Sets and availabilty

"Database prompt"

The following data is to be used as input.

- Table *equipment* with columns *id* and *current_quantity*
  This data is *equipment* that is available for renting. Per equipment id (item) the *current_quantity* is the stock available.
- Table *equipmentsetscontent* with columns *id*, *quantity* and *parent_equipment*.
  This defines a relation between equipment. Some equipment is part of a parent equipment. Therefor the availablity of the parent depends on the quantity needed of the childs. The parent-child relation is an hierachical tree.
- Table plannedequipment with columns *id*, *quantity*, *planperiod_start* and *planperiod_end*.
  This defines the number of equipment items, rented and not availbale during the period planperiod_start unitl planperiod_end (bothe dates are inclusive).

Example

equipment

id	quantity
1	-
2	14
3	6
4	40

equipmentsetscontent

id	quantity	parent_equipment
3	2	1
4	4	1
2	6	1

Because item 1 consists of 2 x item3 + 4 x item 4 + 6 x item 2. Based on the quatity available (from the equipment table), we can build 3 sets with item 3, 10 sets with item 4, and 2 sets with item 2. The total sets (item with id 1) availbale for renting out will be 2.

plannedequipment

id	quantity	planperiode_start	planperiod_end
2	2	2026-05-01	2026-05-10
2	1	2026-05-09	2026-05-12

We have two rentals for item 2. Th efirst rental decreases the quantity from item 2 from 14 to 12. Still 2 sets with id can be construncted so the avialability of the set wiht item id 1 will not be affected and stay 2.

From the 9th till the 12th we have one additional rental. This results in an decrease of stock of 3 in the period from the 9th till the 10th. When 3 items with id 2 are rented out, the remaining stock will become 11. This decreases the availabilyt of set (item) 1 to 1.

Output

I need to determine the availability of all ultimate parents per year and month.

I have a table *reservations* with the columns *equipment_id*, *current_quantity*, *year*, *month*, *availability*.

I want to upsert a row for any ultimate parent: the current_quatity is the default quantity when nothng is rentend out. And the availablity is an array of numbers. Each number represents the availablity of the set (ultimate parent) on that day.

[19, 19, 23, 21, 11, 10, 11, 11, 11, 13, 15, 15, 11, 11, 11, 11, 23, 23, 22, 8, 9, 9, 9, 13, 13, 13, 13, 23, 23, 11]

This means that on the first 2 days of the monthe we have 19 items (sets) and the 3rd 23, etc. etc.

What I need

Create a python script that creates the upserts for the table reservations for the next N months (default N = 12).

Create a method or more if needed in a class.

In order to acces the database you can use:
self.cursor.execute(...), self.conn.commit(), self.cursor.fetchall(), etc.

For logging use:

log(self, message: str, progress: bool = False, warn: bool = False) -> None:

Do you have enough infomration or do you need additional exmples or clarification?
Please ask clarification when needed, otherwise give me a strategy on how to proceed.

--

I have enough information to outline a strategy to proceed.

Summary of What We Need to Do

1. **Identify all ultimate parents** from the `equipmentsetscontent` table (i.e., parents that are not used as a child themselves).
2. **Build a dependency tree**: For each ultimate parent, find all the components and their required quantities (recursively).
3. **Track dynamic availability per day**:
   - Start with `current_quantity` from `equipment` for each item.
   - Apply reservations from `plannedequipment` to reduce availability on affected days.
4. **For each ultimate parent and for each day**, calculate:
   - The number of complete sets that can be built based on the most limiting child.
5. **Summarize per month**:
   - Create an array for each month with the daily availability (e.g., 30-element array for June).
6. **Upsert into the `reservations` table**:
   - If a row for the `(equipment_id, year, month)` exists, update it; otherwise, insert a new one.

Classes & Methods Proposal

We will define a class, e.g., `AvailabilityCalculator`, with the following responsibilities:

1. `get_ultimate_parents()`

Returns a list of ultimate parent equipment IDs.

2. `build_dependency_tree(parent_id)`

Returns a `Dict[item_id -> required_quantity]` for a given parent (resolved recursively).

3. `get_daily_availability_for_item(item_id, year, month)`

Returns a list of integers: available quantity per day of the given item after applying reservations.

4. `calculate_set_availability(dependency_dict, daily_availabilities)`

Takes the dependency definition and child item availabilities and calculates how many sets can be constructed per day.

5. `upsert_reservation(equipment_id, year, month, current_quantity, availability_array)`

Performs an insert or update into the `reservations` table.

6. `generate_reservations(months_ahead: int = 12)`

Main method to loop through upcoming months and ultimate parents, compute availability, and update the database.

