Account Algorithms

Algorithm 1 - signUp()

Input: Email/Phone Number, Password
Output: boolean, true if successful, false if failure
1.1 Validate the email/phone number format.
1.2 **if** the format is invalid:
1.3 *return* false.
1.4 Check if the email/phone number already exists in the system.
1.5 **if** it exists:
1.6 *return* false.
1.7 Validate the password against security requirements (e.g., length, complexity).
1.8 **If** invalid:
1.9 *return* false.
2.0 Create a new account object with the provided credentials.
2.1 Save the account to the database.
2.2 Automatically log the user in.
2.3 *return* true.

Algorithm 2 - deleteAccountInfo()

Input: User ID
Output: boolean, true if successful, false if failure
2.1 Verify if the user is logged in.
2.2 **if** not logged in:
2.3 *return* false.
2.4 Prompt the user for confirmation of account deletion.
2.5 **If** the user cancels the confirmation:
2.6 *return* false
2.7 **else**:
2.8 Delete all associated data for the User ID from the database.
2.9 Remove the account record from the database.
2.10 Display a success message to the user.
2.11 *return* true.

Algorithm 3 - updateAccountInfo()

Input: User ID, New Details (e.g., Email/Phone/Password)
Output: boolean, true if successful, false if failure

1.1 Verify if the user is logged in.
1.2 **if** not logged in:
1.3 *return* false.
1.4 Check which fields the user wants to update.
1.5 **for** email or phone number: Validate the new value's format.
1.6 **if** invalid:
1.7 *return* false.
1.8 **for** passwords: Ensure compliance with security rules.
1.9 **if** invalid:
2.0 *return* false.
2.1 Apply the updates to the user’s account.
2.2 Save the updated account details in the database.
2.3 Notify the user of the successful update.
2.4 *return* true.

Algorithm 4 - moveToNewHouse()

Input: None (Triggered automatically or manually by the user)
Output: boolean, true if successful, false if failure
1.1 Use the update feature and enter your new home address.
1.2 Begin the relocation process.
1.3 Adjust device settings to adapt to the new environment (e.g., energy settings, device locations).
1.4 If relocation fails due to device issues:
1.5 Notify the user and provide troubleshooting steps.
1.6 *return* false.
1.7 Notify the user upon successful recalibration.
1.8 *return* true.

Device Algorithms

Algorithm 1 - addDevice()

Input: Serial Number
Output: boolean, true if successful, false if failure

1.1 Validate serial number
1.2 **If** not a valid serial number:
1.3 *return* False
1.4 **else**:
1.5 Instantiate new device object
1.6 Append new device to User.devices hashmap

Algorithm 2 - deleteDevice()

Input: Serial Number
Output: boolean, true if successful, false if failure

2.1 Search up device in User.devices hashmap by serial number
2.2 **If** no existing serial number matches:
2.3 *return* false
2.4 **else**:
2.5 Remove device from User.devices
2.6 *return* true

Algorithm 3 - turnDeviceOn()

Input: Serial Number
Output: boolean, true if successful, false if failure

3.1 Search up device in User.devices hashmap by serial number
3.2 **If** no existing serial number matches:
3.3 *return* false
3.4 **else**:
3.5 Send API call to turn on device
3.6 Set device.status to true
3.7 *return* true

Algorithm 4 - turnDeviceOff()

Input: Serial Number
Output: boolean, true if successful, false if failure

4.1 Search up device in User.devices hashmap by serial number
4.2 **If** no existing serial number matches:
4.3 *return* false
4.4 **else**:
4.5 Send API call to turn off device
4.6 Set device.status to false
4.7 *return* true

Algorithm 5 - updateDevice()

Input: Serial Number
Output: boolean, true if successful, false if failure

5.1 Search up device in User.devices hashmap by serial number
5.2 **If** no existing serial number matches:
5.3 *Return* false
5.4 **Else**:
5.5 **If** device.updateAvailable is false:
5.6 *Return* false
5.7 **else**:
5.8 Send API call to update device
5.9 Set device.lastUpdate to current time
5.10 *Return* true

Algorithm 6 - setDeviceSchedule()

Input: Serial Number, Days of the Week, Start Time, End Time
Output: boolean, true if successful, false if failure

6.1 Search up device in User.devices hashmap by serial number
6.2 **If** no existing serial number matches:
6.3 *Return* false

6.4 **Else**:
6.5 Instantiate new Schedule with Days of the Week, Start Time, and End Time
6.6 Add schedule to device.schedule
6.7 *Return* true

Alerts Algorithms

Algorithm 1 - The alertRebate() Function

Input: rebate, user
Output: boolean, true if notification sent, false otherwise

1.1 Initialize alert_sent ← false
1.2 foreach rebate in rebateInfo do:
/* 1. Check rebate validity */
1.3 **if** rebate.isInvalid **then**:
1.4 continue;
/*Send Alert*/
1.5 alert ← *Alert*(alertId: rebate.ID, message: rebate.description, type: ENERGY_REBATE,
1.6 severity: 2)
1.7 **if** user.preferences.sendAlert(alert) is true **then**:
1.8 alertSent ← true

1.9 **if** alertSent is false **then**:
1.10 Log("No matching rebates or notification failed.")

