Team X

DashX
Design Report For
E-restaurant App

Version 1.0

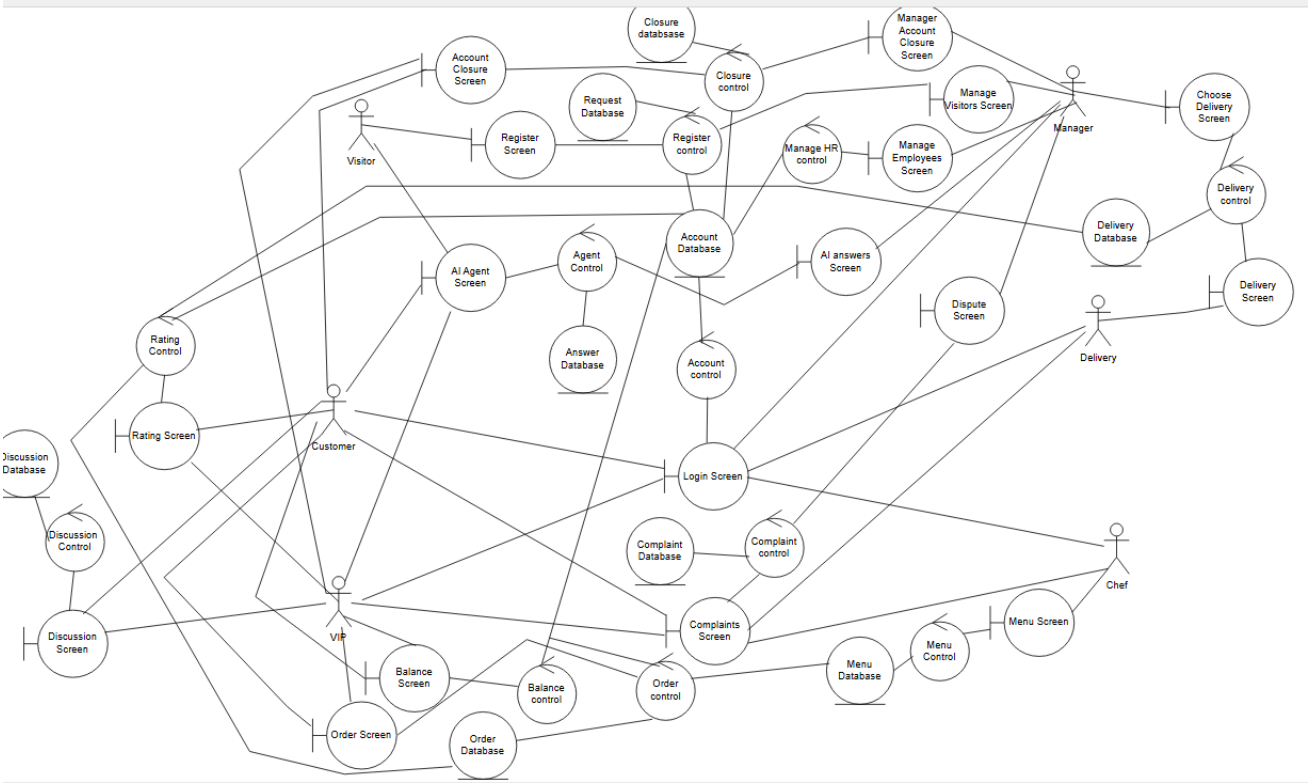
Revision History

Date	Version	Description	Author
11/04/2025	1.0	Compose the second report that presents the low-level design of our E-Restaurant App called DashX. It includes detailed collaboration class diagrams, Petri-net diagrams, and E/R diagrams for all use cases in Report 1, covering both normal and exceptional scenarios. In addition, the report provides the design and description of the major GUI screen that will be used in our application.	Bryan Dong, Yuki Li, Joseph Helfenbein, Jacob Li

Table of Contents

Design Report

1. Introduction



This is a general collaboration class diagram describing the connections between different functionality of our app. It goes into less detail about how the flow of action goes for some functionality for the purpose of making the diagram understandable. In part 2 of this report, collaboration class diagrams will be given for each use case, properly demonstrating the steps that happen when a user uses that functionality.

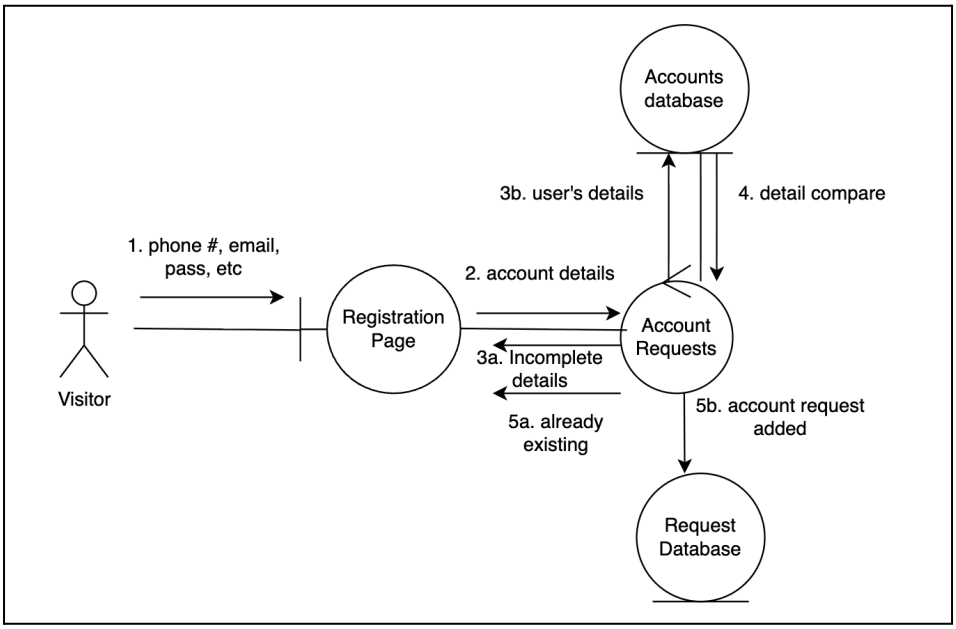
2. Collaboration Class Diagram and Petri-Net Diagram

1. **User Registration**

   Normal: Visitor enters details, they are valid for an account, and they a request is stored for the manager to see.

   Exceptional: Visitor enters details, they are not complete so an error message shows up

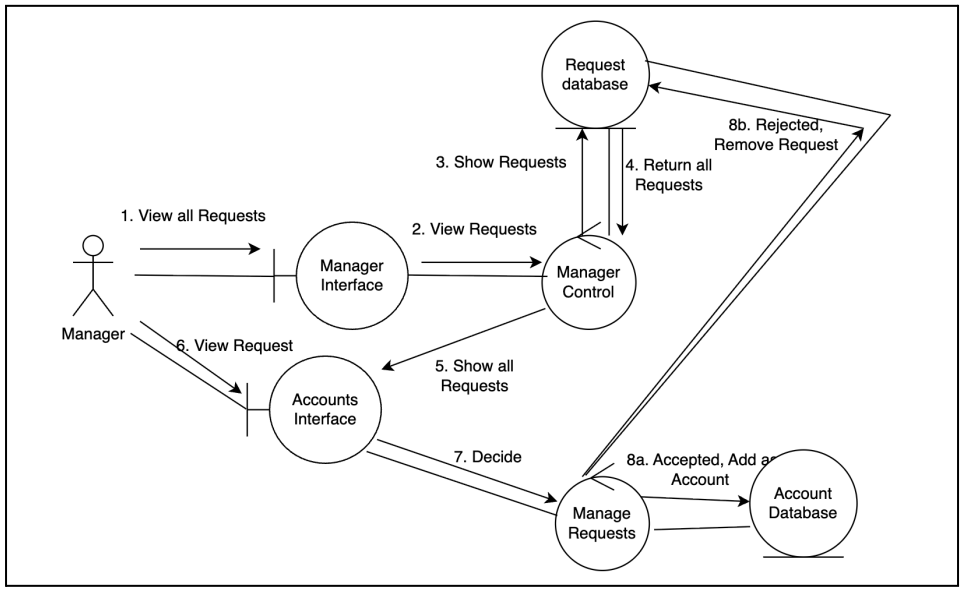
   Visitor enters details, they refer to an already existing account so an error message shows up



2. **Process a Registration**

   Normal: Manager selects a request, and decides to accept or reject the request. This updates the account for the requester.

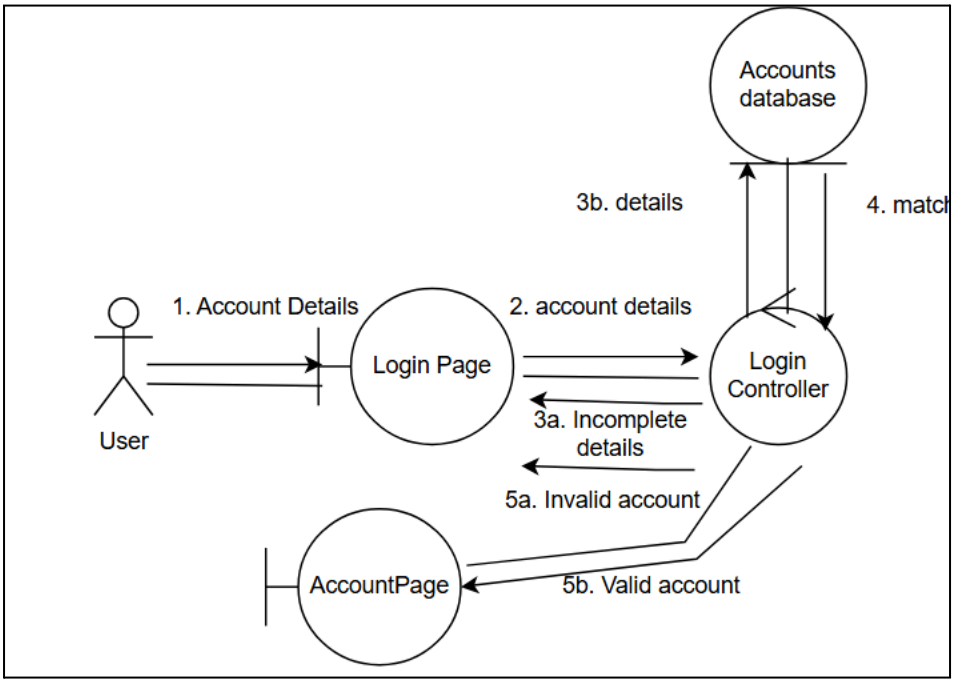
   Exceptional: Database corrupted and incomplete data is now held, so an error must occur.



3. **Login**

   Normal: User is on the login page, enters their details and submits the form logging them in.

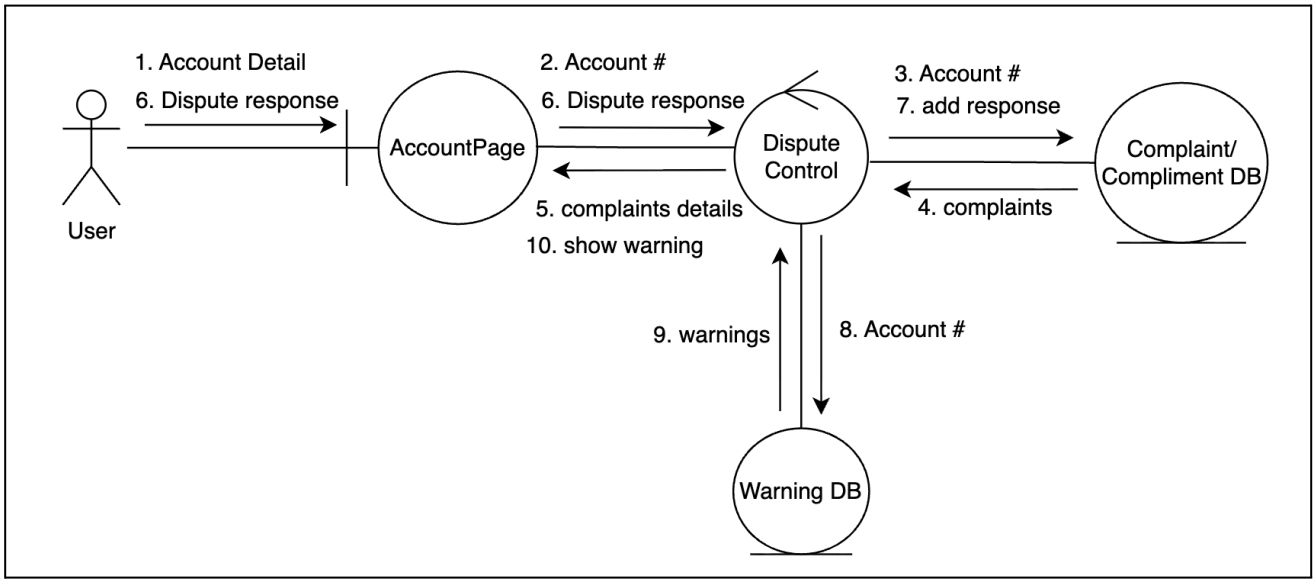
Exceptional: The entered details are incomplete or invalid, granting an error message.



4. **Dispute Complaint**

   Normal: User sees their account has a complaint against it, they want to dispute that complaint so they write an explanation and submit the dispute for the manager to review later.