⚠ Edge Cases to Handle

- Dates in `plannedequipment` are inclusive.
- Leap years and month-end dates.
- Recursive child resolution (tree structure).
- Cycles in the hierarchy (validate and skip with warning if detected).
- Gaps in days with no reservations should default to `current_quantity`.

□ Next Step

Would you like me to now:

1. Implement the skeleton for this class and the `generate_reservations()` method?
2. Or give you the full implementation?

If you're ready, I'll start with the full implementation in parts to keep it manageable and clear.

Code

Here is the **full Python implementation** of the `AvailabilityCalculator`

```
from datetime import datetime, timedelta
import calendar
import json
from collections import defaultdict

class AvailabilityCalculator:
    def __init__(self, conn):
        self.conn = conn
        self.cursor = conn.cursor()

    def log(self, message: str, progress: bool = False, warn: bool = False) -> None:
        prefix = "[WARN]" if warn else "[INFO]" if progress else "[LOG]"
        print(f"{prefix} {message}")

    def get_ultimate_parents(self):
        self.cursor.execute("SELECT DISTINCT parent_equipment FROM equipmentsetscontent")
        parents = set(row[0] for row in self.cursor.fetchall())

        self.cursor.execute("SELECT DISTINCT id FROM equipmentsetscontent")
        children = set(row[0] for row in self.cursor.fetchall())

        # Parents that are not children themselves
        ultimate_parents = parents - children
        return list(ultimate_parents)

    def build_dependency_tree(self, parent_id):
        stack = [(parent_id, 1)]
        dependencies = defaultdict(int)
        visited = set()

        while stack:
            current, factor = stack.pop()
            self.cursor.execute(
                "SELECT id, quantity FROM equipmentsetscontent WHERE parent_equipment = ?",
                (current,))
            rows = self.cursor.fetchall()
```

```
        if not rows:
            dependencies[current] += factor
        else:
            for child_id, qty in rows:
                if (current, child_id) in visited:
                    self.log(f"Cycle detected: {current} -> {child_id}", warn=True)
                    continue
                visited.add((current, child_id))
                stack.append((child_id, factor * qty))
    return dict(dependencies)

def get_reserved_quantities(self, item_id, year, month):
    _, days_in_month = calendar.monthrange(year, month)
    daily_reserved = [0] * days_in_month
    start_of_month = datetime(year, month, 1)
    end_of_month = start_of_month + timedelta(days=days_in_month - 1)

    self.cursor.execute("""
        SELECT quantity, planperiod_start, planperiod_end FROM plannedequipment
        WHERE id = ?
    """, (item_id,))
    for quantity, start_str, end_str in self.cursor.fetchall():
        start = datetime.strptime(start_str, "%Y-%m-%d")
        end = datetime.strptime(end_str, "%Y-%m-%d")
        for i in range(days_in_month):
            day = start_of_month + timedelta(days=i)
            if start <= day <= end:
                daily_reserved[i] += quantity
    return daily_reserved

def get_daily_availability_for_item(self, item_id, year, month):
    self.cursor.execute("SELECT current_quantity FROM equipment WHERE id = ?", (item_id,))
    result = self.cursor.fetchone()
    if not result:
        return []
    base_quantity = result[0]
    reserved = self.get_reserved_quantities(item_id, year, month)
    return [max(0, base_quantity - reserved[i]) for i in range(len(reserved))]
```

```
    def calculate_set_availability(self, dependency_dict, year, month):
        _, days_in_month = calendar.monthrange(year, month)
        availabilities = {}

        for item_id in dependency_dict:
            availabilities[item_id] = self.get_daily_availability_for_item(item_id, year,
month)

        result = []
        for day in range(days_in_month):
            set_count_per_day = [
                availabilities[item_id][day] // dependency_dict[item_id]
                for item_id in dependency_dict
            ]
            result.append(min(set_count_per_day) if set_count_per_day else 0)
        return result

    def upsert_reservation(self, equipment_id, year, month, current_quantity,
availability_array):
        self.cursor.execute("""
            SELECT COUNT(*) FROM reservations WHERE equipment_id = ? AND year = ? AND month =
?
        """, (equipment_id, year, month))
        exists = self.cursor.fetchone()[0] > 0

        availability_json = json.dumps(availability_array)

        if exists:
            self.cursor.execute("""
                UPDATE reservations
                SET current_quantity = ?, availability = ?
                WHERE equipment_id = ? AND year = ? AND month = ?
            """, (current_quantity, availability_json, equipment_id, year, month))
        else:
            self.cursor.execute("""
                INSERT INTO reservations (equipment_id, current_quantity, year, month,
availability)
                VALUES (?, ?, ?, ?, ?)
            """, (equipment_id, current_quantity, year, month, availability_json))
```

```
        self.conn.commit()

    def generate_reservations(self, months_ahead: int = 12):
        today = datetime.today().replace(day=1)

        self.cursor.execute("""
            CREATE TABLE IF NOT EXISTS reservations (
                equipment_id INT NOT NULL,
                current_quantity INT NOT NULL,
                year INT NOT NULL,
                month INT NOT NULL,
                availability JSON NOT NULL,
                PRIMARY KEY (equipment_id, year, month)
            )
        """)

        ultimate_parents = self.get_ultimate_parents()

        for month_offset in range(months_ahead + 1):
            date = today + timedelta(days=32 * month_offset)
            year, month = date.year, date.month

            for parent_id in ultimate_parents:
                dependencies = self.build_dependency_tree(parent_id)
                availability_array = self.calculate_set_availability(dependencies, year,
month)
                default_quantity = min(
                    self.get_daily_availability_for_item(parent_id, year, month) or [0]
                )
                self.upsert_reservation(parent_id, year, month, default_quantity,
availability_array)
                self.log(f"Upserted reservation for {parent_id} in {year}-{month:02}",
progress=True)

        self.log("□ All reservations generated.", progress=True)
```