1.11 *return* alertSent;

Algorithm 2 - the alertHighUsage() method

Input: device
Output: boolean, true if alert sent, false if not sent

2.1 Initialize alertSent ← false
2.2 energyDataLastMonth ← device.calculateTotalCost("lastMonth")
2.3 energyDataCurrent ← device.calculateTotalCost("current")

/* checks to see if current energy is more than last month */
2.3 **if** energyDataCurrent > energyDataLastMonth **then**:
2.4 excess ← energyDataCurrent - energyDataLastMonth
/*Create Alert*/

2.5 alert ← Alert(alertId:device.deviceID, message: “Device … energy is up by: ” + excess,
2.6 type: HIGH_USAGE, severity: 4)
2.7 **if** user.preferences.*sendAlert*(alert) is true **then**:
2.8 alertSent ← true

2.9 **if** alertSent is false **then**:
Log(“Energy levels are normal or notification failed.”)

2.10 *return* alertSent;

Algorithm 3 - the alertVariabilityPrice()

Input: pricingUtility, deviceData, user
Output: boolean, true if notification sent, false if not

3.1 Initialize latestPrice ← pricingUtility.currentPrice
3.2 Initialize currentPrice ← deviceData.cost

/*check to see if the latest is more than the current price*/
3.2 **if** latestPrice ≥ currentPrice **then**:
3.3 priceDiff ← latestPrice - currentPrice

/*Generate alert*/
3.4 alert ← *Alert*(alertId: deviceData.cost, message: “Electricity prices are above threshold
3.5 by: ” + priceDiff, type: HIGH_USAGE, severity: 2)
3.6 **if** user.preferences.*sendAlert*(alert) is true **then**:
3.7 *return* true;

3.8 *return* false;

Algorithm 4 - the alertDeviceMaintenanceIssue() function

Input: device, user
Output: boolean, true if notification sent, false if not

/** Checks to see device part health */
4.1 **if** device.part.usageData == device.part.usageLimit - 10% **then**:
4.2 alert ← *Alert*(alertId: devic.deviceId, message: “The %s of device %s is approaching its
usage limit. Due for servicing.”, type:APPLIANCE_PROBLEM, severity: 3)
4.3

4.4 **if** user.preferences.sendAlert(alert) is true **then**:
4.5 *return* true;

4.6 *return* false;

Algorithm 5: the handleAlertFailure() function

Input: alert, user
Output; boolean, true if notification eventually sent, false if not

5.1 initialize retries ← 0
5.2 initialize maxRetries ← 3
5.3 **while** retries < maxRetries **do**:
/** Try to send the alert*/
5.4 **if** user.preferences.*sendAlert*(alert) is successful **then**:
5.5 *return* true;
/** keep on trying in intervals **/
5.6 Wait(10 minutes)
5.7 retries ← retries++
5.8 Log("Alert could not be sent. Attempt to resend will be made once reachable.")
5.9 *return* false

Optimization Algorithms

Algorithm 1: The lowPowerModeCheckStatus() method

Input: User settings S, Device List D, Energy Grid Status G
Output: Switch devices to low power mode if the function is enabled

1.1 Toggle LowPowerMode in S
1.2 **if** *isAuthenticated*(user) and LowPowerMode in S is true
1.3 **if** G indicates a grid strain
1.4 **for** each device d in D
1.5 **if** device d supports low power mode setting
1.6 d.*activateLowPowerMode*
1.7 **else**
1.8 *notifyUser*("device does not support low power mode")
1.9 **else** // no grid strain
1.10 **if** LowPowerMode is True

```
1.11                        for each device d in D
1.12                            d.dectivateLowPowerMode()
1.13                        notiftyUser("Grid is Stable")
1.14    else
1.15        if G indicates a grid strain
1.16            reccomendLowPowerModeToUser()
1.17    return
```

Algorithm 2: The monthlyEnergyGoals() method

Input: User Energy Data U, User Feedback F, Device List D
Output: Notifies user of monthly energy consumption goal and recommendations

```
2.1     if isAuthenticated(user) and userHasSufficientData(U) then
2.2         goal ← calculateMonthlyGoal(U) through openAI API
2.3         displayGoal(goal)
2.4         recommendations ← createReccomendations(U, goal, D) using openAI API
2.5         displayReccomendations(Recommendations)
2.6         while userHasFeedback()
2.7             if deemedUnnatainable(F)
2.8                 goal ← recalculateGoal(U, F)
2.9         if userIsApproachingGoal(U, goal)
2.10            notifyUser("You are approaching your monthly goal)
2.11            return
2.12        if goalMet(U, Goal)
2.13            rewardUser()
2.14            Increment user.goalsReached
2.15            notifyUser("You have met this months goal! Rewards are Available")
2.16            return
2.17    else
2.18        notifyUser("You have insufficient data for goal calculations")
2.19    return
```