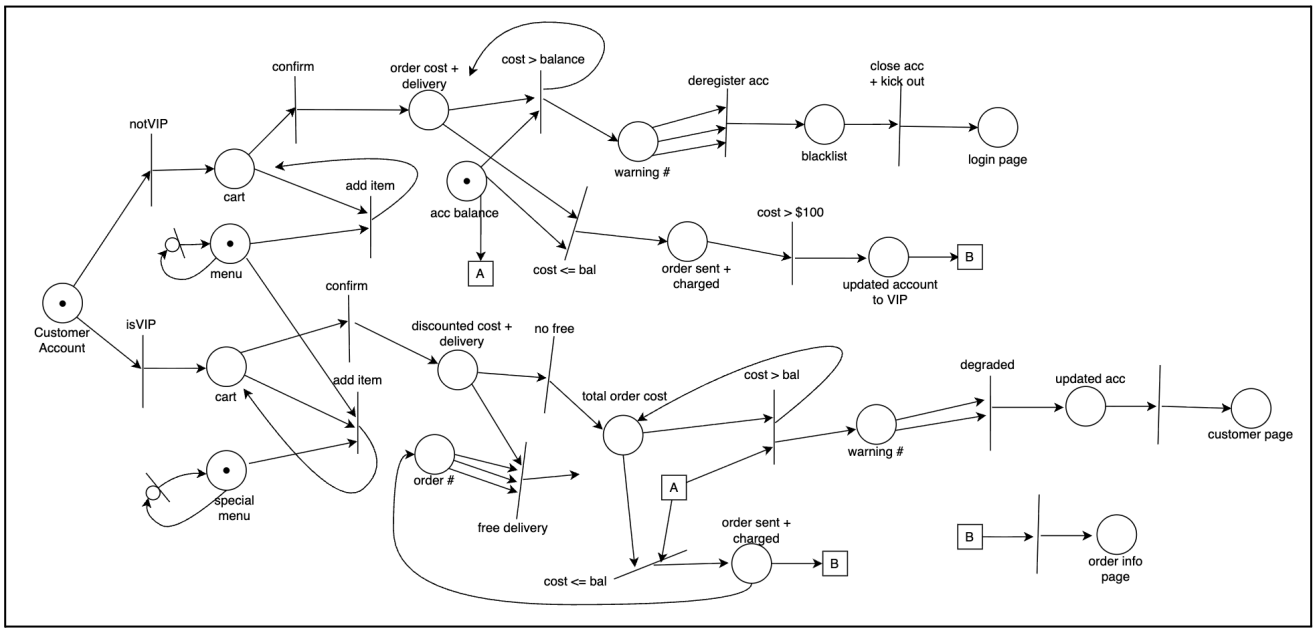
   Exceptional: The user does not enter any information about the complaint/reason for disputing leading to error message.



5. **Place Order**

   Normal: User picks items to order from the menu, they get added to the order, user pays using their balance without problems, and their trustworthiness is increased.

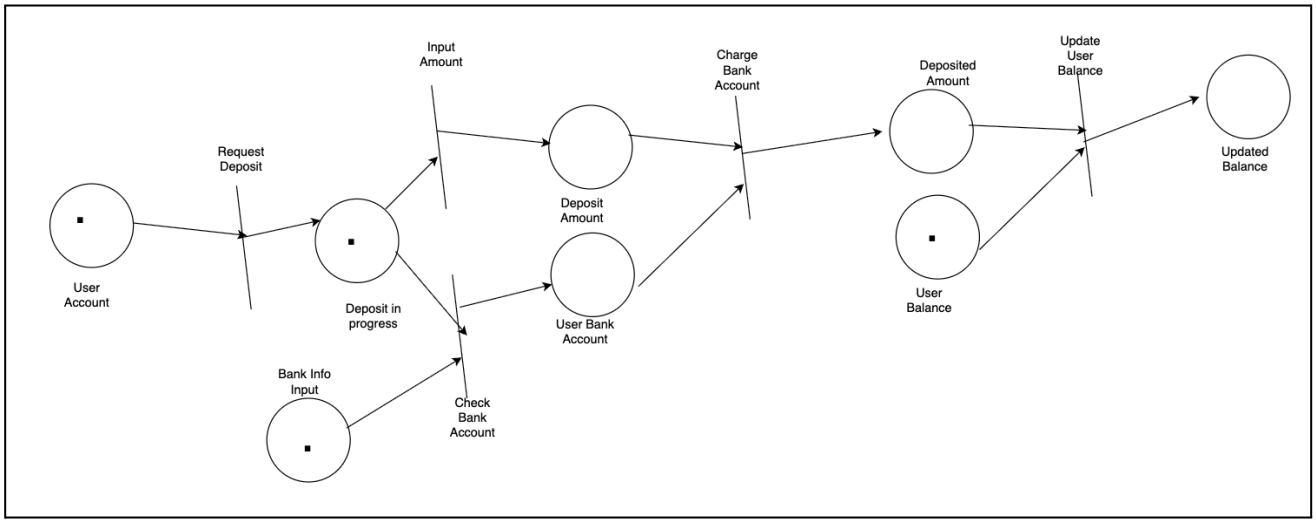
   Exceptional: The user does not have enough money in their account balance to pay for the items they ordered, they will get an error message and a warning.



6. **Deposit Money**

   Normal: The user wants to add money to their account, they input in their bank account details and the amount they would like to deposit into their account. The transaction goes through and their balance is updated.

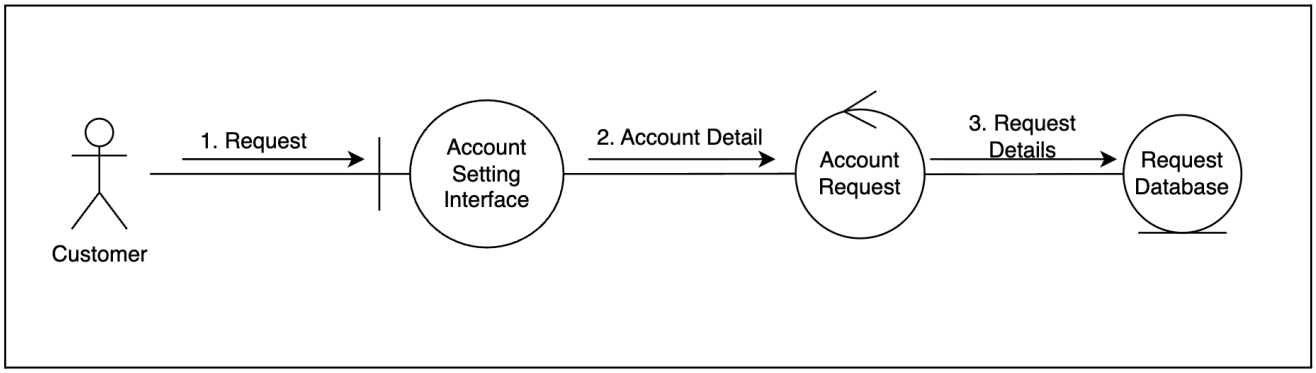
   Exceptional: The user submits invalid bank account information. This causes an error message. (Overdrafting isn't an error, and is something a user could do)



7. **Request Account Closure**

   Normal: The user wants to close their account for some reason. They request a closure and specify the reason and then submit that to be stored for the manager to access the request.

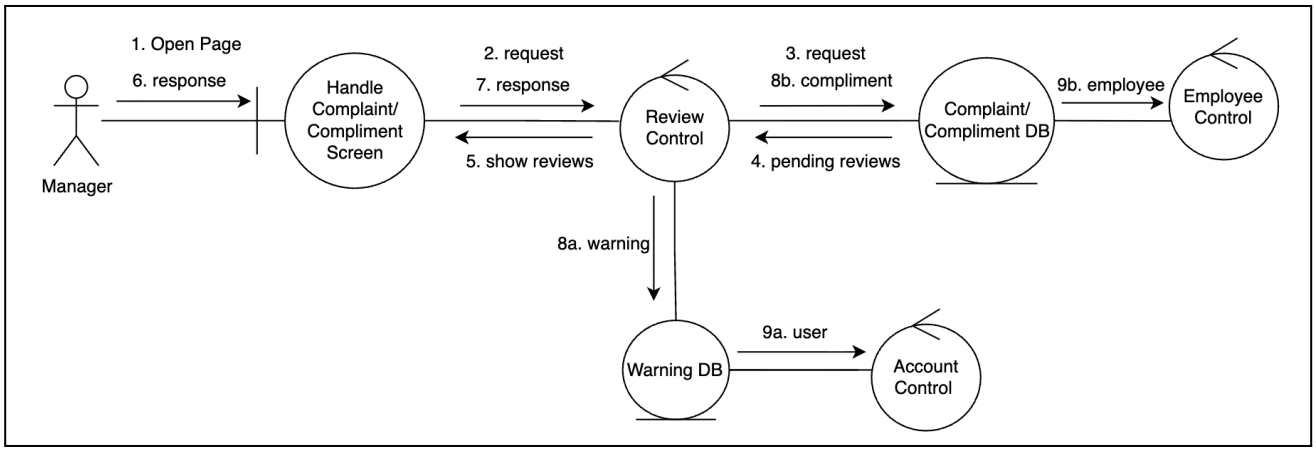
   Exceptional: The user does not submit information about the purpose for closure. An error message shows up.



8. **Handle Complaints / Compliments**

   Normal: The manager takes a look at the requests to dispute a complaint. The manager looks at a dispute request with a reason and decides to accept or deny the dispute request. Warnings are updated for the complainer and the complainee depending on the result of the request.

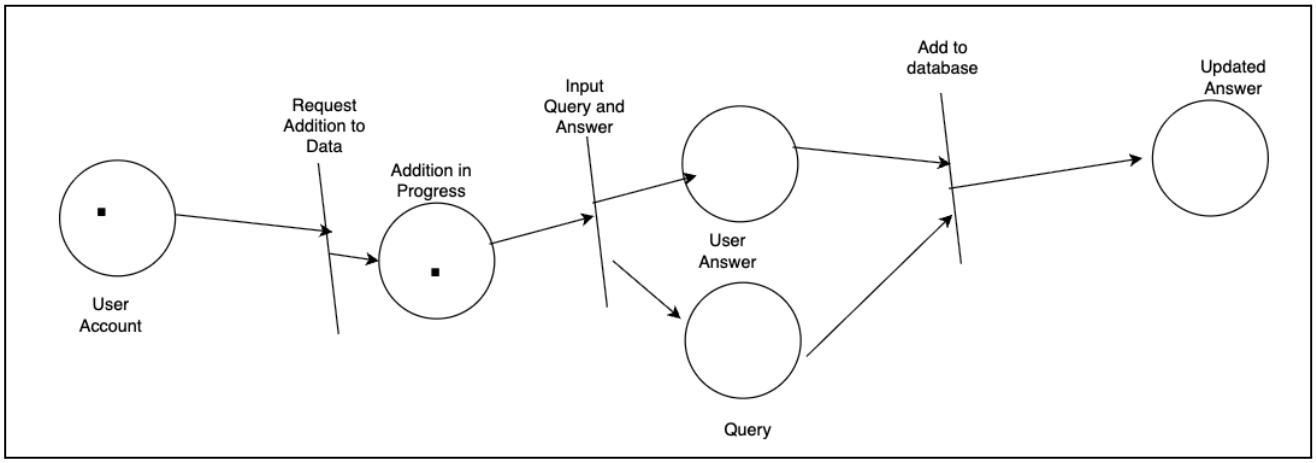
   Exceptional: The database lost data concerning the request leading to the request not being possible to analyze causing an error.



9. **Add Local Information to Service Model**

   Normal: The user decides to add information to the agent, with them choosing the query, and adding the answer to that query. This is saved in the database.

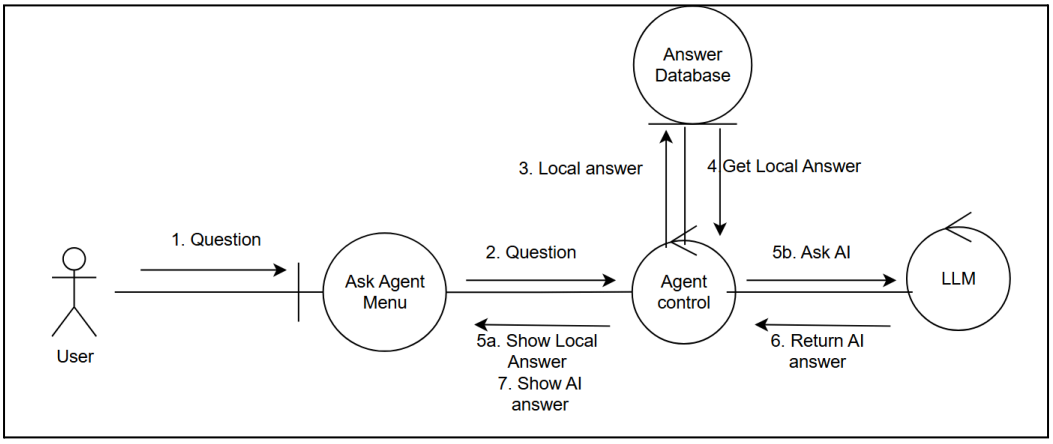
   Exceptional: The database is unavailable and so the user addition cannot be saved currently leading to an error message.



10. **Ask Customer Service Information**

    Normal: The user decides to ask the agent for an answer to a question. If the question has a local answer, that is presented to the user, and the answer is asked to rate the answer. If not, the LLM generates an answer for the user to see.