POC PHP Implementation Sets Availablity

Step 1

Reformat *plannedequipment* **into** *reservations per date*

Processes equipment reservation data.

- Reads **reservation** records from JSON files in the "**plannedequipment**" directory.
- Reads **equipment properties** from JSON files in the "**equipment**" directory.

Calculates daily reservation totals for each equipment item.

- Covers a date range: current month plus a configurable number of months (default 9).
- Organizes data into a nested structure: year → month → day.

Computes daily available quantity: current quantity minus reservations.
Outputs results to JSON files in the "**reservationperdate**" directory.
Offers optional features:

- Suppression of days with zero reservations.
- Prevention of file creation when no reservation data exists.

Script

```
# createReservationperdatev2.py
```

```
import os
import json
from datetime import datetime, date, timedelta
import calendar

# Parameters
SUPPRESS_ZERO = True           # If True, skip storing day nodes with zero reservations.
SUPPRESS_EMPTY_FILE = True     # If True, do not create an output file if the "dates" part is
empty.
ADDITIONAL_MONTHS = 9          # Number of additional months to process (beyond the current
month)

# Define the directories
plannedequipment_dir = 'plannedequipment'
equipment_dir = 'equipment'
output_dir = 'reservationperdate'
os.makedirs(output_dir, exist_ok=True)

# Define the date range: from the first day of this month to the last day of the month
ADDITIONAL_MONTHS later.
today = date.today()
start_date = date(today.year, today.month, 1)
end_month = today.month + ADDITIONAL_MONTHS
end_year = today.year + (end_month - 1) // 12
end_month = ((end_month - 1) % 12) + 1
last_day = calendar.monthrange(end_year, end_month)[1]
end_date = date(end_year, end_month, last_day)

# Define the list of equipment property keys to copy from the equipment JSON files.
equipment_keys = ["is_combination", "is_physical", "current_quantity"]

def add_day(nested_dates, current_date):
    """Ensure that nested_dates[year][month][day] exists and initialize it."""
    year = str(current_date.year)
    month = f"{current_date.month:02d}"
    day = f"{current_date.day:02d}"

    if year not in nested_dates:
        nested_dates[year] = {}
```

```
    if month not in nested_dates[year]:
        nested_dates[year][month] = {}
    if day not in nested_dates[year][month]:
        nested_dates[year][month][day] = {"reservations": 0, "available": None}

def remove_empty_containers(nested_dates):
    """Remove empty month and year nodes if they have no day entries."""
    years_to_remove = []
    for year, months in nested_dates.items():
        months_to_remove = []
        for month, days in months.items():
            if not days:  # If month dictionary is empty
                months_to_remove.append(month)
        for month in months_to_remove:
            del nested_dates[year][month]
        if not nested_dates[year]:
            years_to_remove.append(year)
    for year in years_to_remove:
        del nested_dates[year]

# Process each JSON file in the plannedequipment directory.
for filename in os.listdir(plannedequipment_dir):
    if not filename.endswith('.json'):
        continue

    # Build full paths for plannedequipment, equipment and output files.
    plan_file_path = os.path.join(plannedequipment_dir, filename)
    equipment_file_path = os.path.join(equipment_dir, filename)
    output_path = os.path.join(output_dir, filename)

    # Load the reservation data from the plannedequipment file.
    with open(plan_file_path, 'r', encoding='utf-8') as infile:
        plan_data = json.load(infile)

    # Initialize the nested dates dictionary.
