Design and Implementation of Jamazon

Development Team
Roger Peña Pérez

February 2, 2026

1 Introduction

The *Jamazon* project was designed with the aim of managing and optimizing task planning in a fast-food environment. The application domain covers the temporal administration of limited resources such as fryers, ovens, cooks, delivery drivers.

The central problem that the software resolves is the efficient allocation of time intervals to perform operational tasks (events). Each event, like "Prepare Order", requires the simultaneous blocking of a specific amount of multiple resources during a determined duration. The system must guarantee that, at no moment of the proposed interval, the accumulated demand of a resource exceeds its maximum installed capacity. Furthermore, hierarchical dependency constraints are handled; for example, the use of computer equipment automatically blocks a quota of the "Electricity" and "Internet" resources.

The application is capable of suggesting to the user the nearest possible moment to start a task when resources are occupied, calculating available "gaps" in the agenda that satisfy all resource constraints simultaneously.

2 Architecture

2.1 Structure and Design Decisions

To guarantee system maintainability and scalability, a modular architecture was chosen that strictly decouples business logic from the user interface. It was decided to use JSON files as the data persistence mechanism instead of a relational database. This decision was made to simplify project portability and eliminate the need for external services running in the background, given that the expected data volume for a small venue is manageable in memory.

The structure is clearly divided into:

1. **main.py:** Entry point that orchestrates the application life cycle.
2. **modules (Backend):** Contains the hard logic and algorithms.
3. **guicore (Frontend):** Manages interaction with the graphical library, consuming backend services without knowing their internal implementation.

2.2 Development Environment

The implementation has been carried out in a Linux environment using Git and VSCode; compatibility has been tested on Arch Linux, Ubuntu, and Windows 11 systems.

3 Module Logic

3.1 utils.py

This module only contains useful functions used throughout the project. It was created with the aim of avoiding repeated code in the project and keeping individual functions used in it more organized.

- **log:** This function redirects input data to a file, used for debugging.
- **getsourcesdependency:** This function implements a DFS which will be explained in following sections.
- **CheckISODate:** Returns True if the input ISODate is correct.
- **getsavedjson:** Returns a dictionary with the data of a specified JSON.
- **readfile:** Loads a file directly into memory.
- **tominute:** Converts a date into its corresponding value in minutes.
- **addtodict:** This function recursively adds a value to a dictionary (more details in utils.py).
- **eventoptionlabel:** Generates a small descriptive text from an event.
- **buildeventoptionlabels:** Generates a list with the data returned by the previous function for each event.
- **formateventinfo:** Generates the complete information of an event.

3.2 iohandler.py

Here lies the BasicHandler class, which handles most JSON file IO.

- Saves to a JSON.
- Reads data from a JSON.
- Extracts the extension of a file.
- Converts a JSON to dict.
- Converts a dict to JSON.

3.3 gvar.py

This module's main function is to load JSON data and it is responsible for containing the global data for the entire program.

3.4 events.py

The Event class is found here; it inherits from BasicHandler and is responsible for saving a specific task within itself, saving its data. Furthermore, it implements data transformations:

- Converts a task to dict.
- Converts a task to str.
- Extracts which resources are not used with a specified one.

4 calendar.py

Since there is much to explain in this module, it was divided into several points. Here lies the Calendar class, which is responsible for:

- Adding tasks. One of the main challenges faced during development was handling time ranges based on Unix timestamps, which generate very large numbers (e.g., 167888888). Creating an array of that size for the Segment Tree would consume too much memory. To solve this, a **Coordinate Compression** technique was implemented within this logic. The algorithm identifies points of interest (starts and ends of existing events) and maps them to small sequential indices (0, 1, 2...), allowing for the construction of an efficient tree that only considers moments where the system state changes, ignoring intermediate "empty" periods.
- Sorting dates.
- Removing old tasks upon system startup.

4.1 add_event

At the time of making this function, the problem was found that time ranges could be very large, so it was necessary to implement coordinate compression so that (R - L) is smaller to be able to verify more quickly if the new task can be added in that range.

4.2 suggest_brute_lr

This function implements via brute force the logic behind suggesting a date range to the user. What it does is that for each of the events...

4.3 check_available

Verifies if a given date range is available to assign resources. This function checks if, for the specified time interval, the amount of requested resources does not exceed the total available capacity, ensuring that no conflicts exist with other already scheduled tasks.

4.4 remove

This function removes a specific task from the calendar, freeing the resources that were assigned to said task and updating the data structure to reflect the elimination.

4.5 sort

This function sorts the dates (dictionary keys) by start time.

4.6 remove_old_events

This function is responsible for iterating through all registered tasks and removing those whose end date is prior to the system's current date and time. This allows keeping the calendar clean and optimized , reducing the size of the segment tree needed for future calculations.

5 Challenges and Solutions

During the development cycle, several technical obstacles were faced that defined the final architecture:

- **Dependency Validation:** Managing resources that depend on others (e.g., Internet requires Electricity) complicated verification logic. This was resolved by implementing a depth-first search (DFS) in `utils.py` that recursively traverses the dependency graph to ensure all base resources are available before confirming a task.
- **Data Persistence:** Maintaining consistency between JSON files and in-memory objects was complex. Methods were standardized in `iohandler.py` to automatically serialize and deserialize objects, avoiding data corruption upon unexpected program closure.

6 gui_core

All classes that inherit from `CTkToplevel` or graphic library components are encapsulated in this folder, acting as a bridge between user interaction and logic.

6.1 EventCreator

This class manages the interface for defining **event types** in the system. It allows the user to specify the name of a new operational activity (e.g., "Fry Fries") and associate it with a list of resources needed for its execution.

6.2 TaskCreator

This class implements the **scheduling** interface. It provides fields to enter the start and end date and time, as well as a selector for previously defined activities. It interacts directly with the `calendar` module to verify resource availability in the requested interval and confirms the creation of the event instance in the schedule.

6.3 ResAdder

Class responsible for managing the base resource inventory. It allows the administrator to register new resources in the system (e.g., "Fryer", "Staff", "Electricity"), defining their initial maximum capacity. It updates the global resource data structure, making them available to be requested by future tasks.

6.4 EventShower

Viewer for active and scheduled events. Generates a dynamic list of buttons representing each task in the system's execution queue, showing its ID, name, and assigned time interval. Interacting with an element displays detailed information about the specific event, facilitating real-time monitoring of local operations.

6.5 ResourceShower

Resource consultation interface. Displays a list of all registered resources in the system along with their total available quantity. Allows administrators to have a clear view of the premises' installed capacity and verify each resource's properties.

6.6 TaskRemover

Management tool for task cancellation. Provides an interface where scheduled events are listed in a dropdown menu, allowing the user to select and delete a specific task. Upon confirming the action, it invokes backend cleanup methods to free resources reserved by said event in the *Segment Tree*, immediately updating system availability.

References

[1] Python Software Foundation. *Python 3.12.1 Documentation*. Available at: `https://docs.python.org/3/`.

[2] Tom Schimansky. *CustomTkinter: Official Documentation And Tutorial*. Available at: `https://customtkinter.tomschimansky.com/`.

[3] CP-Algorithms. *CP-Algorithms: Competitive Programming Library*. Available at: `https://cp-algorithms.com/`.