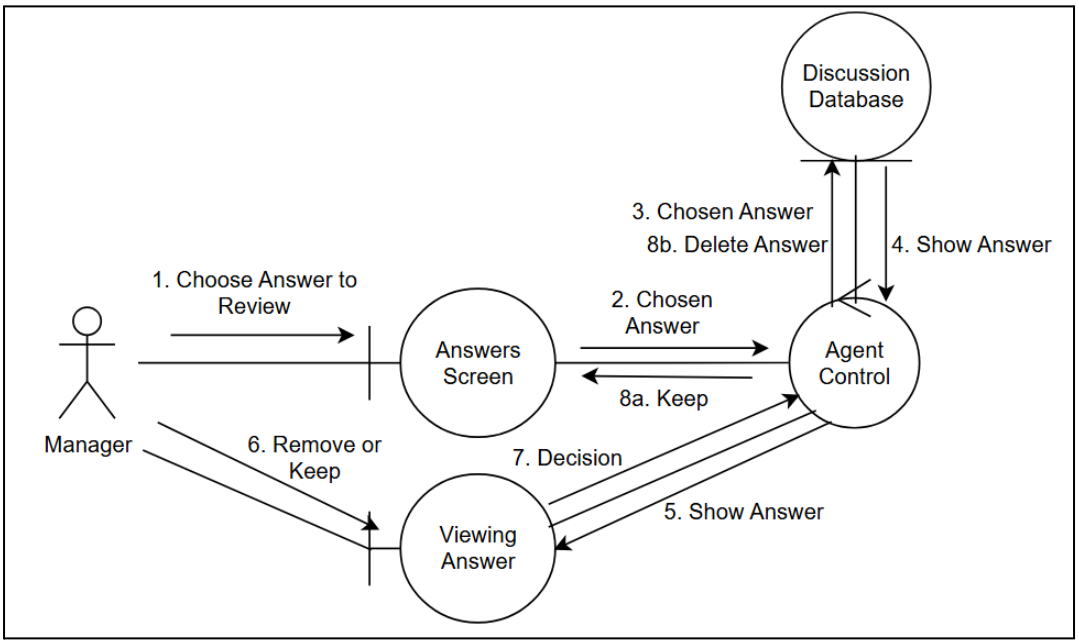
    Exceptional: The AI agent is not available and cannot be queried currently, resulting in an error message.



11. **Review Local Knowledge Answer**

    Normal: The manager takes a look at a low rated answer. They decide to keep the answer, or they decide to remove the answer, and remove the query as well if that was the only answer for that query.

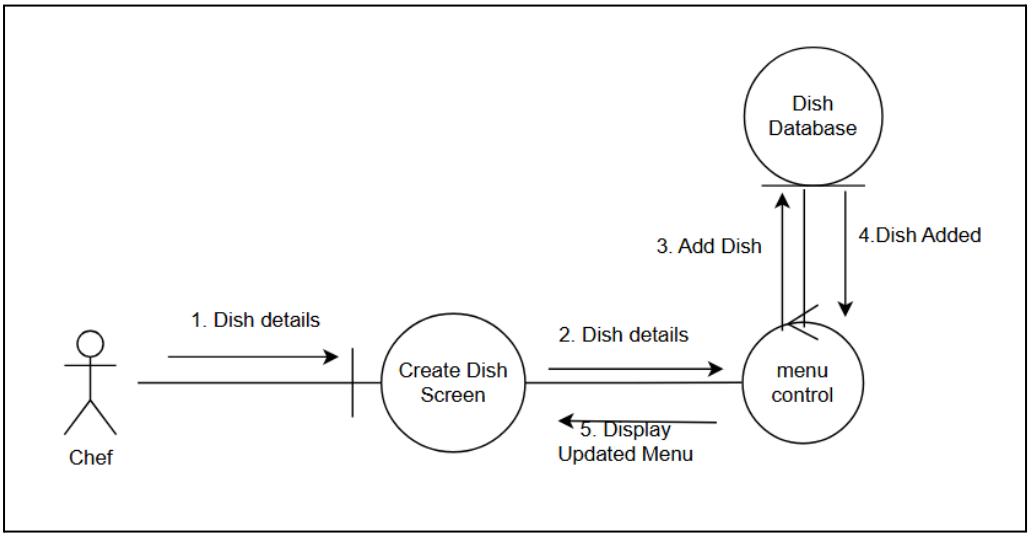
    Exceptional: The answer and question database is not available and cannot be accessed and updated currently, resulting in an error message.



12. **Create / Update Menu**

    Normal: The chef decides to add or modify a dish to the menu so they input the details like the name, description, details, picture, and price. This finished dish is saved to the database.

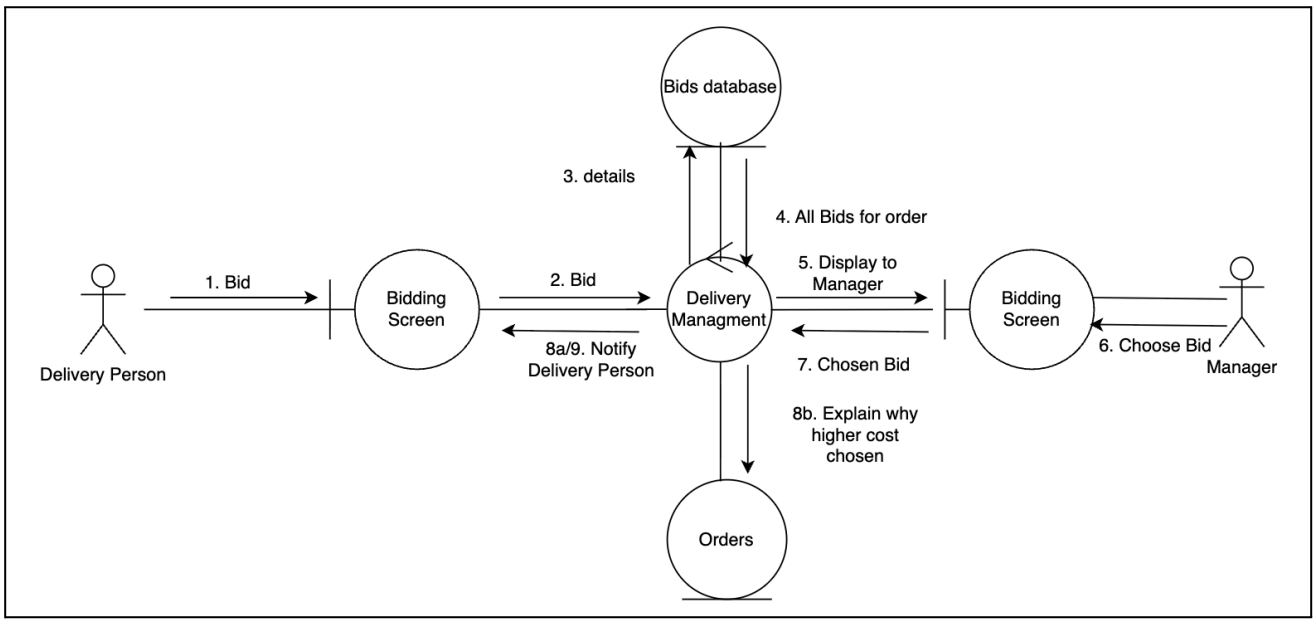
    Exceptional: The chef did not fill out the details of the dish like not giving a price before trying to create it. This leads to an error message and the dish is not saved.



13. **Assign Delivery**

Normal: An order is available to be delivered, delivery people bid on the order to deliver it. The manager successfully picks one of the bids to act as the delivery person. If it is not the lowest, a note is added by the manager about the reason.

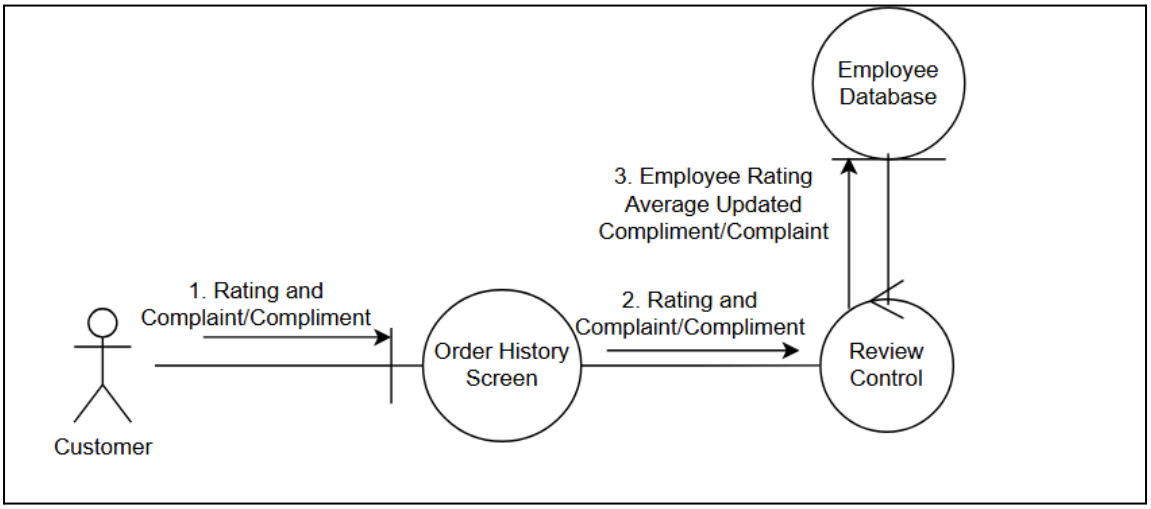
Exceptional: A delivery person bids for an order at the same time as the order is closed because the manager chose a bid already, this leads to an error message.



14. **Rate Food / Delivery**

Normal: The customer ordered a dish with delivery at some point in the past. Now they want to rate one of those past experiences. They select the delivery person or dish, then they rate it and give a complaint/compliment if they feel it is necessary. This is successfully saved to the database.

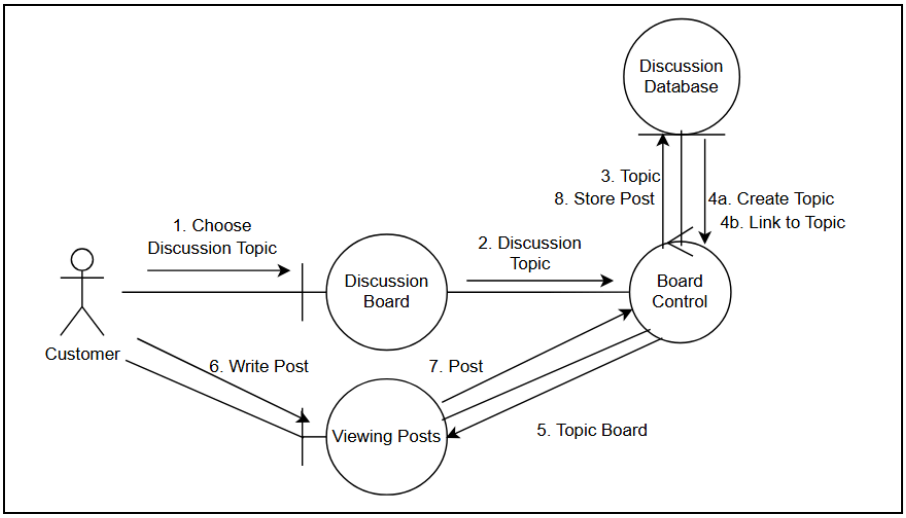
Exceptional: The dish or the delivery person was removed/fired, meaning the thing the customer is trying to rate is not stored in the database. This leads to an error message.



15. **Start / Join Discussion**

    Normal: The customer decides to talk with their fellow customers on a forum. They write the chosen item/employee/relevant topic as the subject of the thread they would like to discuss. If that thread is already available the customer can talk in that thread. Otherwise they can create a new thread and post in it.

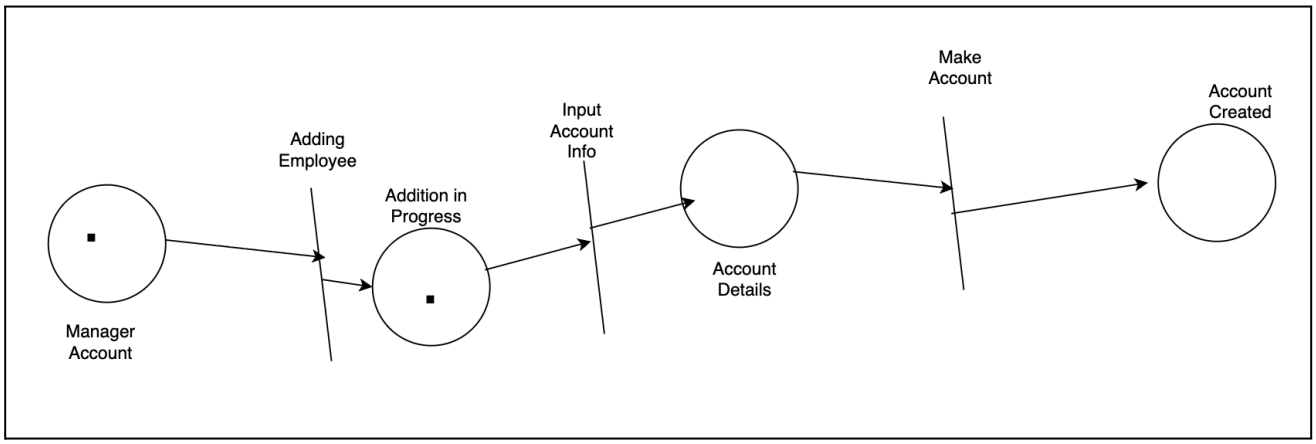
    Exceptional: The forum posts database is not available and cannot be accessed and updated currently, resulting in an error message.



16. **Hire Employee**

    Normal: The manager decides to hire an employee. They know this person in the real world. Now they create an account for them. These account details are now available for the employee to log into.

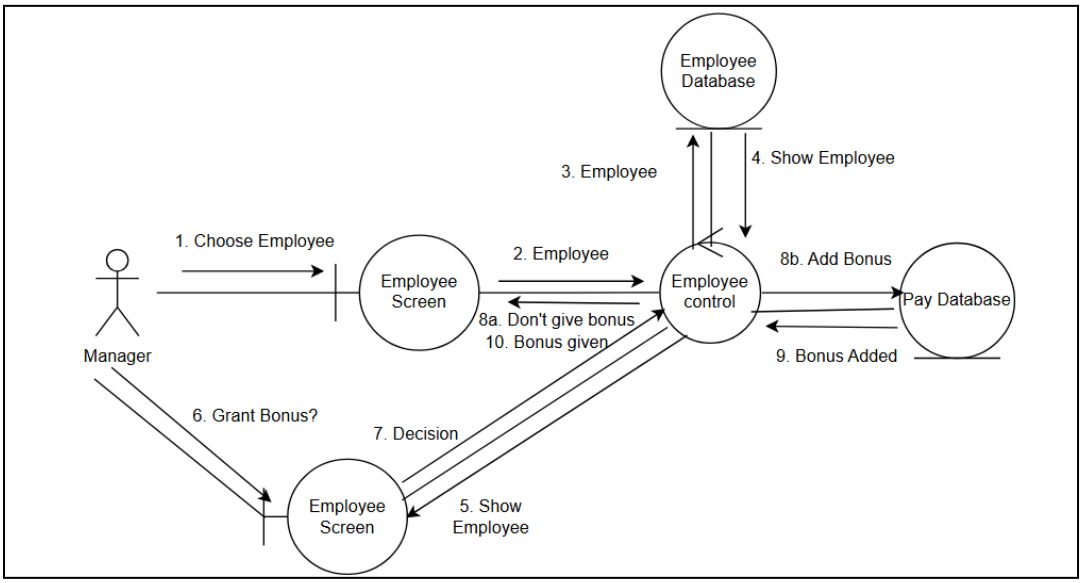
    Exceptional: The manager does not fill in all the details necessary for creating an employee account like the type. This leads to an error message.