    nested_dates = {}
    current_date = start_date
    while current_date <= end_date:
        add_day(nested_dates, current_date)
```

```
        current_date += timedelta(days=1)

    # Process each reservation record.
    for record in plan_data.get('data', []):
        start_str = record.get('planperiod_start')
        end_str = record.get('planperiod_end')
        if not start_str or not end_str:
            continue

        record_start = datetime.fromisoformat(start_str).date()
        record_end = datetime.fromisoformat(end_str).date()
        # Clamp the reservation period to our target range.
        period_start = max(record_start, start_date)
        period_end = min(record_end, end_date)
        if period_start > period_end:
            continue

        quantity = record.get('quantity', 0)
        current = period_start
        while current <= period_end:
            year = str(current.year)
            month = f"{current.month:02d}"
            day = f"{current.day:02d}"
            if year in nested_dates and month in nested_dates[year] and day in
nested_dates[year][month]:
                nested_dates[year][month][day]["reservations"] += quantity
            current += timedelta(days=1)

    # Initialize equipment properties.
    item_properties = {}
    current_quantity = 0

    # Read the corresponding equipment file.
    if os.path.exists(equipment_file_path):
        with open(equipment_file_path, 'r', encoding='utf-8') as eq_file:
            equipment_data = json.load(eq_file)
        equipment_item = equipment_data.get('data', {})
        # Copy the desired properties.
        for key in equipment_keys:
```

```
            item_properties[key] = equipment_item.get(key, None)
        current_quantity = equipment_item.get('current_quantity', 0)
    else:
        print(f"Error: Equipment file '{filename}' not found in '{equipment_dir}'. Assuming
current_quantity = 0 and default item properties.")
        for key in equipment_keys:
            item_properties[key] = None

    # Update each day with the available quantity and optionally suppress days with zero
reservations.
    for year in list(nested_dates.keys()):
        for month in list(nested_dates[year].keys()):
            for day in list(nested_dates[year][month].keys()):
                day_data = nested_dates[year][month][day]
                reservations = day_data["reservations"]
                day_data["available"] = current_quantity - reservations
                if SUPPRESS_ZERO and reservations == 0:
                    del nested_dates[year][month][day]
            # If a month becomes empty after removing days, remove the month.
            if not nested_dates[year][month]:
                del nested_dates[year][month]
        # If a year becomes empty after removing months, remove the year.
        if not nested_dates[year]:
            del nested_dates[year]

    # If SUPPRESS_EMPTY_FILE is True and the nested dates dictionary is empty,
    # remove the output file if it exists and skip writing.
    if SUPPRESS_EMPTY_FILE and not nested_dates:
        if os.path.exists(output_path):
            os.remove(output_path)
            print(f"Removed existing file {output_path} because there are no reservations.")
        continue

    # Combine the item properties and the nested date reservation data.
    output_data = {
        "item": item_properties,
        "dates": nested_dates
    }
```

```
    # Write the result to the output directory.
    with open(output_path, 'w', encoding='utf-8') as outfile:
        json.dump(output_data, outfile, indent=2)

print("Reservation, availability, and item data have been processed and saved.")
```