Algorithm 3: The compareToOtherHomeOwners() method

Input: User Data U, User Settings S

Output: Displays differences between average homeowner in area consumption and user consumption
(comparisonMap results)

```
3.1   if dataSharingEnabled(S)
3.2         homeownerData ← requestDataFromProvider()
3.3         if userHasSufficientData() and homeownerDataIsValid(H)
3.4               initialize comparisonMap to store trend differences
3.5               for each data point d in H
3.6                     if user.hasCorrespondingDataPoint()
3.7                           diff = user.correspondingDataPoint() - d
3.8                           comparisonMap.add(diff)
3.9               comp ← generateComparisonGraph(comparisonMap)
3.10              displayResults(comp)
3.11  else
3.12        notifyUser("There is not sufficient data to generate comparisons")
3.13  return
```

Algorithm 4: The generateRecommendations() method

Input: DeviceList D, User data U
Output: Recommendations List

```
4.1   if userHasSufficentData(U)
4.2         prompt ← "User has energy usage …"
4.3         for each device d in D
4.4               usage ← d.getUsage()
4.5               prompt += usage
4.6         prompt += "provide recommendations in (SEASON) season using this data"
4.7         result ← call openAI API with prompt to gather recommendations
4.8         if result
4.9               recommendations ← parseResult(result)
4.10              displayReccomendations(recommendations)
4.11              Log("Recommendations Successfully parsed")
4.12              return recommendations
4.13        else
4.14              Log("Error Producing Recommendations")
4.15  else
4.16        notifyUser("You do not have sufficient data to generate recommendations")
4.17        return null
```

Algorithm 5: the rewardUser() method

Input: Energy Goal G, User Data U, Rewards Database R, reward options O
Output: No output (Rewards Database is updated)

```
5.1   rewardTier ← findTier(G, U)
5.2   reward ← NULL
5.3   rewardsList ← initialized empty list
5.4   for reward option o in O
5.5         if o.tier <= rewardTier
5.6               rewardList.append(o)
5.7   reward ← choose random option from rewardList
5.8   if reward != NULL
5.9         Add reward to rewards database
5.10        updateRewardsView()
5.11        Log("new reward has been granted")
5.12  else
5.13        Log("Error generating reward for user")
5.14  return
```

Data Algorithms

Algorithm 1: The weeklyEnergyConsumption() function

Input: devices (dictionary), current week (list of dates)
Output: boolean

```
1.1   Initialize totalConsumption ← 0
1.2   For through each deviceID, deviceData in devices:
1.3         For each day in current week:
1.4               If device contains data for the day:
1.5                     Add device[day] to totalConsumption
1.6   Display devices for each day and totalConsumption to the user for each day this week
1.7   Return true
```

Algorithm 2: deviceRanking() function

Input: devices (dictionary), current month (list of dates)
Output: boolean

```
2.1     Initialize monthlyConsumption ← empty hashmap
2.2     Initialize deviceRanking ← list
2.2     For through each deviceID, deviceData in devices:
2.3             Initialize monthlyUsage ← 0
2.4             For each day in currentMonth:
2.5                     If day exists in deviceData["usage"]:
2.6                             If device has higher usage than previously iterated one:
2.7                                     Rank higher than previous in deviceRanking
2.8                             else
2.9                                     Rank lower than previous in deviceRanking
2.6                             Add deviceData["usage"][day] to monthlyUsage
2.7             Set monthlyConsumption[deviceID] ← monthlyUsage
2.8     Sort monthlyConsumption by values in descending order
2.9     Display sorted monthlyConsumption device rankings to the user
2.10    Return true
```

Algorithm 4: costPatternOverTime() function

Input: devices (dictionary), costFactor (int), timePeriod (list of dates)
Output: boolean

```
4.1     Initialize costOverTime ← list
4.2     For each day in timePeriod:
4.3             Initialize dailyCost ← 0
4.4             For each deviceID, deviceData in devices:
4.5                     If day exists in deviceData["usage"]:
4.6                             Add deviceData["usage"][day] * costFactor to dailyCost
4.7             Append (day, dailyCost) to costOverTime
4.8     Display costOverTime as a line chart
4.9     Return true
```

Algorithm 5: carbonFootprint() function

Input: devices (dictionary), carbonFootprintFactor (constant int)

Output: boolean

5.1 Initialize carbonFootprint ← 0
5.2 **For** each deviceID, deviceData in devices:
5.3 **For** each day in deviceData["usage"]:
5.4 Add deviceData["usage"][day] * carbonFootprintFactor
5.5 Display carbonFootprint to the user
5.6 *Return* true