Haus:
Programmer's Documentation

Izzy Harker, Andrew Rehmann, Alex Peterson Santos, Connie Williamson, and Carter Young
Department of Computer Science: University of Oregon
CS 422: Software Methodologies
Prof. Anthony Hornof
March 13, 2024

GitHub Repository

Table of Contents

1. Introduction

This documentation should assist the programmers of Haus in making future changes and modifications to the app and executable, including adding functionality, maintaining the software, and changing mechanisms for hosting the system.

All code for this project was written in Python, CSS, and JSX (JavaScript XML). The entirety of the backend functionality for this project was implemented in Python 3.12.2. The frontend functionality was implemented in both CSS and JSX using the ReactJS user interface library. A Flask application on the server is used to pass information between the frontend and the backend.

Some notes about the relationships between different files or modules are provided in this documentation. However, for a better understanding of these interactions, we encourage prospective programmers or maintainers to visit the Software Design Specification for this program.

2. Program Files

This section details the program files. We also included brief mentions of CSS files in this section.

Here is where each file fits into the software architecture defined in the SDS:
- DataInput.py: Data Access and Input module
- AutoAssign.py: Automatic Chore Assignment module
- Login.py: Chore Server Connection module
- Flask_integration.py: Chore Server Connection Module
- App.css and App.jsx : Household User Interface
- Main.jsx and index.css: Household User Interface
- components directory: Household User Interface

2.1 DataInput.py

This file is the sole reader and writer for the Household Data Storage database. That means DataInput.py is imported by many other parts of the program that wish to access or make modifications to the database.

To fulfill its purpose, it provides a set of "getter" functions (return information from the database) and "setter" functions (update the database from given arguments).

Chore Class

The Chore class defines an object representing a single chore. Chore objects are only used in DataInput.py and AutoAssign.py. Unlike a dictionary describing a row from the CSV, it is guaranteed to contain all of the keys (attributes) of the chore, and can contain more useful datatypes, like enums or datetime objects. This makes them easy to work with.

However, please keep in mind that these are only used in the backend. The Python objects are turned into CSV rows when updating the database or into JSONs when chore information needs to be sent to the frontend.

The chore class has the following attributes, which are populated by the constructor using a dictionary representing a CSV row:
- name (string)

- id (string)
- description (string)
- category (string) (not in use)
- expected_duration (integer)
- status (CHORE_STATUS enum)
- assignee_id (string)
- deadline_date (date object)
- frequency (integer)
- completion_date (date object)

The Chore class also has a method to_csv_row, which returns its output as a dictionary representing a CSV row. This essentially returns the dictionary that was used to create the Chore object from the constructor in the first place. This facilitates switching easily from Chore objects and CSV row dictionary representations.

Setter Functions

def add_occupant_name(filename, occupant_uid, occupant_username, occupant_password) -> bool:

- Adds a new CSV row representing a user the occupants file at filename
- The new row represents a user with ID occupant_uid, name occupant_username, and password occupant_password
- Returns a bool indicating success

def new_chore_by_object(chore: Chore) -> None:

- Adds a new entry to the chores.csv database using the attributes from the chore object
- The Chore object may already have an id (e.g. a repeating chore, whose ID is similar to the previous ID) or not (an ID will be generated for it)

```
def new_chore_by_args(name: str,
            desc: str,
            id: Union[str, None] = None,
            category: str = "",
            expected_duration: int = 10,
            status: Union[CHORE_STATUS, None] = CHORE_STATUS.UNASSIGNED,
            assignee_id: Union[str, None] = None,
            frequency: int = 0,
            deadline_date: Union[date, None] = None,
            completion_date: Union[date, None] = None) -> None:
```

- Adds a new chore to the chores.csv database using the passed in chore properties def update_chore_by_object(chore: Chore) -> None:

- Update an existing CSV row in the chores.csv database to match the attributes of the given Chore object
- The existing chore is found based on id

def set_chore_complete(chore_id: str) -> None:

- Update the chores.csv database to show the chore with id chore_id as complete
- This also sets the “Completion Data” attribute of that chore to the current date

def remove_user(username: str, occupant_filepath: str) -> None:

- Remove a user of name username from the occupants.csv database at occupant_filepath

Getter Functions

def get_username_list(filename: str) -> list[str]:

- Return a list of the usernames of users from the occupants.csv at filename

def get_password(filename: str, username: str) -> str:

- Return the password for the user with username username

def retrieve_occupants_names_and_uids(OCCUPANTS_FILEPATH: str) -> dict[str, str]:

- Return a dictionary mapping occupant IDs to their names based on the occupants information from the occupants.csv at OCCUPANTS_FILEPATH

def retrieve_occupant_uid_from_username(username: str, OCCUPANTS_FILEPATH: str) -> str:

- Returns the ID of the first user in the occupants.csv file at OCCUPANTS_FILEPATH matching the given username username

def get_chore_by_id(id: str) -> Chore:

- Return a Chore object populated with all of the attributes of the chore with id id as present in the chores.csv database.

```
def get_chores_by_filters(assignee_id: str = None,
                status: CHORE_STATUS = None,
                min_deadline_date: date = None,
                max_deadline_date: date = None,
                repeating_only: bool = False
                ) -> list[Chore]:
```

- Returns a list of Chore objects which match the stipulations given as the function arguments.
- The main use case for this is finding unassigned chores to assign or finding repeating chores to renew (generate a new instance of). These uses are covered by AutoAssign.py.

def get_user_ids() -> list[str]:

- Returns a list of all the user IDs in the chores.csv database.

Other/Helper Functions

def generate_uid() -> str:

- Using the Python uuid library, generate a unique ID which can be used to identify an occupant or a chore in the occupants.csv and chores.csv files respectively.

def ensure_csv_headers(filename: str, headers: list[str]) -> None:

- Checks the CSV headers (the first row) of file at filename and compares them against headers
- If there are missing headers, they are added to the CSV file and the file is rewritten

Deprecated Functions

These functions no longer exist in the source code. Many of them include command-line interface functionality (i.e. prompting the user for information), which was helpful during an earlier stage of development but is no longer relevant.

def append_to_csv(filename, data) [deprecated]:

- Allows user input to be appended to the appropriate and specified CSVs

def get_haus_info() [deprecated]:

- Ask users for Haus information, then sends user-generated data to CSV/JSON for storage and retrieval. Also assigns each Haus a UID for easy reference.
- Inputs:
    - Haus name
    - Haus type
    - Num Occupants

def verify_uid_and_get_occupants(filename, uid) [deprecated] :

- Checks a user-specified UID and searches the appropriate file for it (based on the function which calls it, this will differ). Returns ‘True’ if the UID is found and ‘False’ otherwise.

def save_occupant_names(filename, occupant_uid, occupant_name) [deprecated]:

- Upon the user entering a new name or signing in for the first time, this function saves them as a new occupant and includes a UID and name def initialize_chores(chores_file) [deprecated]:

- Initializes a default and pre-defined set of chores and saves them to the chores CSV/JSON

def rank_chores(occupants_file, chores_file, chore_rankings_file) [deprecated]:

- Allows users to rank chores. Takes an occupant name as input and, based on that user's inputs, stores their preference for both default and user-added chores

def add_or_remove_chores(chores_file) [deprecated]:

- Acts as a simple logic gate for user input when deciding whether to add or remove chores

def add_chore(chores_file) [deprecated]:

- Adds a chore from chores.csv/.json and asks user for input regarding the chore's characteristics

def remove_chore(chores_file) [deprecated]:

- Removes a chore and its characteristics from chores.csv/.json

def list_chores(chores_file) [deprecated]:

- Provides a list of chores so that users may rank their preference for them

def save_chore_rankings(chore_rankings_file, rankings) [deprecated]:

- Saves chore preferences per user per chore

def update_chore(chore: Chore) -> None [deprecated]:

- Given a chore object, this function updates the CSV containing that chore to match the object's given attributes. If the specified chore does not exist, the function throws an error

def main() [deprecated]:

- Supports CLI for users to input new chores, add occupants, and rank existing chores. This was implemented purely to ensure the logic for DataInput.py was stalworth, and has since been deprecated

2.2 AutoAssign.py

This file handles workload calculation, automatic chore assignment (which is itself based on workload calculations), and renewal of repeating chores. It relies heavily on DataInput.py as an intermediary to access and manipulated the information in the chores.csv database.