Step 2

Download the set hiarchy from the API

(takes long, build in delays to prevent throttling)

```
# getSets.py

import json
import time
import requests
import sys

# Configuration variables (adjust these as needed)
BASE_URL = "https://api.rentman.net/"  # Replace with your actual base URL

FILTER_ON_TAGS = False  # Set to True if you want to apply tag filtering
API_TOKEN = "abc"  # Replace with your actual API token

def log_message(message, force=False):
    """Logs a message. You can modify this to log to a file if needed."""
    sys.stderr.write(message + "\n")
    if force:
        print(message)

def count_api_usage():
    """Stub function for counting API usage."""
    # Extend this function to track API usage if needed.
    pass

def filter_array_by_selections(items, selections):
    """
    Filters an array of dictionaries based on the presence of certain keys with truthy values.
```

```
    :param items: List of dictionaries to filter.
    :param selections: List of keys to check.
    :return: Filtered list of dictionaries.
    """
    filtered = []
    for item in items:
        for sel in selections:
            if item.get(sel):
                filtered.append(item)
                break
    return filtered

def make_request(method, endpoint, data=None):
    """
    Makes an HTTP request using the requests library.

    :param method: HTTP method ('GET' or 'POST').
    :param endpoint: API endpoint (appended to BASE_URL).
    :param data: Dictionary of data or query parameters.
    :return: Parsed JSON response or None if error occurs.
    """
    if data is None:
        data = {}

    url = BASE_URL + endpoint
    headers = {
        "Authorization": f"Bearer {API_TOKEN}",
        "Content-Type": "application/json"
    }

    try:
        if method.upper() == 'POST':
            response = requests.post(url, headers=headers, json=data)
        elif method.upper() == 'GET':
            response = requests.get(url, headers=headers, params=data)
        else:
            log_message(f"Unsupported HTTP method: {method}")
            return None
```

```
    except requests.RequestException as e:
        log_message(f"Request exception: {e}")
        return None

    if not response.ok:
        log_message(f"HTTP error: {response.status_code} | Response Body: {response.text}")
    try:
        return response.json()
    except json.JSONDecodeError:
        log_message("Error decoding JSON response.")
        return None

def get_request(endpoint, params=None):
    """
    Performs a GET request and counts API usage.

    :param endpoint: API endpoint.
    :param params: Query parameters as a dictionary.
    :return: Parsed JSON response.
    """
    count_api_usage()
    return make_request('GET', endpoint, params)

def equipmentsetscontent(filter_on_tags=FILTER_ON_TAGS):
    """
    Retrieves equipment sets content by fetching data in batches.

    :param filter_on_tags: If True, applies tag filtering.
    :return: Complete list of items.
    """
    all_items = []
    offset = 0
    batch_size = 300
    total_fetched = 0

    while True:
        params = {'offset': offset, 'limit': batch_size}
        response = get_request('equipmentsetscontent', params)
```

```
        if not response or 'data' not in response:
            print("<pre><h1>Error in API Request</h1>")
            print(response)
            sys.exit(1)

        items = response['data']
        all_items.extend(items)
        total_fetched += len(items)

        # If fewer than batch_size items are returned, no more records are available.
        if len(items) < batch_size:
            break

        offset += batch_size

        # If 3000 records have been fetched, pause for 1 second to avoid rate limits.
        if total_fetched >= 3000:
            log_message("Pausing for 1 second to avoid rate limits...", True)
            time.sleep(1)
            total_fetched = 0  # Reset the counter after the pause

    if filter_on_tags:
        all_items = filter_array_by_selections(all_items, ['hoofditem', 'set'])

    return all_items

# Main execution
if __name__ == "__main__":
    complete_data = equipmentsetscontent()
    print(json.dumps(complete_data, indent=4))
```

Step 3,

Convert the output from step 2 into sets-childs format (parrent with list of childs).

For each entry, the script removes the "/equipment/" prefix from both the parent and child equipment values using the helper function.

For every entry, after stripping the prefixes, a new dictionary is created for the child item (containing the stripped child equipment and its quantity). This child dictionary is then appended to the list associated with the corresponding parent key in the dictionary.

Once all entries are processed, the dictionary keys (i.e., the parent equipment numbers) are sorted in ascending order to organize the final output.

Prompt

Write a Python script that performs the following tasks:

- Define variables at the top of the script for an input file and an output file. The input file is "sets/sets.json" and the output file is "sets/sets-childs.json".
- Read the JSON data from the input file. The file contains a list of objects, each with fields such as "parent_equipment", "equipment", and "quantity". The "parent_equipment" and "equipment" fields are strings that start with the prefix "/equipment/".
- Create a helper function to remove the "/equipment/" prefix from these strings.
- Group the objects by the numeric part of the parent equipment (after stripping the prefix). For each group, store a list of child items, where each child is represented by a dictionary containing the stripped "equipment" number and its "quantity".
- Sort the groups by their parent equipment keys in ascending numerical order.
- Write the resulting grouped and sorted data to the output file in JSON format with proper indentation.
- Please generate the complete Python code to achieve this.

```
# set-combined.py

import json
from collections import defaultdict

# Define file paths
input_file = "sets/sets.json"
output_file = "sets/sets-childs.json"

def strip_prefix(equipment_str):
    prefix = "/equipment/"
    return equipment_str[len(prefix):] if equipment_str.startswith(prefix) else equipment_str

# Load the input JSON file
```

```
with open(input_file, "r") as infile:
    data = json.load(infile)

# Group child items by parent equipment (after stripping the prefix)
sets_dict = defaultdict(list)
for entry in data:
    parent = strip_prefix(entry["parent_equipment"])
    child_item = {
        "equipment": strip_prefix(entry["equipment"]),
        "quantity": entry["quantity"]
    }
    sets_dict[parent].append(child_item)

# Sort the dictionary by parent keys in ascending numerical order
sorted_sets_dict = {k: sets_dict[k] for k in sorted(sets_dict.keys(), key=lambda x: int(x))}

# Write the output JSON file with sorted parent keys
with open(output_file, "w") as outfile:
    json.dump(sorted_sets_dict, outfile, indent=4)

print("The new JSON file has been created successfully.")
```

Step 4

Calculate set availability into reservationperdate-sets folder.