17. **Grant Bonus**

Normal: The manager looks at all the employees finding one that meets the criteria for a bonus. The manager enters the bonus amount they would like to reward. This bonus is saved and applied to the employee.

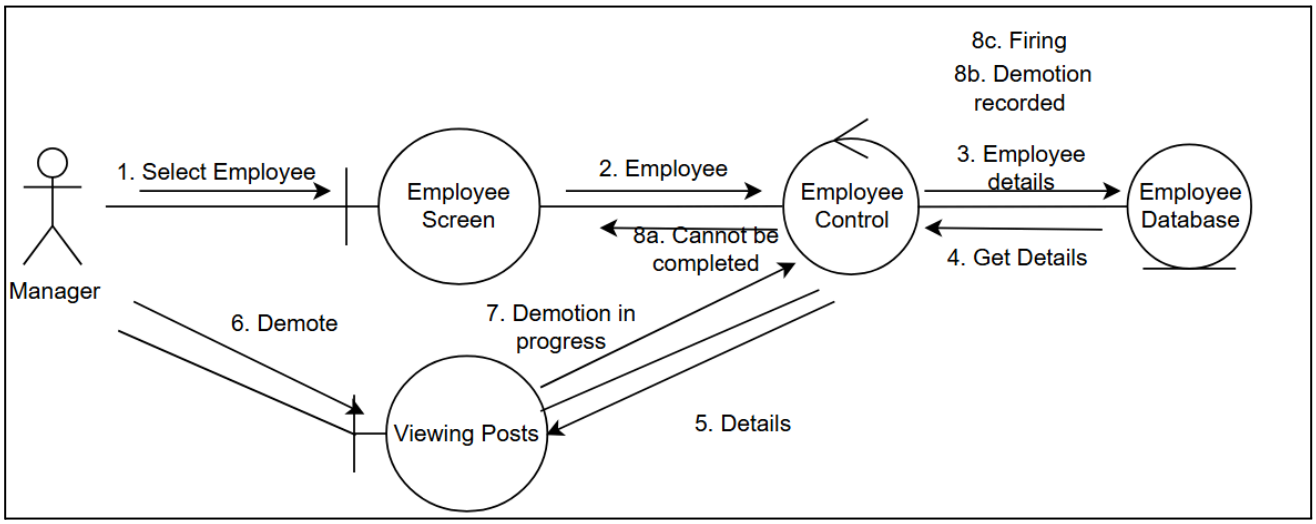
Exceptional: The manager does not fill in all the details necessary for granting a bonus like the dollar amount to give. This leads to an error message.



18. **Demote Employee**

Normal: An employee meets the criteria for demotion. The manager decides to demote them. If it is their first demotion, their salary is cut and the demotion is recorded. Otherwise, they are fired due to having been demoted before. This is all saved.

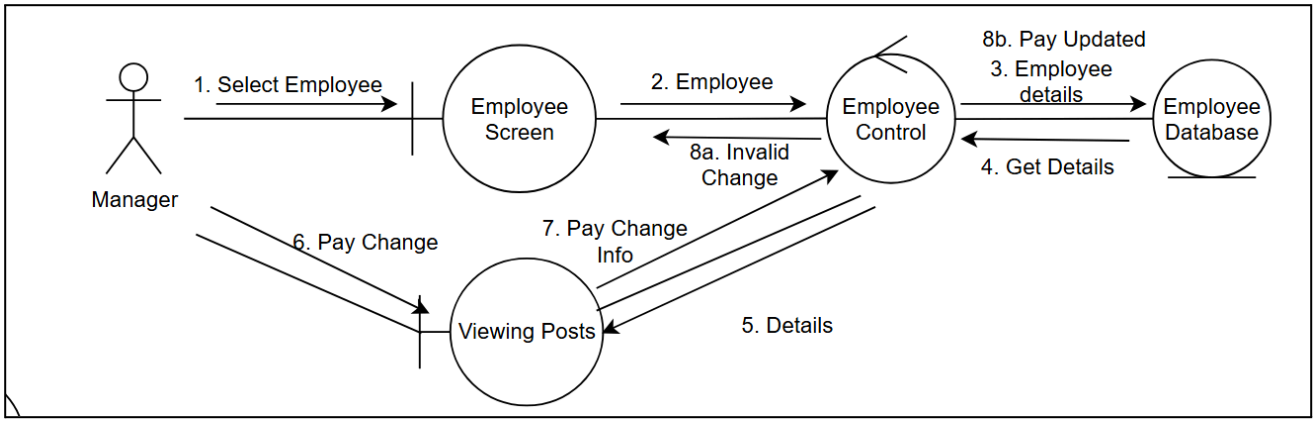
Exceptional: The employee has been demoted before and now is set to be fired because of their 2nd demotion. However, there are not enough employees for the restaurant to continue if this employee is fired, so an error message shows up and the employee is not demoted again for now.



19. Adjust Pay

Normal: The manager wants to change the pay of an employee. They write the updated wage and the system saves it to the employee database.

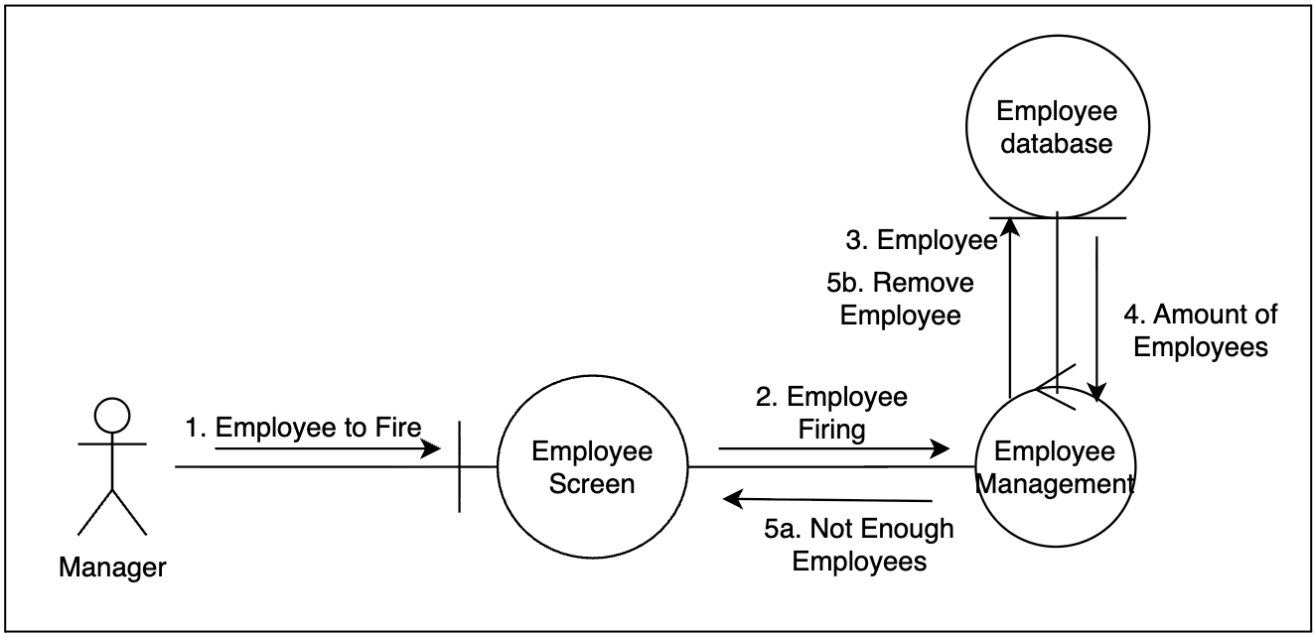
Exceptional: The manager made a decreased wage that is not valid like if it is below the minimum wage. This leads to an error screen and the change does not go through.



20. Terminate/Fire Employee

Normal: An employee meets the criteria for termination. The manager decides to terminate them. The employee is fired and their account is removed. This is all saved in the database.

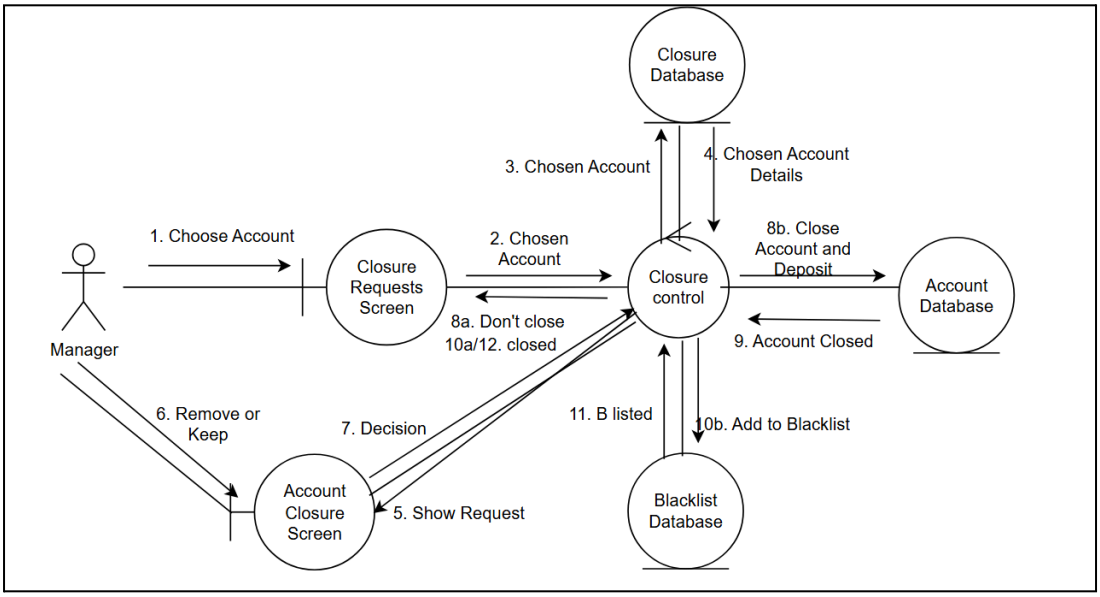
Exceptional: The employee is set to be fired because the manager selected them. However, there are not enough employees for the restaurant to continue if this employee is fired, so an error message shows up and the employee is not fired for now.