Functions

def assign_unassigned_chores() -> None:

- Finds unassigned chores and assigns them to users with the help of user_workload

def assign_chore(chore: Chore, assignee_id: str) -> None:

- Attaches an occupant's ID to a chore for any given week

def user_workload(user_id: str) -> int:

- Calculates an occupant's chore workload based on the chores assigned to them (regardless of completion) which have a deadline within the last 7 or next 7 days
- Used in assign_unassigned_chores to ensure an equitable and consistent assignment of tasks

def renew_repeating_chores() -> None:

- Renews all repeating chores that are ready to be renewed (i.e., they are both completed and the deadline has passed).
- This also marks these chores as renewed such that they will never be renewed again (instead, the new instance of the chore will be renewed later, when it is completed).

    def increment_id(old_id: str) -> str:

    - This is a function defined within the scope of renew_repeating_chores
    - Adds a number in parentheses after a chore ID to indicate a repeating chore
    - e.g. if a chore has id "somelonguuid" its successor will be "somelonguuid(1)"

2.3 Login.py

Login.py - The file responsible for establishing methods for creating, deleting, and verifying Haus users. It sends occupant data to DataInput.py for storage, which is then visualized on the system's frontend.

<u>Function Description</u>

def create_user(username, password, occupant_filepath):

- Used to add new occupants to a Haus and correlate them with a user-defined username and password. Calls add_occupant_name in DataInput.py to append new users to the existing occupants.csv/.json
    - If user is successfully added, returns True.
    - If user already exists, returns False

def delete_user(username, password, occupant_filepath):

- Verifies that a user exists, then removes from the Haus. References the occupants.csv/.json, finds the specified username and password, and removes the appropriate row from the file

def log_in_user(username, password, occupant_filepath):

- Verifies that a username and password are valid and logs in the user attached to those credentials. References the occupants.csv/.json and searches for the username and password.
    - If user exists, displays Haus landing page
    - If user does not exist, function returns an error

def verify_user_exists(username, occupant_filepath):

- Given a username, verifies that that username exists in the occupants file

2.4 flask_integration.py

flask_integration.py - This file contains the flask integration between the frontend and backend. It translates and sends user requests from the frontend to the backend, storing it appropriately and accurately.

Function Description

def after_request(response):

- Processes a request after it is returned from flask, in order to allow cross-origin resource sharing (cors) to successfully communicate. Without this, depending on your environment, communication may be automatically blocked for safety reasons.

def flask_login_user():

- Flask endpoint for logging in a user. Accessed by POSTing a request to ‘/user/login’ with a form with parameters ‘user’ and ‘pass’ for the username and password respectively.
- Returns a JSON with ‘user_exists’ and ‘pass_valid’ fields.
    - user_exists: True if there is a user associated with the username, false otherwise

        - pass_valid: True if the password matches the specified user's password, false otherwise

def flask_create_user():

- Flask endpoint for creating a user. Accessed by POSTing a request to '/user/create' with a form with parameters 'user' and 'pass' for the username and password respectively. Fails if the username already exists.
- Returns a JSON with 'success' field.
    - success: True if the user was successfully created, False if the user could not be created (username already exists)

def flask_serve_users():

- Flask endpoint for getting the users. Takes a GET request.
    - Input:
      GET request
    - Output:
      JSON reply with list of users and IDs

def flask_logout_user():

- Flask endpoint to remove a user from an active session. Effectively sets the user to null.

def flask_delete_user():

- Flask endpoint to remove a user from the occupants file. Removes the specified username and the corresponding password. Then, ensures that user_id is *null*.

def flask_complete_chore():

- Flask endpoint for marking a chore as complete. Takes a POST request with a form attribute with a json/dict of keys 'Chore ID'.
    - Input:
      POST form request with 'Chore ID'
      Chore ID: The ID of the chore to be marked as complete
    - Output:
      JSON reply with 'success' parameter
      success: True if the chore completion succeeded, False if it failed

def flask_create_chore():

- Flask endpoint for creating a chore. Takes a POST request with a form attribute with a json/dict of keys 'Chore Name', 'Description'.
    - Input:
      POST form request with 'Chore ID'
      Chore ID: The ID of the chore to be marked as complete

- Output:
  JSON reply with 'success' parameter
  success: True if the chore completion succeeded, False if it failed

def flask_serve_chores():

