
      "tbResMonth19": "0.00",
      "tbResMonth20": "0.00",
      "tbResMonth21": "0.00",
      "tbResMonth22": "0.00",
      "tbResMonth23": "0.00",
      "tbResMonth24": "0.00",
      "tbResEmail": "joe@rlan.ca",
      "tbResExtSystemResID": null,
```

```
        "tbResDepartment": "Mechanical",
        "tbResManager": "George Bush",
        "tbResLocation": "Ottawa",
        "tbResTeamLeader": "App Team",
        "tbResSupervisor": "Helen Hunt",
        "tbResPartTimeFullTime": "Full Time",
        "tbResResourceType": "Labour",
        "tbResResourceClass": "Permenant",
        "tbResCostCode": "A404",
        "tbResCustomerID": "44",
        "tbResSortOrder": "1",
        "tbResPIN": "111111",
        "tbResEmail2": null,
        "tbResFirstName": null,
        "tbResLastName": null,
        "tbResMiddleName": null,
        "tbResStreet": null,
        "tbResCity": null,
        "tbResStateProvince": null,
        "tbResZipPostalCode": null,
        "tbResPhoneNo": null,
        "tbResPhoneNo2": null,
        "tbResCellPhoneNo": null,
        "tbResCountry": null,
        "tbResCompany": null,
        "tbResBirthday": null,
        "tbResLanguage": null,
        "tbResSocialLinks": null,
        "tbResCourse": null,
        "tbResCourseFinishDate": null,
        "tbResPhoto": null,
        "tbResPOBoxNo": null,
        "tbResAptNo": null,
        "tbResPostalStation": null,
        "tbResOccupation": null,
        "tbResSignature": null,
        "tbResCertNo": null,
        "tbResStatus": null,
        "tbResState": null,
        "tbResStage": null,
        "tbResStep": null,
        "tbResStepNotes": null,
        "tbResOffice": null,
        "tbResTransDate": null,
        "tbResStartDate": null,
        "tbResTermDate": null,
        "tbResRemarks": null,
        "id": 6486
      }
    ]
  }
```

tbTags Field Reference

The `tbTags` store defines all picklist dropdown values used throughout the metadata forms. Tags are grouped by `tbTagGroup`, which maps to a specific metadata field via `tbTagTbInternalName`.


```
{
  "tbTags": [
    {
      "tbTagID": "1",
      "tbTagGroup": "Benefit Cost Ratio",
      "tbTagName": "> 5:1",
      "tbTagSortOrder": null,
      "tbTagOpFieldName": null,
      "tbTagOpInternalName": null,
      "tbTagTbInternalName": "tbMDBenefitCostRatio",
      "tbTagOpTagName": null,
      "tbTagOpCustomOptionID": null,
      "tbTagEntity": null,
      "tbTagPopular": null,
      "tbTagNameShort": "> 5:1",
      "tbTagPurpose": "Tagging financial values on business case.",
      "tbTagDescription": null,
      "tbTagOwner": null,
      "tbTagCustomerID": null,
      "tbTagLastModified": null
    }
  ]
}
```

End of Document