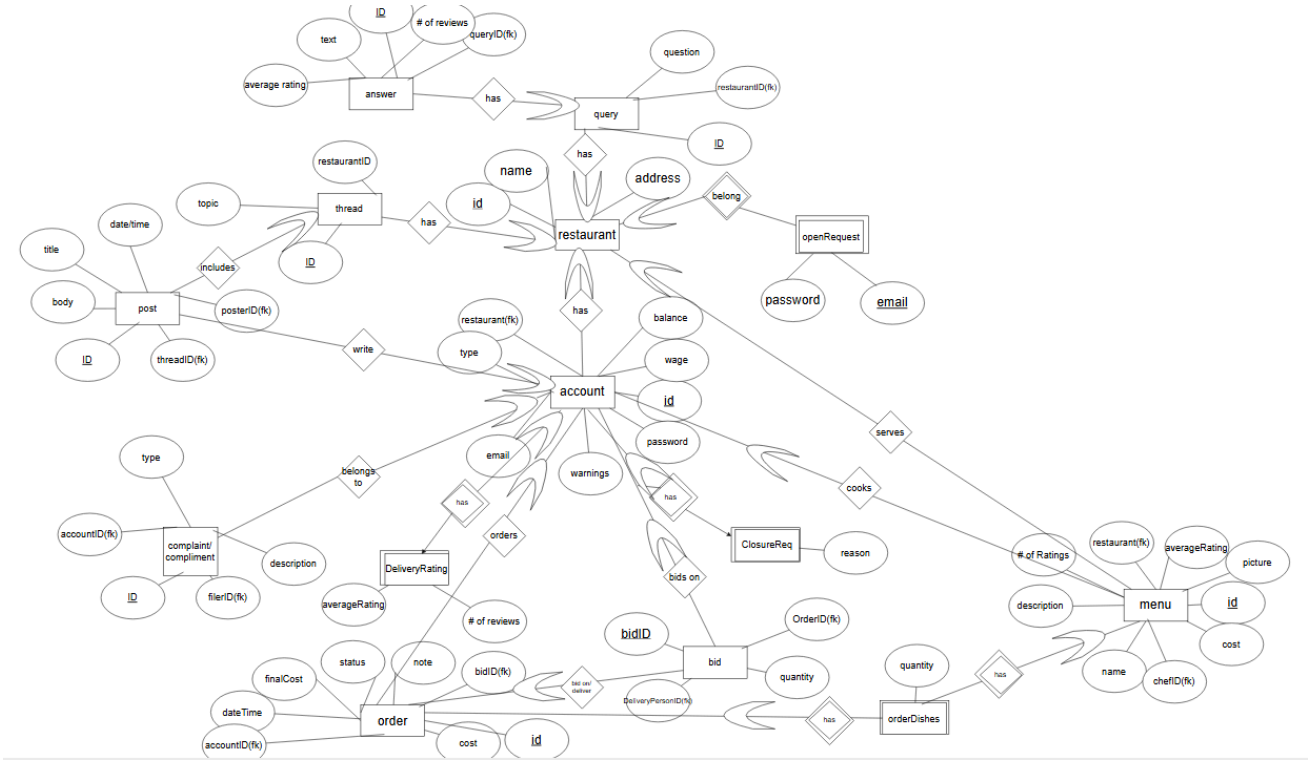
21. **Close Account**

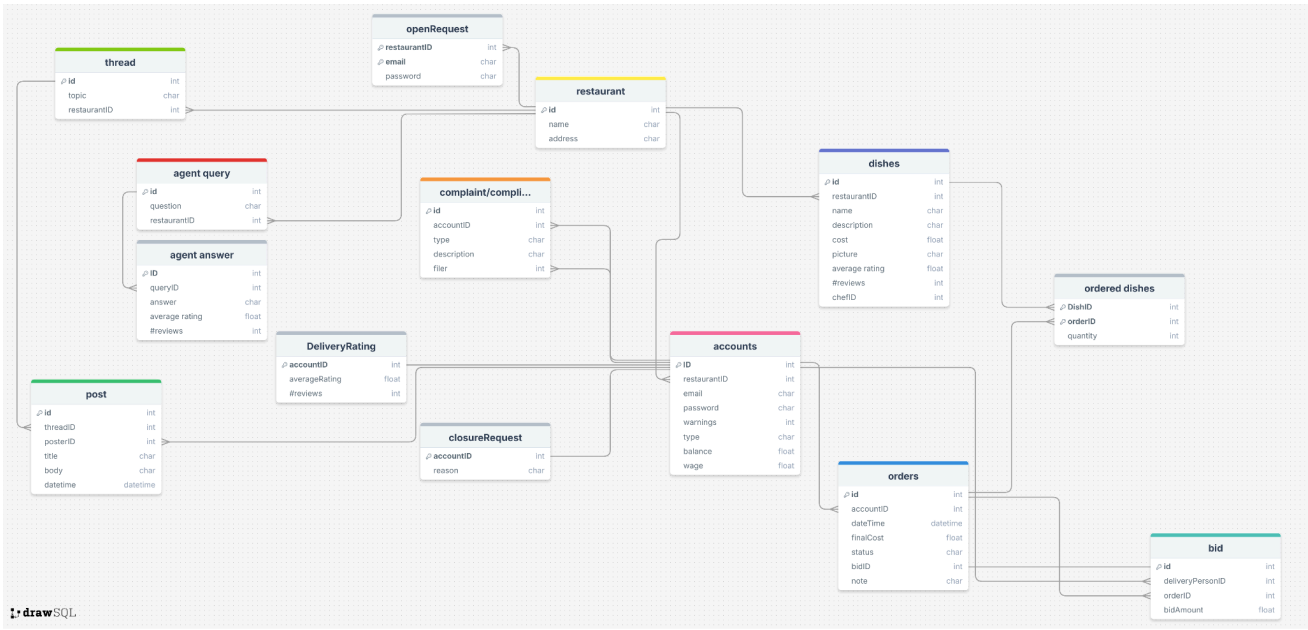
Normal: A customer sent a request to close their account. The manager now takes a look at that request, with the reason for closure and decides whether to close the account. If they accept and close the account this change is updated on the database.

Exceptional: The database is unavailable and so the user addition cannot be saved currently leading to an error message.



3. E-R Diagram for the Entire System





4. Detailed design

1. User Registration

Main → RegistrationPage.collectInput → RegistrationPage.handleSubmit → AccountRequests.processDetails → DB.insert(PendingRequests)

RegistrationPage.collectInput

Input: none (reads UI fields)

Output: form: UserRegistrationForm

Main:

```
form.phone = trim(UI.getValue("phone"))
form.email = lowerCase(trim(UI.getValue("email")))
form.pass  = UI.getValue("pass")
return form
```

RegistrationPage.handleSubmit

Input: form: UserRegistrationForm

Output: result: "INCOMPLETE" | "ALREADY_EXISTS" | "REQUEST_STORED"

Main:

```
validation = RegistrationPage.validateForm(form)
if validation.isValid == false: return "INCOMPLETE"
return AccountRequests.processDetails(form)
```

RegistrationPage.validateForm

Input: form

Output: ValidationResult { isValid: bool, errors: map }

Main:

```
result.isValid = true
if form.email == "": result.isValid = false; result.errors["email"] = "required"
// enforce password of length >= 8
if form.pass == "" or length(form.pass) < 8: r.isValid=false; r.errors["pass"]="required/min 8”
return result
```

AccountRequests.processDetails

Input: form: UserRegistrationForm

Output: "INCOMPLETE" | "ALREADY_EXISTS" | "REQUEST_STORED"

Main:

```
if not AccountRequests.isComplete(form): return "INCOMPLETE"
// check Accounts database for duplicate
existingUser = AccountsDatabase.lookupUserByEmail(form.email)
if existingUser != null: return "ALREADY_EXISTS"
// build request and persist to Request Database
req = AccountRequests.buildRegistrationRequest(form)
DB.insert("PendingRequests", req)  // stores to PendingRequests function assumed
return "REQUEST_STORED"
```

AccountRequests.isComplete

Input: form

Output: bool

Main:

```
if form.email == "": return false
if trim(form.pass)  == "": return false
```

return true

AccountRequests.buildRegistrationRequest

Input: form

Output: RegistrationRequest

Main:

hashed = Security.hashPassword(form.pass) // hash function assumed

req.id = UUID() // UUID function assumed

req.email = form.email

req.phone = form.phone

req.hashedPass = hashed

req.status = "PENDING"

req.createdAt = now()

return req

AccountsDatabase.lookupUserByEmail

Input: email: string

Output: UserRecord | null

Main:

user = DB.findOne("Users", { email: email }) // db lookup function assumed

return user or null

2. **Process a Registration**

Manager → ManagerInterface.viewAllRequests → ManagerControl.getAllRequests → RequestDatabase.getAllPending → ManagerControl → ManagerInterface.showAllRequests → Manager → AccountsInterface.viewRequest → AccountsInterface.decide → ManageRequests.processDecision → (ACCEPT → AccountsDatabase.saveUser + RequestDatabase.deleteById) | (REJECT → RequestDatabase.deleteById)

ManagerInterface.viewAllRequests

Input: none

Output: requests[] | "ERROR_DB_CORRUPT"

Main:

result = ManagerControl.getAllRequests()

if result == "ERROR_DB_CORRUPT": return "ERROR_DB_CORRUPT"

return result

ManagerControl.getAllRequests

Input: none

Output: requests[] | "ERROR_DB_CORRUPT"

Main:

try:

requests = RequestDatabase.getAllPending()

return requests

catch DBError:

return "ERROR_DB_CORRUPT"

RequestDatabase.getAllPending

Input: none

Output: requests[] | throws DBError

Main:
  return DB.findAll("PendingRequests", { status: "PENDING" }) // db function assumed

ManagerInterface.showAllRequests

Input: requests[]

Output: void

Main:
  UI.renderList("requests", requests)

AccountsInterface.viewRequest

Input: requestId: string

Output: RegistrationRequest | "NOT_FOUND" | "ERROR_DB_CORRUPT"

Main:
  try:
    req = DB.find("PendingRequests", { id: requestId }) // db function assumed
  catch DBError:
    return "ERROR_DB_CORRUPT"
  if req == null: return "NOT_FOUND"
  return req

AccountsInterface.decide

Input: requestId: string, decision: "ACCEPT" | "REJECT"

Output: "OK" | "NOT_FOUND" | "ERROR_DB_CORRUPT" | "ERROR_INCOMPLETE_DATA"

Main:
  return ManageRequests.processDecision(requestId, decision)

ManageRequests.processDecision

Input: requestId: string, decision: "ACCEPT" | "REJECT"

Output: "OK" | "NOT_FOUND" | "ERROR_DB_CORRUPT" | "ERROR_INCOMPLETE_DATA" | "ERROR_DUPLICATE_USER"

Main:
  try:
    req = DB.find("PendingRequests", { id: requestId })
  catch DBError:
    return "ERROR_DB_CORRUPT"
  if req == null: return "NOT_FOUND"
  if not ManageRequests.isRequestDataComplete(req): return "ERROR_INCOMPLETE_DATA"
  email = lowerCase(trim(req.email))
  if decision == "ACCEPT":
    try:
      DB.beginTransaction()
      existing = DB.find("Users", { email: email }) // check duplicate in Users
      if existing != null:
        DB.rollback()
        return "ERROR_DUPLICATE_USER"
      user = UserRecord(id = UUID(), email = email, phone = req.phone, hashedPass = req.hashedPass, createdAt = now())
      DB.insert("Users", user)
      DB.delete("PendingRequests", { id: requestId })
      DB.commit()
      return "OK"

```
    catch DBError:
      DB.rollback()
      return "ERROR_DB_CORRUPT"
  else if decision == "REJECT":
    try:
      DB.delete("PendingRequests", { id: requestId })
      return "OK"
    catch DBError:
      return "ERROR_DB_CORRUPT"
```

ManageRequests.isRequestDataComplete

```
Input: req: RegistrationRequest
Output: bool
Main:
  if req.email == null or trim(req.email) == "": return false
  if req.hashedPass == null or req.hashedPass == "": return false
  return true
```

AccountsDatabase.saveUser

```
Input: user: UserRecord
Output: "OK" | throws DBError
Main:
  existing = DB.find("Users", { email: user.email }) // db function assumed
  if existing != null: throw DBError("duplicate")
  DB.insert("Users", user)
  return "OK"
```

3. Login

Main → LoginPage.collectInput → LoginPage.handleSubmit → LoginController.authenticate → AccountsDatabase.lookupUserByEmail → LoginController → (VALID → AccountPage.display) | (INVALID → LoginPage.showError)

LoginPage.collectInput

```
Input: none (reads UI fields)
Output: form: LoginForm
Main:
  form.email = lowerCase(trim(UI.getValue("email")))
  form.pass  = UI.getValue("pass")
  return form
```

LoginPage.handleSubmit

```
Input: form: LoginForm
Output: result: "INCOMPLETE" | "INVALID_ACCOUNT" | "VALID_ACCOUNT"
Main:
  validation = LoginPage.validateForm(form)
  if validation.isValid == false: return "INCOMPLETE"
  return LoginController.authenticate(form)
```

LoginPage.validateForm

```
Input: form: LoginForm
Output: ValidationResult { isValid: bool, errors: map }
```

Main:
  result.isValid = true
  if form.email == "": result.isValid = false; result.errors["email"] = "required"
  if form.pass == "": result.isValid = false; result.errors["pass"] = "required"
  return result

LoginController.authenticate

Input: form: LoginForm
Output: "INCOMPLETE" | "INVALID_ACCOUNT" | "VALID_ACCOUNT"
Main:
  if not LoginController.isComplete(form): return "INCOMPLETE"
  user = AccountsDatabase.lookupUserByEmail(form.email)
  if user == null: return "INVALID_ACCOUNT"
  if not Security.verifyPassword(form.pass, user.hashedPass): return "INVALID_ACCOUNT"
  Session.setCurrentUser(user) // session function assumed
  return "VALID_ACCOUNT"

LoginController.isComplete

Input: form: LoginForm
Output: bool
Main:
  if form.email == "": return false
  if trim(form.pass) == "": return false
  return true

LoginPage.showError

Input: errorType: "INCOMPLETE" | "INVALID_ACCOUNT"
Output: void
Main:
  if errorType == "INCOMPLETE":
    UI.displayError("Please fill in all fields")
  else if errorType == "INVALID_ACCOUNT":
    UI.displayError("Invalid email or password")

AccountPage.display

Input: none (reads session)
Output: void
Main:
  user = Session.getCurrentUser()
  UI.renderPage("account", user)

AccountsDatabase.lookupUserByEmail

Input: email: string
Output: UserRecord | null
Main:
  user = DB.findOne("Users", { email: email }) // db lookup function assumed
  return user or null

4. **Dispute Complaint**

Main → AccountPage.viewComplaints → DisputeControl.getComplaintsForAccount → ComplaintDatabase.getComplaintsByAccount → DisputeControl → AccountPage.showComplaints →

AccountPage.collectDisputeResponse → AccountPage.submitDispute → DisputeControl.processDispute → ComplaintDatabase.addDisputeResponse → DisputeControl.getWarningsForAccount → WarningDatabase.getWarningsByAccount → DisputeControl → AccountPage.showWarning

AccountPage.viewComplaints
Input: accountId: string
Output: complaints[] | "ERROR_DB_CORRUPT"
Main:
  result = DisputeControl.getComplaintsForAccount(accountId)
  if result == "ERROR_DB_CORRUPT": return "ERROR_DB_CORRUPT"
  return result
DisputeControl.getComplaintsForAccount
Input: accountId: string
Output: complaints[] | "ERROR_DB_CORRUPT"
Main:
  try:
    complaints = ComplaintDatabase.getComplaintsByAccount(accountId)
    return complaints
  catch DBError:
    return "ERROR_DB_CORRUPT"
ComplaintDatabase.getComplaintsByAccount
Input: accountId: string
Output: complaints[] | throws DBError
Main:
  return DB.findAll("Complaints", { accountId: accountId }) // db function assumed
AccountPage.showComplaints
Input: complaints[]
Output: void
Main:
  UI.renderList("complaints", complaints)
AccountPage.collectDisputeResponse
Input: complaintId: string (reads UI fields)
Output: disputeForm: DisputeForm
Main:
  disputeForm.complaintId = complaintId
  disputeForm.response = trim(UI.getValue("disputeResponse"))
  return disputeForm
AccountPage.submitDispute
Input: disputeForm: DisputeForm
Output: result: "INCOMPLETE" | "DISPUTE_SUBMITTED"
Main:
  validation = AccountPage.validateDisputeForm(disputeForm)
  if validation.isValid == false: return "INCOMPLETE"
  return DisputeControl.processDispute(disputeForm)
AccountPage.validateDisputeForm
Input: disputeForm: DisputeForm