- Flask endpoint for serving chores. Takes a POST request with a form attribute with a json/dict of keys 'user'.
    - Input:
      POST form request with 'user'
      user: The username of the provided user. Leave empty if fetching all chores
    - Output:
      JSON reply with a list of the chores assigned to the user. Looks like:

```
[
        {
          'Chore ID': *value*,
          'Chore Name': *value*,
          'Description': *value*,
          'Category': *value*,
          'Expected Duration': *value*,
          'Status': *value*,
          'Assignee ID': *value*,
          'Deadline Date': *value*,
          'Frequency': *value*,
          'Completion Date': *value*
        },
        …
]
```

def flask_assign_chores():

- Flask endpoint for auto assigning users. Takes a GET request and calls ‘assign_unassigned_chores’.
    - Input:
      GET request

The main .css and .jsx files used to implement Haus are as follows. They are all contained in the Frontend/src/ directory of the project.

2.5 App.css and App.jsx

App.css and App.jsx - The files responsible for defining the basic UI of Haus, including entry boxes, containers, and titles. These files dictate the aesthetic characteristics of our frontend.

function App()

- Input: None
- Output: A dynamically rendered frontend module containing the login forms and the components for the home screen of the app

2.6 Main.jsx

Main.jsx - This file provides a root from which to display the React components defined in App.jsx.

2.7 index.css

index.css - This file contains default and global styling guidelines.

2.8 components (directory)

components/- This directory contains three sub-directories (Chores, House, NavBar), which each contain the components for their respective sections.

2.8.1 NavBar/

- Help.jsx - Defines and exports the "Help" button component
    - Input: None
    - Output: html <a> element which links to the Github
- Logout.jsx - Defines and exports the "Logout" button component
    - Input: props (React construct for passing arguments)
        - setLogin
    - Output: html <button> which logs a user out when clicked
- Title.jsx - Defines and exports the "Title" component
    - Input: props (React construct for passing arguments)
        - title
    - Output: html <div> displaying the app title

- NavBar.jsx - Defines and exports the "NavBar" component
    - Input: props (React construct for passing arguments)
        - renderButtons
        - setLogin
    - Output: Complete navigation bar, including all the elements above.
- NavBar.css - Contains rendering instructions for the 4 files described above.

2.8.2 House/

- HouseContainer.jsx - Defines and exports the "HouseContainer" component, which dynamically fetches the occupants of the Haus.
    - Input: props (React construct for passing arguments)
    - Output: html <div> containing the HouseDetails component
- HouseDetails.jsx - Defines and exports the "HouseDetails" component, which renders the occupants and buttons.
    - Input: props (React construct for passing arguments) with elements:
        - houseName
        - houseData
        - setLogin
    - Output: House information and working buttons.
- HouseContainer.css - Contains rendering instructions for the 2 files described above.

2.8.3 ChoreCards/

- ChoreContainer.jsx - Defines and exports the "HouseContainer" component, which creates the space for the chore cards and title.
    - Input: None
    - Output: html <div> containing the ChoreList component
- ChoreList.jsx - Defines and exports the "ChoreList" component, which dynamically renders the title and the chores in a grid.
    - Input: props (React construct for passing arguments)
    - Output: Title and a card element for each assigned chore
- Chores.css - Contains rendering instructions for the 2 files described above.

3. Database

The database, referred to in the SDS as the Household Data Storage database, consists of two CSV files. The DataInput.py file is the sole reader and writer of these files. When information is sent to and from React (i.e. the user interface), JSON files containing the same information are used instead.

The database consists of two files in the ‘csvs’ directory:

- chores.csv
- occupants.csv

Their contents and function follow. You will find also a section about a file chore_rankings.csv which is now deprecated and serves no purpose in the program.

3.1 chores.csv and chores.json

This file hosts data related to the chores which a) are included by default and b) are added by the user. A user may add or remove chores at will, and may do so via the frontend as seen in the Haus User Documentation.

The data fields within Chores.json are as follows: Chore ID, Chore Name, Description, Category, Expected Duration, Status, Assignee ID, Deadline Date, and Completion Date.