Prompt: prompt to create set reservations.docx

```
#!/usr/bin/env python3
# setReservationperdate.py

import os
import json
import math
from collections import defaultdict

# Directories for input and output files
```

```
SETS_FILE = os.path.join("sets", "sets-childs.json")
RESERVATION_DIR = "reservationperdate"
OUTPUT_DIR = "reservationperdate-sets"

# Ensure output directory exists
os.makedirs(OUTPUT_DIR, exist_ok=True)

def load_json(filepath):
    with open(filepath, "r") as f:
        return json.load(f)

def get_child_reservation(child_id):
    """
    Given a child equipment id, load its reservation file.
    Assumes filename is child_id + ".json" in the RESERVATION_DIR.
    Returns a dict with keys "item" and optionally "dates".
    """
    filepath = os.path.join(RESERVATION_DIR, f"{child_id}.json")
    if os.path.exists(filepath):
        return load_json(filepath)
    else:
        raise FileNotFoundError(f"Reservation file for equipment {child_id} not found at
{filepath}")

def extract_date_availability(reservation):
    """
    Extract a mapping from (year, month, day) tuple to available quantity.
    If a reservation file has no "dates" node, returns an empty dict.
    """
    date_avail = {}
    dates = reservation.get("dates", {})
    for year, months in dates.items():
        for month, days in months.items():
            for day, data in days.items():
                date_avail[(year, month, day)] = data.get("available", 0)
    return date_avail

def nested_date_structure(date_dict):
    """
```

```
    Given a mapping with keys (year, month, day) and value (availability),
    build a nested dictionary: year -> month -> day -> {reservations: 0, available: value}
    """
    nested = defaultdict(lambda: defaultdict(dict))
    for (year, month, day), avail in date_dict.items():
        nested[year][month][day] = {"reservations": 0, "available": avail}
    return {year: dict(months) for year, months in nested.items()}

def main():
    # Load the sets definitions
    sets_def = load_json(SETS_FILE)

    # Process each set defined in sets-childs.json
    for set_id, children in sets_def.items():
        child_infos = []  # List of tuples: (child_id, required_quantity, full_stock,
date_availability mapping)
        for child in children:
            child_id = child["equipment"]
            req_qty = child["quantity"]
            try:
                reservation = get_child_reservation(child_id)
            except FileNotFoundError as e:
                print(e)
                continue  # Skip this child if its reservation file is missing
            full_stock = reservation.get("item", {}).get("current_quantity", 0)
            date_avail = extract_date_availability(reservation)
            child_infos.append((child_id, req_qty, full_stock, date_avail))

        # Calculate the full availability (without date restrictions)
        available_sets_full = math.inf
        for (_, req_qty, full_stock, _) in child_infos:
            child_full_sets = full_stock // req_qty
            available_sets_full = min(available_sets_full, child_full_sets)
        if available_sets_full == math.inf:
            available_sets_full = 0

        # Gather the union of all dates from the children
        union_dates = set()
        for (_, _, full_stock, date_avail) in child_infos:
```

```
            union_dates.update(date_avail.keys())

        # Compute availability per date from the union of dates
        date_results = {}
        for date in union_dates:
            avail_for_date = math.inf
            for (_, req_qty, full_stock, date_avail) in child_infos:
                # Use date-specific available if provided, otherwise use full stock
                child_avail = date_avail.get(date, full_stock)
                child_sets = child_avail // req_qty
                avail_for_date = min(avail_for_date, child_sets)
            date_results[date] = avail_for_date

        # Only include dates where the available set count is lower than full availability
        filtered_dates = {date: avail for date, avail in date_results.items() if avail <
available_sets_full}

        # Build output JSON structure following the reservation file structure
        output = {}
        output["item"] = {
            "is_combination": True,
            "current_quantity": available_sets_full
        }
        if filtered_dates:
            output["dates"] = nested_date_structure(filtered_dates)

        # Write the output JSON file for the set to the OUTPUT_DIR
        out_filepath = os.path.join(OUTPUT_DIR, f"{set_id}.json")
        with open(out_filepath, "w") as outfile:
            json.dump(output, outfile, indent=4)
        print(f"Set {set_id}: full availability {available_sets_full} written to
{out_filepath}")

if __name__ == "__main__":
    main()
```

Add available to parents

Prompt: *Calculate Parent Availability from Equipment JSON*

Context

You are given a JSON file representing a list of equipment items. Each item has:

- A `displayname` and `parent_equipment_number`
- A list of `children`, and each child contains:
  - `current_quantity`: how many are in stock
  - `quantity`: how many are needed to build **1 parent**
  - `usage_by_day`: reservations per day (structured as `YYYY-MM` → day → reserved quantity)

Task

Write a PHP function that:

1. **Input:** Takes an array decoded from this JSON structure.
2. **Output:** Adds **two new keys** to each **parent item**, inserted directly **after the** `displayname` key:
   - `'available'`: the maximum number of parents that can be assembled based on the current stock and child requirements.
     - For each child, calculate: `floor(current_quantity / quantity)`
     - The **minimum** across all children determines the parent's availability.
   - `'available_by_day'`: a structure similar to `usage_by_day`, but showing the availability of the parent **on specific days**, accounting for **child reservations**.
     - For each day, subtract reserved quantity from current stock per child.
     - Then divide by the number of that child required per parent.
     - Again, take the **minimum** across children for that day.
     - If that value equals the overall `'available'`, omit the day.
     - If it drops below `0`, set it to `0` (administrative error handling).
     - Output structure:

```
{
 "2025-06": {
  "13": 3,
  "14": 2
```

```
    },
    "2025-07": {
     "1": 4
    }
   }
```

- Ensure the final order of keys for each parent is:
  - `parent_equipment_number`
  - `displayname`
  - `available`
  - `available_by_day`
  - `children`

Function name: `addParentAvailability`

Code generated

```
/**
 * Enhances parent equipment data with:
 * 1. An overall 'available' count based on current stock of children.
 * 2. A per-day 'available_by_day' structure that shows date-based limitations.
 *
 * 'available' shows the max number of parents that can be built given stock.
 * 'available_by_day' shows reductions on specific dates due to reservations
 * on child equipment. Dates with full availability are omitted. Negative availability
 * (due to administrative error) is corrected to 0.
 *
 * @param array $equipmentSet The array of parent equipment items with their children.
 * @return array The modified array with the new availability keys added.
 */
function addParentAvailability(array $equipmentSet): array {
    foreach ($equipmentSet as &$parent) {
        $minAvailable = PHP_INT_MAX;
        $dailyAvailability = [];

        foreach ($parent['children'] as $child) {
            $neededPerParent = $child['quantity'];
            $childAvailable = floor($child['current_quantity'] / $neededPerParent);
            $minAvailable = min($minAvailable, $childAvailable);
```

```
            if (!isset($child['usage_by_day'])) continue;

            foreach ($child['usage_by_day'] as $ym => $days) {
                foreach ($days as $day => $usedQty) {
                    $available = floor(($child['current_quantity'] - $usedQty) /
$neededPerParent);
                    if ($available < 0) $available = 0;

                    // If not yet set, initialize with high value
                    if (!isset($dailyAvailability[$ym][$day])) {
                        $dailyAvailability[$ym][$day] = $available;
                    } else {
                        $dailyAvailability[$ym][$day] = min($dailyAvailability[$ym][$day],
$available);
                    }
                }
            }
        }

        // Clean up: remove days that match the full availability
        foreach ($dailyAvailability as $ym => &$days) {
            foreach ($days as $day => $value) {
                if ($value >= $minAvailable) {
                    unset($days[$day]);
                }
            }
            if (empty($days)) {
                unset($dailyAvailability[$ym]);
            }
        }

        // Rebuild parent structure with displayname -> available -> available_by_day -> rest
        $newParent = [];
        foreach ($parent as $key => $value) {
            $newParent[$key] = $value;
            if ($key === 'displayname') {
                $newParent['available'] = $minAvailable;
                $newParent['available_by_day'] = $dailyAvailability;
```

```
            }
        }

        $parent = $newParent;
    }

    return $equipmentSet;
}
```

Correction (last step)

```
 □□Prompt: Calculate Parent Availability from Equipment JSON (Full Version with Bug Fix)

## □□Context
You are given a JSON file representing a list of equipment items. Each item represents a
**parent item**, and contains:
- A `displayname` and `parent_equipment_number`.
- A list of `children`, each with:
  - `current_quantity`: number in stock.
  - `quantity`: number required to build **1 parent**.
  - `usage_by_day`: reservations per date in the structure:
    ```json
    {
      "2025-06": {
        "13": 2,
        "14": 3
      }
    }
    ```

□□Goal
Write a PHP function that does the following:

□ 1. Input
The function receives a decoded JSON structure (i.e., a PHP array), containing an array of
```

parent items, each with `children`.

🔢 2. Calculates `available`
For each parent item:
- Determine how many complete parents can be built from the current stock of all required children.
- For each child:
  ```php
  floor(current_quantity / quantity_required_per_parent)
  ```
- The **minimum** result across all children is the number of complete parent items that can be built.
- Add this number as a new key:
  ```php
  "available": <int>
  ```
- Insert it **immediately after `displayname`** in the parent's object.

🔢 3. Calculates `available_by_day`
Also add a new key **after** `available`, named:

```json
"available_by_day": {
  "YYYY-MM": {
    "DD": <int>
  }
}
```

This shows for which dates the availability of parents is **lower than** the full availability due to **child reservations**.