Output: ValidationResult { isValid: bool, errors: map }
Main:
  result.isValid = true
  if disputeForm.response == "": result.isValid = false; result.errors["response"] = "required"
  return result
DisputeControl.processDispute
Input: disputeForm: DisputeForm
Output: "INCOMPLETE" | "DISPUTE_SUBMITTED" | "ERROR_DB_CORRUPT" | "COMPLAINT_NOT_FOUND"
Main:
  if not DisputeControl.isDisputeComplete(disputeForm): return "INCOMPLETE"
  try:
    complaint = DB.find("Complaints", { id: disputeForm.complaintId })
  catch DBError:
    return "ERROR_DB_CORRUPT"
  if complaint == null: return "COMPLAINT_NOT_FOUND"
  try:
    ComplaintDatabase.addDisputeResponse(disputeForm.complaintId, disputeForm.response)
    warnings = DisputeControl.getWarningsForAccount(complaint.accountId)
    return "DISPUTE_SUBMITTED"
  catch DBError:
    return "ERROR_DB_CORRUPT"
DisputeControl.isDisputeComplete
Input: disputeForm: DisputeForm
Output: bool
Main:
  if disputeForm.response == null or trim(disputeForm.response) == "": return false
  return true
ComplaintDatabase.addDisputeResponse
Input: complaintId: string, response: string
Output: "OK" | throws DBError
Main:
  DB.update("Complaints", { id: complaintId }, { disputeResponse: response, disputeStatus: "DISPUTED", disputedAt: now() }) // db function assumed
  return "OK"
DisputeControl.getWarningsForAccount
Input: accountId: string
Output: warnings[] | "ERROR_DB_CORRUPT"
Main:
  try:
    warnings = WarningDatabase.getWarningsByAccount(accountId)
    return warnings
  catch DBError:
    return "ERROR_DB_CORRUPT"
WarningDatabase.getWarningsByAccount
Input: accountId: string
Output: warnings[] | throws DBError
Main:

```
  return DB.findAll("Warnings", { accountId: accountId }) // db function assumed
```

AccountPage.showWarning

```
Input: warnings[]
Output: void
Main:
  if length(warnings) > 0:
    UI.displayWarning("You have " + length(warnings) + " warning(s) on your account")
  UI.renderList("warnings", warnings)
```

5. Place Order

Main → CustomerAccount.checkStatus → (VIP → SpecialMenu.display | NON_VIP → Menu.display) → Menu.browseItems → Cart.addItem → Cart.confirmOrder → OrderControl.calculateTotal → (VIP → OrderControl.checkFreeDelivery) → OrderControl.processCost → PaymentControl.validateBalance → (SUFFICIENT → PaymentControl.processPayment → OrderControl.sendOrder → AccountControl.checkUpgrade) | (INSUFFICIENT → WarningControl.issueWarning → WarningControl.checkThreshold)

CustomerAccount.checkStatus

```
Input: accountId: string
Output: "VIP" | "NON_VIP" | "ERROR_DB_CORRUPT"
Main:
  try:
    account = DB.find("Accounts", { id: accountId })
  catch DBError:
    return "ERROR_DB_CORRUPT"
  if account.status == "VIP": return "VIP"
  return "NON_VIP"
```

Menu.display

```
Input: accountType: "VIP" | "NON_VIP"
Output: menuItems[]
Main:
  if accountType == "VIP":
    items = DB.findAll("MenuItems", { includeSpecial: true })
  else:
    items = DB.findAll("MenuItems", { includeSpecial: false })
  return items
```

SpecialMenu.display

```
Input: none
Output: specialItems[]
Main:
  items = DB.findAll("MenuItems", { special: true })
  return items
```

Menu.browseItems

```
Input: menuItems[]
Output: void
Main:
  UI.renderMenu("items", menuItems)
```

Cart.addItem

Input: itemId: string, accountType: "VIP" | "NON_VIP"

Output: cartItem: CartItem

Main:

```
  item = DB.find("MenuItems", { id: itemId })
  cartItem.itemId = item.id
  cartItem.name = item.name
  cartItem.price = item.price
  cartItem.accountType = accountType
  Session.addToCart(cartItem) // session function assumed
  return cartItem
```

Cart.confirmOrder

Input: accountId: string

Output: order: Order

Main:

```
  cartItems = Session.getCart()
  order.id = UUID()
  order.accountId = accountId
  order.items = cartItems
  order.createdAt = now()
  return order
```

OrderControl.calculateTotal

Input: order: Order, accountType: "VIP" | "NON_VIP"

Output: orderCost: OrderCost

Main:

```
  subtotal = sum(item.price for item in order.items)
  if accountType == "VIP":
    orderCost.subtotal = subtotal * 0.9 // 10% VIP discount
    orderCost.accountType = "VIP"
  else:
    orderCost.subtotal = subtotal
    orderCost.accountType = "NON_VIP"
    orderCost.deliveryFee = 5.00
    orderCost.total = orderCost.subtotal + orderCost.deliveryFee
  return orderCost
```

OrderControl.checkFreeDelivery

Input: orderCost: OrderCost, accountId: string

Output: orderCost: OrderCost

Main:

```
  orderCount = DB.count("Orders", { accountId: accountId, status: "COMPLETED" })
  if orderCount >= 3:
    orderCost.deliveryFee = 0.00
    orderCost.freeDeliveryApplied = true
  else:
    orderCost.deliveryFee = 5.00
    orderCost.freeDeliveryApplied = false
```

```
  orderCost.total = orderCost.subtotal + orderCost.deliveryFee
  return orderCost
```

OrderControl.processCost

```
Input: orderCost: OrderCost
Output: totalCost: decimal
Main:
  return orderCost.total
```

PaymentControl.validateBalance

```
Input: accountId: string, totalCost: decimal
Output: "SUFFICIENT" | "INSUFFICIENT" | "ERROR_DB_CORRUPT"
Main:
  try:
    account = DB.find("Accounts", { id: accountId })
  catch DBError:
    return "ERROR_DB_CORRUPT"
  if account.balance >= totalCost: return "SUFFICIENT"
  return "INSUFFICIENT"
```

PaymentControl.processPayment

```
Input: accountId: string, totalCost: decimal, orderId: string
Output: "OK" | "ERROR_DB_CORRUPT"
Main:
  try:
    DB.beginTransaction()
    account = DB.find("Accounts", { id: accountId })
    newBalance = account.balance - totalCost
    DB.update("Accounts", { id: accountId }, { balance: newBalance })
    DB.update("Orders", { id: orderId }, { status: "PAID", paidAt: now() })
    DB.commit()
    return "OK"
  catch DBError:
    DB.rollback()
    return "ERROR_DB_CORRUPT"
```

OrderControl.sendOrder

```
Input: orderId: string
Output: "OK" | "ERROR_DB_CORRUPT"
Main:
  try:
    DB.update("Orders", { id: orderId }, { status: "SENT", sentAt: now() })
    return "OK"
  catch DBError:
    return "ERROR_DB_CORRUPT"
```

AccountControl.checkUpgrade

```
Input: accountId: string, totalCost: decimal
Output: "UPGRADED" | "NO_CHANGE" | "ERROR_DB_CORRUPT"
Main:
  if totalCost < 100.00: return "NO_CHANGE"
```

```
  try:
    account = DB.find("Accounts", { id: accountId })
    if account.status != "VIP":
      DB.update("Accounts", { id: accountId }, { status: "VIP", upgradedAt: now() })
      return "UPGRADED"
    return "NO_CHANGE"
  catch DBError:
    return "ERROR_DB_CORRUPT"
```

WarningControl.issueWarning

Input: accountId: string

Output: "WARNING_ISSUED" | "ERROR_DB_CORRUPT"

Main:

```
  try:
    warning = Warning(id = UUID(), accountId = accountId, reason = "INSUFFICIENT_BALANCE", createdAt = now())
    DB.insert("Warnings", warning)
    UI.displayError("Insufficient balance to complete order")
    return "WARNING_ISSUED"
  catch DBError:
    return "ERROR_DB_CORRUPT"
```

WarningControl.checkThreshold

Input: accountId: string

Output: "BLACKLISTED" | "DEGRADED" | "WARNING_ONLY" | "ERROR_DB_CORRUPT"

Main:

```
  try:
    warningCount = DB.count("Warnings", { accountId: accountId })
    if warningCount >= 3:
      DB.update("Accounts", { id: accountId }, { status: "BLACKLISTED", blacklistedAt: now() })
      Session.logout()
      UI.redirect("login")
      return "BLACKLISTED"
    else if warningCount >= 2:
      account = DB.find("Accounts", { id: accountId })
      if account.status == "VIP":
        DB.update("Accounts", { id: accountId }, { status: "REGULAR", degradedAt: now() })
        return "DEGRADED"
    return "WARNING_ONLY"
  catch DBError:
    return "ERROR_DB_CORRUPT"
```

CustomerPage.display

Input: accountId: string

Output: void

Main:

```
  account = DB.find("Accounts", { id: accountId })
  UI.renderPage("customer", account)
```

OrderInfoPage.display

Input: orderId: string

Output: void
Main:
  order = DB.find("Orders", { id: orderId })
  UI.renderPage("orderInfo", order)

6. Deposit Money

Main → UserAccount.requestDeposit → DepositControl.initiate → DepositControl.collectAmount → DepositControl.collectBankInfo → BankControl.validateAccount → BankControl.chargeAccount → BalanceControl.updateBalance → UserAccount.showUpdatedBalance

UserAccount.requestDeposit
Input: accountId: string
Output: depositSession: DepositSession
Main:
  depositSession.id = UUID()
  depositSession.accountId = accountId
  depositSession.status = "IN_PROGRESS"
  depositSession.createdAt = now()
  return depositSession
DepositControl.initiate
Input: depositSession: DepositSession
Output: void
Main:
  UI.displayDepositForm(depositSession.id)
DepositControl.collectAmount
Input: depositSession: DepositSession (reads UI fields)
Output: depositForm: DepositForm
Main:
  depositForm.sessionId = depositSession.id
  depositForm.amount = UI.getValue("amount")
  return depositForm
DepositControl.collectBankInfo
Input: depositForm: DepositForm (reads UI fields)
Output: depositForm: DepositForm
Main:
  depositForm.bankAccount = trim(UI.getValue("bankAccount"))
  depositForm.routingNumber = trim(UI.getValue("routingNumber"))
  depositForm.accountHolder = trim(UI.getValue("accountHolder"))
  return depositForm
DepositControl.validateInput
Input: depositForm: DepositForm
Output: ValidationResult { isValid: bool, errors: map }
Main:
  result.isValid = true
  if depositForm.amount <= 0: result.isValid = false; result.errors["amount"] = "must be positive"
  if depositForm.bankAccount == "": result.isValid = false; result.errors["bankAccount"] = "required"

```
  if depositForm.routingNumber == "": result.isValid = false; result.errors["routingNumber"] = "required"
  return result
```

BankControl.validateAccount

```
Input: depositForm: DepositForm
Output: bankAccount: BankAccount | "INVALID_ACCOUNT"
Main:
  validation = DepositControl.validateInput(depositForm)
  if validation.isValid == false: return "INVALID_ACCOUNT"
  isValid = BankControl.verifyBankDetails(depositForm.bankAccount, depositForm.routingNumber,
depositForm.accountHolder)
  if not isValid: return "INVALID_ACCOUNT"
  bankAccount.accountNumber = depositForm.bankAccount
  bankAccount.routingNumber = depositForm.routingNumber
  bankAccount.accountHolder = depositForm.accountHolder
  bankAccount.verified = true
  return bankAccount
```

BankControl.verifyBankDetails

```
Input: accountNumber: string, routingNumber: string, accountHolder: string
Output: bool
Main:
  // validate format
  if length(accountNumber) < 8 or length(accountNumber) > 17: return false
  if length(routingNumber) != 9: return false
  if accountHolder == "": return false
  if not BankControl.isValidRoutingNumber(routingNumber): return false
  return true
```

BankControl.isValidRoutingNumber

```
Input: routingNumber: string
Output: bool
Main:
  // simplified checksum validation assumed
  if not isNumeric(routingNumber): return false
  return ExternalBankAPI.verify(routingNumber) // external bank API validation assumed
```

BankControl.chargeAccount

```
Input: bankAccount: BankAccount, amount: decimal
Output: transaction: Transaction | "CHARGE_FAILED"
Main:
  try:
    // external bank API call assumed
    transactionId = ExternalBankAPI.charge(bankAccount.accountNumber, bankAccount.routingNumber, amount)
    transaction.id = transactionId
    transaction.amount = amount
    transaction.status = "COMPLETED"
    transaction.timestamp = now()
    return transaction
  catch BankAPIError:
    return "CHARGE_FAILED"
```

BalanceControl.updateBalance

Input: accountId: string, depositAmount: decimal, transactionId: string

Output: "OK" | "ERROR_DB_CORRUPT"

Main:

```
  try:
    DB.beginTransaction()
    account = DB.find("Accounts", { id: accountId })
    currentBalance = account.balance
    newBalance = currentBalance + depositAmount
    DB.update("Accounts", { id: accountId }, { balance: newBalance })
    depositRecord = DepositRecord(id = UUID(), accountId = accountId, amount = depositAmount, transactionId =
transactionId, timestamp = now())
    DB.insert("Deposits", depositRecord)
    DB.commit()
    return "OK"
  catch DBError:
    DB.rollback()
    return "ERROR_DB_CORRUPT"
```

UserAccount.showUpdatedBalance

Input: accountId: string

Output: void

Main:

```
  account = DB.find("Accounts", { id: accountId })
  UI.displaySuccess("Deposit successful. New balance: $" + account.balance)
  UI.updateBalanceDisplay(account.balance)
```

DepositControl.showError

Input: errorType: "INVALID_ACCOUNT" | "CHARGE_FAILED" | "ERROR_DB_CORRUPT"