- **Chore ID** is utilized as a primary key and allows occupants to track their preference for various chores
- **Chore Name** is a short label for the chore, and is how they are represented in Haus’ frontend
- **Description** is a short summary of what the chore involves, so that all roommates understand the success and completion criteria for it
- **Category** separates chores by either sections of the house or the kinds of tasks the chores involve
- **Expected Duration** is a user-defined value which estimates how long a chore will take to complete in minutes. It is used to assign chores equitably (i.e., ensure each occupant receives roughly the same amount of work over a certain time period)
- **Status** is a measure of whether the chore has been assigned and whether it has been completed. It is used as a logic gate for chore assignment and rotation
- **Assignee ID** is used to track what occupant is responsible for a chore at any given time
- **Deadline Date** defines when a chore should be completed by. This date automatically changes when a chore is completed
- **Completion Date** defines when a chore was completed. It is also used to assign chores equitably (i.e., ensure the same chore is not assigned to one individual overly often)

Chores.json is the most critical file in Haus, since it is responsible for the primary functionality of AutoAssign.py and, indeed, guarantees that Haus operates in a fair and consistent manner.

3.2 occupants.csv and occupants.json

This file hosts data related to the occupants of a given Haus.

The data fields within Occupants.json are as follows: Occupant ID and Occupant Name

- **Occupant ID** is utilized as a primary key and allows occupants chore rankings and assignments to be tracked
- **Occupant Name** acts as a label for each occupant in Haus' frontend.

3.3 chore_Rankings.json (deprecated)

(Currently unused)
This file hosts data related to the chore rankings for each occupant of a Haus. Specifically, each occupant is asked to provide a 0, 1, or 2 for each chore. These values correspond to 'do not prefer', 'no preference', and 'prefer'. The contents of Chore_Rankings.json are less lengthy than that of Chores.json, but the data included is critical in assigning the correct chores reliably.

The data fields within Chore_Rankings.json are as follows: Occupant ID, Chore ID, and Rank.

- **Occupant ID** is utilized as a foreign key to track which occupant is ranking chores
- **Chore ID** is utilized as a foreign key and allows occupants to track their preference for various chores
- **Rank** is a user-defined preference value for each chore. Input includes 0, 1, and/or 2, as described above.

4. Common Tasks

Adding a chore

Event	**Code**	**Description**
1. A user enters a new chore to add	def flask_create_chore():	The user inputs a description of their desired chore, including its name, description, category, and the expected duration for completion.
2. The user's specifications are sent to DataInput.py	def new_chore_by_args():	The user's input is 'piped' to DataInput for collection and storage in key data files so that the chore may be continually tracked and reliably assigned.
3. The user's specifications are stored in the chores file	def new_chore_by_args():	The user's input is stored in chores.csv/.json for future reference. All data fields will either have been specified by the user in their initial entry, or generated automatically by AutoAssign.py.

Adding an occupant

Event	**Code**	**Description**
1. A new user logs in for the first time	def flask_create_user():	The user inputs their username and password.
2. The user's specifications are sent to Login.py	def create_user(username, password, occupant_filepath):	The user's input is 'piped' to Login for comparison to the existing set of occupants.
3. The occupant's specified attributes are stored in the occupants file.	def add_occupant_name (filename, occupant_uid, occupant_username, occupant_password):	After verification that the user does not already exist, their input is saved in the occupants.csv and occupants.json.

Removing occupants

Event	**Code**	**Description**
1. A user selects ‘Delete Account’ in the UI	def flask_delete_user():	The user presses ‘Delete Account’ and enters their password to permanently delete their account.
2. The user’s selection is sent to Login.py	def delete_user(username, password, occupant_filepath):	The user’s input is ‘piped’ to Login to ensure they have input the correct password.
3. The user’s specifications are compared with the occupants file	def get_password (occupant_filepath, username):	The name input by the user is compared with the contents of the occupants file. If the name does not exist in the file, the function returns an error.
4. The specified occupant is removed from the occupants file	def remove_user(username, occupant_filepath)	After verification, the specified occupant and all of its attributes are removed from occupants.csv and occupants.json.

References

The Project 2 Description was provided by Professor Anthony J. Hornof.

The templates for Programmer’s Documentation were provided by Professor Anthony J. Hornof’s “World Tax Planner Programmer’s Documentation” (1993) and “EyeDraw Programmer’s Documentation” (2004).