Logic:
- Loop through **all dates** that appear in any child's `usage_by_day`.
- For each date:
  - For each child:

    - Subtract reserved amount on that date from current_quantity.
    - Compute:
      ```php
      floor((current_quantity - reserved) / quantity_required)
      ```
    - If the result < 0, treat it as 0.
  - The **minimum** across all children determines the **parent's availability for that date**.
  - If that value is **less than** the overall `available`, include it in `available_by_day`.
  - If the value is **equal to or greater than** `available`, skip it (don't output it).
- If no days are limited, the key `available_by_day` may be an empty object.

□ 4. Output Structure
Each parent item will be rewritten to insert:
- `available`
- `available_by_day`

…right after `displayname`.

Example:
```json
{
  "parent_equipment_number": 1234,
  "displayname": "Cool Parent",
  "available": 2,
  "available_by_day": {
    "2025-06": {
      "13": 1,
      "14": 0
    }
  },
  "children": [ ... ]
}
```

□ 5. Edge Cases

```
- If no usage data is available for any child, the daily availability is not included.
- If calculated availability is negative on any date, set it to `0` (administrative error).
- Never allow availability on any date to exceed the total `available`.

---

This prompt includes all requirements, handles bugs related to overestimation, and ensures robust availability calculations per parent.
```

Code:

```
/**
 * Enhances each parent item with:
 * - 'available': max number of parents that can be built from current stock.
 * - 'available_by_day': date-specific limits due to reservations in children.
 *
 * For 'available_by_day', only include dates where the availability is lower than 'available'.
 * Values below 0 are treated as 0.
 * Dates with full availability are skipped.
 *
 * @param array $equipmentSet Decoded JSON with parent items and their children.
 * @return array Modified data with availability info inserted.
 */
function addParentAvailability(array $equipmentSet): array {
    foreach ($equipmentSet as &$parent) {
        $minAvailable = PHP_INT_MAX;
        $dailyAvailability = [];
        $allDates = [];

        // Step 1: Compute total availability and gather all relevant dates
        foreach ($parent['children'] as $child) {
            $neededPerParent = $child['quantity'];
            $childAvailable = floor($child['current_quantity'] / $neededPerParent);
            $minAvailable = min($minAvailable, $childAvailable);

            if (!isset($child['usage_by_day'])) continue;

            foreach ($child['usage_by_day'] as $ym => $days) {
                foreach ($days as $day => $_) {
```

```
                $allDates[$ym][$day] = true;
            }
        }
    }

    // Step 2: For every known date, compute availability from all children
    foreach ($allDates as $ym => $days) {
        foreach ($days as $day => $_) {
            $minForDay = PHP_INT_MAX;

            foreach ($parent['children'] as $child) {
                $needed = $child['quantity'];
                $stock = $child['current_quantity'];
                $used = $child['usage_by_day'][$ym][$day] ?? 0;
                $available = floor(($stock - $used) / $needed);
                $available = max(0, $available); // prevent negative availability

                $minForDay = min($minForDay, $available);
            }

            if ($minForDay < $minAvailable) {
                $dailyAvailability[$ym][$day] = $minForDay;
            }
        }
    }

    // Step 3: Rebuild parent array, insert new keys after 'displayname'
    $newParent = [];
    foreach ($parent as $key => $value) {
        $newParent[$key] = $value;
        if ($key === 'displayname') {
            $newParent['available'] = $minAvailable;
            $newParent['available_by_day'] = $dailyAvailability;
        }
    }

    $parent = $newParent;
}

return $equipmentSet;
```

```
}
```

Open issues

1. At the moment we only read and cache equipment belonging to a client. Set can consist out of items not belonging to any client directly, so there is no data available (in equipment). This means that there's no data for the availabilty in the cache. This would mean that we need to load and cach more equipment and more plannedequipment.

   These are appr. 1000 equipments (current number is 1200).

2. The calendar format for each month is:

   ```
   Array ( [1] => 1 [2] => 1 [3] => 1 [4] => 1 [5] => 1 [6] => 1 [7] => 1 [8] => 1 [9] => 1 [10] => 1 [11] => 1 [12] => 1 [13] => 1 [14] => 1 [15] => 1 [16] => 1 [17] => 1 [18] => 1 [19] => 1 [20] => 1 [21] => 1 [22] => 1 [23] => 1 [24] => 1 [25] => 1 [26] => 1 [27] => 1 [28] => 1 [29] => 1 [30] => 1 )
   ```

Revision #17
Created 2025-03-29 18:15:40 UTC by Max
Updated 2025-06-08 10:53:32 UTC by Max