Output: void

Main:

```
  if errorType == "INVALID_ACCOUNT":
    UI.displayError("Invalid bank account information. Please check your details and try again.")
  else if errorType == "CHARGE_FAILED":
    UI.displayError("Unable to charge bank account. Please verify your account has sufficient funds.")
  else if errorType == "ERROR_DB_CORRUPT":
    UI.displayError("System error. Please try again later.")
```

7. Request Account Closure

Main → AccountSettingInterface.requestClosure → AccountSettingInterface.collectClosureDetails → AccountSettingInterface.submitRequest → AccountRequest.processClosureRequest → RequestDatabase.storeClosureRequest

AccountSettingInterface.requestClosure

Input: accountId: string

Output: void

Main:

```
  UI.displayClosureForm(accountId)
```

AccountSettingInterface.collectClosureDetails

Input: accountId: string (reads UI fields)
Output: closureForm: ClosureForm
Main:
  closureForm.accountId = accountId
  closureForm.reason = trim(UI.getValue("reason"))
  closureForm.additionalInfo = trim(UI.getValue("additionalInfo"))
  return closureForm

AccountSettingInterface.submitRequest

Input: closureForm: ClosureForm
Output: result: "INCOMPLETE" | "REQUEST_SUBMITTED"
Main:
  validation = AccountSettingInterface.validateClosureForm(closureForm)
  if validation.isValid == false: return "INCOMPLETE"
  return AccountRequest.processClosureRequest(closureForm)

AccountSettingInterface.validateClosureForm

Input: closureForm: ClosureForm
Output: ValidationResult { isValid: bool, errors: map }
Main:
  result.isValid = true
  if closureForm.reason == "": result.isValid = false; result.errors["reason"] = "required"
  return result

AccountRequest.processClosureRequest

Input: closureForm: ClosureForm
Output: "INCOMPLETE" | "REQUEST_SUBMITTED" | "ERROR_DB_CORRUPT"
Main:
  if not AccountRequest.isClosureComplete(closureForm): return "INCOMPLETE"
  try:
    account = DB.find("Accounts", { id: closureForm.accountId })
  catch DBError:
    return "ERROR_DB_CORRUPT"
  request = AccountRequest.buildClosureRequest(closureForm, account)
  result = RequestDatabase.storeClosureRequest(request)
  if result == "OK": return "REQUEST_SUBMITTED"
  return "ERROR_DB_CORRUPT"

AccountRequest.isClosureComplete

Input: closureForm: ClosureForm
Output: bool
Main:
  if closureForm.reason == null or trim(closureForm.reason) == "": return false
  return true

AccountRequest.buildClosureRequest

Input: closureForm: ClosureForm, account: Account
Output: closureRequest: ClosureRequest
Main:
  closureRequest.id = UUID()
  closureRequest.accountId = closureForm.accountId

```
  closureRequest.email = account.email
  closureRequest.reason = closureForm.reason
  closureRequest.additionalInfo = closureForm.additionalInfo
  closureRequest.status = "PENDING"
  closureRequest.requestedAt = now()
  return closureRequest
```

RequestDatabase.storeClosureRequest

```
Input: closureRequest: ClosureRequest
Output: "OK" | "ERROR_DB_CORRUPT"
Main:
  try:
    DB.insert("ClosureRequests", closureRequest)
    return "OK"
  catch DBError:
    return "ERROR_DB_CORRUPT"
```

AccountSettingInterface.showSuccess

```
Input: none
Output: void
Main:
  UI.displaySuccess("Your account closure request has been submitted. A manager will review it shortly.")
```

```
AccountSettingInterface.showError
Input: errorType: "INCOMPLETE" | "ERROR_DB_CORRUPT"
Output: void
Main:
  if errorType == "INCOMPLETE":
    UI.displayError("Please provide a reason for account closure")
  else if errorType == "ERROR_DB_CORRUPT":
    UI.displayError("System error. Please try again later.")
```

8. Handle Complaints / Compliments

Main → ManagerInterface.openComplaintScreen → ReviewControl.getComplaintRequests → ComplaintDatabase.getPendingReviews → ReviewControl → ManagerInterface.showReviews → ManagerInterface.collectResponse → ManagerInterface.submitDecision → ReviewControl.processDecision → (ACCEPT_DISPUTE → WarningControl.updateWarnings → AccountControl.updateComplainerWarnings) | (DENY_DISPUTE → WarningControl.updateWarnings → AccountControl.updateComplaineeWarnings) | (ACCEPT_COMPLIMENT → ComplaintDatabase.updateCompliment → EmployeeControl.updateEmployeeRecord)

ManagerInterface.openComplaintScreen

```
Input: none
Output: void
Main:
  UI.displayComplaintReviewScreen()
```

ReviewControl.getComplaintRequests

```
Input: none
Output: reviews[] | "ERROR_DB_CORRUPT"
```

Main:
  try:
    reviews = ComplaintDatabase.getPendingReviews()
    return reviews
  catch DBError:
    return "ERROR_DB_CORRUPT"

ComplaintDatabase.getPendingReviews

Input: none
Output: reviews[] | throws DBError
Main:
  complaints = DB.findAll("Complaints", { disputeStatus: "DISPUTED" })
  compliments = DB.findAll("Compliments", { status: "PENDING" })
  reviews = complaints.concat(compliments)
  return reviews

ManagerInterface.showReviews

Input: reviews[]
Output: void
Main:
  UI.renderList("pendingReviews", reviews)

ManagerInterface.collectResponse

Input: reviewId: string, reviewType: "COMPLAINT" | "COMPLIMENT" (reads UI fields)
Output: responseForm: ReviewResponseForm
Main:
  responseForm.reviewId = reviewId
  responseForm.reviewType = reviewType
  responseForm.decision = UI.getValue("decision") // "ACCEPT" | "DENY"
  responseForm.managerNotes = trim(UI.getValue("managerNotes"))
  return responseForm

ManagerInterface.submitDecision

Input: responseForm: ReviewResponseForm
Output: result: "INCOMPLETE_DATA" | "DECISION_PROCESSED" | "ERROR_DB_CORRUPT"
Main:
  return ReviewControl.processDecision(responseForm)

ReviewControl.processDecision

Input: responseForm: ReviewResponseForm
Output: "INCOMPLETE_DATA" | "DECISION_PROCESSED" | "ERROR_DB_CORRUPT" | "NOT_FOUND"
Main:
  try:
    if responseForm.reviewType == "COMPLAINT":
      complaint = DB.find("Complaints", { id: responseForm.reviewId })
      if complaint == null: return "NOT_FOUND"
      if not ReviewControl.isComplaintDataComplete(complaint): return "INCOMPLETE_DATA"
      return ReviewControl.processComplaintDispute(complaint, responseForm)
    else if responseForm.reviewType == "COMPLIMENT":
      compliment = DB.find("Compliments", { id: responseForm.reviewId })

```
    if compliment == null: return "NOT_FOUND"
    if not ReviewControl.isComplimentDataComplete(compliment): return "INCOMPLETE_DATA"
    return ReviewControl.processCompliment(compliment, responseForm)
  catch DBError:
    return "ERROR_DB_CORRUPT"
```

ReviewControl.isComplaintDataComplete

```
Input: complaint: Complaint
Output: bool
Main:
  if complaint.complainerId == null or complaint.complainerId == "": return false
  if complaint.complaineeId == null or complaint.complaineeId == "": return false
  if complaint.disputeResponse == null or trim(complaint.disputeResponse) == "": return false
  return true
```

ReviewControl.isComplimentDataComplete

```
Input: compliment: Compliment
Output: bool
Main:
  if compliment.employeeId == null or compliment.employeeId == "": return false
  if compliment.customerId == null or compliment.customerId == "": return false
  return true
```

```
ReviewControl.processComplaintDispute
Input: complaint: Complaint, responseForm: ReviewResponseForm
Output: "DECISION_PROCESSED" | "ERROR_DB_CORRUPT"
Main:
  try:
    DB.beginTransaction()
    if responseForm.decision == "ACCEPT": // Accept dispute - complaint was invalid, warn complainer
      DB.update("Complaints", { id: complaint.id }, { disputeStatus: "ACCEPTED", resolvedAt: now(), managerNotes:
responseForm.managerNotes })
      WarningControl.updateWarnings(complaint.complainerId, "INVALID_COMPLAINT")
      AccountControl.updateComplainerWarnings(complaint.complainerId)
    else if responseForm.decision == "DENY": // Deny dispute - complaint was valid, warn complainee
      DB.update("Complaints", { id: complaint.id }, { disputeStatus: "DENIED", resolvedAt: now(), managerNotes:
responseForm.managerNotes })
      WarningControl.updateWarnings(complaint.complaineeId, "VALID_COMPLAINT")
      AccountControl.updateComplaineeWarnings(complaint.complaineeId)
    DB.commit()
    return "DECISION_PROCESSED"
  catch DBError:
    DB.rollback()
    return "ERROR_DB_CORRUPT"
```

ReviewControl.processCompliment

```
Input: compliment: Compliment, responseForm: ReviewResponseForm
Output: "DECISION_PROCESSED" | "ERROR_DB_CORRUPT"
Main:
  try:
```

```
    DB.beginTransaction()
    if responseForm.decision == "ACCEPT":
      DB.update("Compliments", { id: compliment.id }, { status: "ACCEPTED", resolvedAt: now(), managerNotes:
responseForm.managerNotes })
      ComplaintDatabase.updateCompliment(compliment.id, "ACCEPTED")
      EmployeeControl.updateEmployeeRecord(compliment.employeeId, "COMPLIMENT_RECEIVED")
    else if responseForm.decision == "DENY":
      DB.update("Compliments", { id: compliment.id }, { status: "DENIED", resolvedAt: now(), managerNotes:
responseForm.managerNotes })
    DB.commit()
    return "DECISION_PROCESSED"
  catch DBError:
    DB.rollback()
    return "ERROR_DB_CORRUPT"
```

WarningControl.updateWarnings

Input: accountId: string, warningType: "INVALID_COMPLAINT" | "VALID_COMPLAINT"

Output: "OK" | "ERROR_DB_CORRUPT"

Main:

```
  try:
    warning = Warning(id = UUID(), accountId = accountId, reason = warningType, createdAt = now())
    DB.insert("Warnings", warning)
    return "OK"
  catch DBError:
    return "ERROR_DB_CORRUPT"
```

AccountControl.updateComplainerWarnings

Input: complainerId: string

Output: "OK" | "ERROR_DB_CORRUPT"

Main:

```
  try:
    warningCount = DB.count("Warnings", { accountId: complainerId, reason: "INVALID_COMPLAINT" })
    if warningCount >= 3:
      DB.update("Accounts", { id: complainerId }, { status: "RESTRICTED", restrictedAt: now() })
    return "OK"
  catch DBError:
    return "ERROR_DB_CORRUPT"
```

AccountControl.updateComplaineeWarnings

Input: complaineeId: string

Output: "OK" | "ERROR_DB_CORRUPT"

Main:

```
  try:
    warningCount = DB.count("Warnings", { accountId: complaineeId, reason: "VALID_COMPLAINT" })
    if warningCount >= 3:
      DB.update("Accounts", { id: complaineeId }, { status: "SUSPENDED", suspendedAt: now() })
    return "OK"
  catch DBError:
    return "ERROR_DB_CORRUPT"
```

ComplaintDatabase.updateCompliment

Input: complimentId: string, status: string
Output: "OK" | throws DBError
Main:
  DB.update("Compliments", { id: complimentId }, { status: status, processedAt: now() })
  return "OK"

EmployeeControl.updateEmployeeRecord

Input: employeeId: string, updateType: "COMPLIMENT_RECEIVED"
Output: "OK" | "ERROR_DB_CORRUPT"
Main:
  try:
    employee = DB.find("Employees", { id: employeeId })
    complimentCount = employee.complimentCount + 1
    DB.update("Employees", { id: employeeId }, { complimentCount: complimentCount, lastComplimentAt: now() })
    return "OK"
  catch DBError:
    return "ERROR_DB_CORRUPT"

ManagerInterface.showSuccess

Input: decision: "ACCEPT" | "DENY"
Output: void
Main:
  if decision == "ACCEPT":
    UI.displaySuccess("Dispute accepted. Warnings updated accordingly.")
  else:
    UI.displaySuccess("Dispute denied. Warnings updated accordingly.")

ManagerInterface.showError

Input: errorType: "INCOMPLETE_DATA" | "ERROR_DB_CORRUPT" | "NOT_FOUND"
Output: void
Main:
  if errorType == "INCOMPLETE_DATA":
    UI.displayError("Incomplete data in request. Unable to process.")
  else if errorType == "NOT_FOUND":
    UI.displayError("Request not found.")
  else if errorType == "ERROR_DB_CORRUPT":
    UI.displayError("Database error. Please try again later.")

9. Add Local Information to Service Model

Main → UserAccount.requestDataAddition → DataAdditionControl.initiate → DataAdditionControl.collectQueryAndAnswer → DataAdditionControl.submitAddition → KnowledgeDatabase.storeInformation

UserAccount.requestDataAddition

Input: accountId: string
Output: additionSession: AdditionSession
Main:
  additionSession.id = UUID()
  additionSession.accountId = accountId
  additionSession.status = "IN_PROGRESS"

```
  additionSession.createdAt = now()
  return additionSession
```

DataAdditionControl.initiate

```
Input: additionSession: AdditionSession
Output: void
Main:
  UI.displayDataAdditionForm(additionSession.id)
```

DataAdditionControl.collectQueryAndAnswer

```
Input: additionSession: AdditionSession (reads UI fields)
Output: additionForm: DataAdditionForm
Main:
  additionForm.sessionId = additionSession.id
  additionForm.accountId = additionSession.accountId
  additionForm.query = trim(UI.getValue("query"))
  additionForm.answer = trim(UI.getValue("answer"))
  return additionForm
```

DataAdditionControl.validateInput

```
Input: additionForm: DataAdditionForm
Output: ValidationResult { isValid: bool, errors: map }
Main:
  result.isValid = true
  if additionForm.query == "": result.isValid = false; result.errors["query"] = "required"
  if additionForm.answer == "": result.isValid = false; result.errors["answer"] = "required"
  return result
```

DataAdditionControl.submitAddition

```
Input: additionForm: DataAdditionForm
Output: "INCOMPLETE" | "ADDITION_SAVED" | "ERROR_DB_UNAVAILABLE"
Main:
  validation = DataAdditionControl.validateInput(additionForm)
  if validation.isValid == false: return "INCOMPLETE"
  knowledgeEntry = DataAdditionControl.buildKnowledgeEntry(additionForm)
  result = KnowledgeDatabase.storeInformation(knowledgeEntry)
  if result == "OK": return "ADDITION_SAVED"
  return "ERROR_DB_UNAVAILABLE"
```

DataAdditionControl.buildKnowledgeEntry

```
Input: additionForm: DataAdditionForm
Output: knowledgeEntry: KnowledgeEntry
Main:
  knowledgeEntry.id = UUID()
  knowledgeEntry.query = additionForm.query
  knowledgeEntry.answer = additionForm.answer
  knowledgeEntry.contributorId = additionForm.accountId
  knowledgeEntry.status = "ACTIVE"
  knowledgeEntry.createdAt = now()
  return knowledgeEntry
```

KnowledgeDatabase.storeInformation

Input: knowledgeEntry: KnowledgeEntry
Output: "OK" | "ERROR_DB_UNAVAILABLE"
Main:
```
try:
  DB.insert("KnowledgeBase", knowledgeEntry)
  return "OK"
catch DBError:
  return "ERROR_DB_UNAVAILABLE"
```
DataAdditionControl.showSuccess
Input: none
Output: void
Main:
```
UI.displaySuccess("Your information has been added to the service model successfully.")
```
DataAdditionControl.showError
Input: errorType: "INCOMPLETE" | "ERROR_DB_UNAVAILABLE"
Output: void
Main:
```
if errorType == "INCOMPLETE":
  UI.displayError("Please provide both a query and an answer")
else if errorType == "ERROR_DB_UNAVAILABLE":
  UI.displayError("Database is currently unavailable. Please try again later.")
```

10. Ask Customer Service Information

Main → AskAgentMenu.collectQuestion → AskAgentMenu.submitQuestion → AgentControl.processQuestion → AgentControl.checkLocalAnswer → AnswerDatabase.getLocalAnswer → (LOCAL_FOUND → AgentControl.returnLocalAnswer → AskAgentMenu.showLocalAnswer → AskAgentMenu.requestRating) | (LOCAL_NOT_FOUND → AgentControl.queryLLM → LLM.generateAnswer → AgentControl.returnAIAnswer → AskAgentMenu.showAIAnswer)

AskAgentMenu.collectQuestion
Input: accountId: string (reads UI fields)
Output: questionForm: QuestionForm
Main:
```
questionForm.accountId = accountId
questionForm.question = trim(UI.getValue("question"))
questionForm.timestamp = now()
return questionForm
```
AskAgentMenu.submitQuestion
Input: questionForm: QuestionForm
Output: result: "INCOMPLETE" | "ANSWER_DISPLAYED" | "ERROR_AI_UNAVAILABLE"
Main:
```
validation = AskAgentMenu.validateQuestion(questionForm)
if validation.isValid == false: return "INCOMPLETE"
return AgentControl.processQuestion(questionForm)
```
AskAgentMenu.validateQuestion
Input: questionForm: QuestionForm
Output: ValidationResult { isValid: bool, errors: map }

```
Main:
  result.isValid = true
  if questionForm.question == "": result.isValid = false; result.errors["question"] = "required"
  return result
```

AgentControl.processQuestion

```
Input: questionForm: QuestionForm
Output: "ANSWER_DISPLAYED" | "ERROR_AI_UNAVAILABLE"
Main:
  localAnswer = AgentControl.checkLocalAnswer(questionForm.question)
  if localAnswer != null:
    AgentControl.logQuery(questionForm.accountId, questionForm.question, "LOCAL", localAnswer.id)
    return AgentControl.returnLocalAnswer(localAnswer)
  else:
    aiAnswer = AgentControl.queryLLM(questionForm.question)
    if aiAnswer == "ERROR_AI_UNAVAILABLE": return "ERROR_AI_UNAVAILABLE"
    AgentControl.logQuery(questionForm.accountId, questionForm.question, "AI", null)
    return AgentControl.returnAIAnswer(aiAnswer)
```

AgentControl.checkLocalAnswer

```
Input: question: string
Output: localAnswer: LocalAnswer | null
Main:
  result = AnswerDatabase.getLocalAnswer(question)
  return result
```

AnswerDatabase.getLocalAnswer

```
Input: question: string
Output: localAnswer: LocalAnswer | null
Main:
  // search for matching query in knowledge base
  normalizedQuestion = lowerCase(trim(question))
  matches = DB.findAll("KnowledgeBase", { status: "ACTIVE" })
  for match in matches:
    if AgentControl.isQuestionMatch(normalizedQuestion, lowerCase(match.query)):
      localAnswer.id = match.id
      localAnswer.query = match.query
      localAnswer.answer = match.answer
      localAnswer.source = "LOCAL"
      return localAnswer
  return null
```

AgentControl.isQuestionMatch

```
Input: question1: string, question2: string
Output: bool
Main:
  // simple keyword matching (more sophisticated matching could be implemented)
  if question1 == question2: return true
  keywords1 = split(question1, " ")
  keywords2 = split(question2, " ")
```

```
  matchCount = 0
  for word in keywords1:
    if word in keywords2: matchCount = matchCount + 1
  matchThreshold = length(keywords1) * 0.7
  if matchCount >= matchThreshold: return true
  return false
```

AgentControl.returnLocalAnswer

```
Input: localAnswer: LocalAnswer
Output: "ANSWER_DISPLAYED"
Main:
  AskAgentMenu.showLocalAnswer(localAnswer)
  return "ANSWER_DISPLAYED"
```

AskAgentMenu.showLocalAnswer

```
Input: localAnswer: LocalAnswer
Output: void
Main:
  UI.displayAnswer(localAnswer.answer, "LOCAL")
  UI.displayMessage("This answer was provided by our knowledge base.")
```

AskAgentMenu.requestRating

```
Input: answerId: string
Output: void
Main:
  UI.displayRatingPrompt("Was this answer helpful?", answerId)
```

AskAgentMenu.submitRating

```
Input: answerId: string, rating: int
Output: "OK" | "ERROR_DB_CORRUPT"
Main:
  try:
    DB.update("KnowledgeBase", { id: answerId }, {
      rating: rating,
      ratedAt: now()
    })
    UI.displaySuccess("Thank you for your feedback!")
    return "OK"
  catch DBError:
    return "ERROR_DB_CORRUPT"
```

AgentControl.queryLLM

```
Input: question: string
Output: aiAnswer: string | "ERROR_AI_UNAVAILABLE"
Main:
  try:
    response = LLM.generateAnswer(question)
    return response
  catch LLMError:
    return "ERROR_AI_UNAVAILABLE"
```

LLM.generateAnswer

Input: question: string
Output: answer: string | throws LLMError
Main:
  response = ExternalLLMAPI.query(question) // call to external LLM API assumed
 if response == null or response == "": throw LLMError("unavailable")
 return response.answer

AgentControl.returnAIAnswer

Input: aiAnswer: string
Output: "ANSWER_DISPLAYED"
Main:
 AskAgentMenu.showAIAnswer(aiAnswer)
 return "ANSWER_DISPLAYED"

AskAgentMenu.showAIAnswer

Input: aiAnswer: string
Output: void
Main:
 UI.displayAnswer(aiAnswer, "AI")
 UI.displayMessage("This answer was generated by our AI assistant.")

AgentControl.logQuery

Input: accountId: string, question: string, answerType: "LOCAL" | "AI", answerId: string | null
Output: void
Main:
 try:
  queryLog = QueryLog(id = UUID(), accountId = accountId, question = question, answerType = answerType, answerId = answerId, timestamp = now())
  DB.insert("QueryLogs", queryLog)
 catch DBError:
  // log error but don't fail the user request
  return

AskAgentMenu.showError

Input: errorType: "INCOMPLETE" | "ERROR_AI_UNAVAILABLE"
Output: void
Main:
 if errorType == "INCOMPLETE":
  UI.displayError("Please enter a question")
 else if errorType == "ERROR_AI_UNAVAILABLE":
  UI.displayError("AI agent

11. Review Local Knowledge Answer

Flow: Review Local Knowledge Answer Main → ManagerPage.viewFlaggedAnswers → ManagerPage.reviewAnswer → KnowledgeBaseSystem.processReview

ManagerPage.viewFlaggedAnswers

Input: none
Output: list<FlaggedAnswer>
Main:
flaggedAnswers = DB.findAll("KnowledgeBaseAnswers", { flagged: true, reviewStatus: "PENDING" })

return flaggedAnswers

ManagerPage.reviewAnswer
Input: none
Output: ReviewForm
Main:
form.answerId = UI.getValue("answerId")
form.decision = UI.getValue("decision") // "KEEP" | "REMOVE"
form.reviewNotes = trim(UI.getValue("reviewNotes"))
return form

ManagerPage.handleReviewSubmit
Input: form: ReviewForm
Output: "SUCCESS" | "ANSWER_NOT_FOUND"
Main:
return KnowledgeBaseSystem.processReview(form)

KnowledgeBaseSystem.processReview
Input: form: ReviewForm
Output: "SUCCESS" | "ANSWER_NOT_FOUND"
Main:
answer = DB.findOne("KnowledgeBaseAnswers", { id: form.answerId })
if answer == null:
  return "ANSWER_NOT_FOUND"

if form.decision == "REMOVE":
  KnowledgeBaseSystem.removeAnswer(form.answerId, answer.authorId)
else:
  // keep answer, just unflag it
  DB.update("KnowledgeBaseAnswers", { id: form.answerId }, {
    flagged: false,
    reviewStatus: "APPROVED",
    reviewedAt: now(),
    reviewedBy: currentManagerId, // assumed context
    reviewNotes: form.reviewNotes
  })

return "SUCCESS"

KnowledgeBaseSystem.removeAnswer
Input: answerId: string, authorId: string
Output: none
Main:
answer = DB.findOne("KnowledgeBaseAnswers", { id: answerId })

// mark answer as removed

```
DB.update("KnowledgeBaseAnswers", { id: answerId }, {
  status: "REMOVED",
  removedAt: now(),
  removedBy: currentManagerId  // assumed context
})

// check if query has other valid answers
queryId = answer.queryId
remainingAnswers = DB.count("KnowledgeBaseAnswers", {
  queryId: queryId,
  status: "ACTIVE"
})

// if no other answers remain, remove the query
if remainingAnswers == 0:
  DB.update("KnowledgeBaseQueries", { id: queryId }, {
    status: "REMOVED",
    removedAt: now()
  })

// prohibit author from providing future answers
KnowledgeBaseSystem.banAuthor(authorId)
```

KnowledgeBaseSystem.banAuthor

```
Input: authorId: string
Output: none
Main:
author = DB.findOne("Users", { id: authorId })
if author == null:
  return

DB.update("Users", { id: authorId }, {
  knowledgeBaseContributor: false,
  bannedFromKB: true,
  bannedAt: now()
})

// notify author of ban
UserNotification.sendKnowledgeBaseBanNotification(author.email)
```

KnowledgeBaseSystem.flagAnswerForReview

```
Input: answerId: string, flaggedBy: string
Output: none
Main:
DB.update("KnowledgeBaseAnswers", { id: answerId }, {
  flagged: true,
```

```
    flaggedBy: flaggedBy,
    flaggedAt: now(),
    reviewStatus: "PENDING"
})

// notify manager
ManagerNotification.sendKBReviewAlert(answerId)
```

12. Create/Update Menu
    Flow: Main → DishEditorPage.collectInput → DishEditorPage.validate → DishManager.createOrUpdateDish → MenuDatabase.insert/update

DishEditorPage.collectInput

```
Input: none (reads UI fields)
Output: DishForm
Main:
form.name        = trim(UI.getValue("dishName"))
form.description = trim(UI.getValue("description"))
form.price       = UI.getNumber("price")
form.allergens   = UI.getList("allergens")     // optional
form.calories    = UI.getNumber("calories")    // optional
form.imageFile   = UI.getFile("dishImage")     // optional
form.dishId      = UI.getValue("dishId")       // empty if creating new
return form
```

DishEditorPage.validate

```
Input: form: DishForm
Output: “INVALID INFORMATION” | “VALID - SUCCESS”
Main:
if form.name == "" or  form.price == null or form.price <= 0 or form.description == "" or form.imageFile == null AND
form.dishId is empty:
    return “INVALID INFORMATION”
else
    return “VALID - SUCCESS”
```

DishManager.createOrUpdateDish

```
Input: form: DishForm
Output: CreateUpdateResult ("INCOMPLETE" | "UPDATED" | "CREATED")
```

```
Main:
validation = DishEditorPage.validate(form)
if validation.isValid == false:
   return "INCOMPLETE"

// creating a new dish
if form.dishId is empty:
   newDish = DishManager.buildDishRecord(form)
   MenuDatabase.insert(newDish)
   return "CREATED"

// updating an existing dish
existing = MenuDatabase.findById(form.dishId)
if existing == null:
   return "INCOMPLETE"   // unexpected case: invalid ID

updated = DishManager.updateDishRecord(existing, form)
MenuDatabase.update(updated)
return "UPDATED"
```

DishManager.buildDishRecord

```
Input: form: DishForm
Output: DishRecord
Main:
dish.id          = UUID()
dish.name        = form.name
dish.description = form.description
dish.price       = form.price
dish.allergens   = form.allergens
dish.calories    = form.calories
dish.imageUrl    = ImageStore.save(form.imageFile)  // assumed external utility
dish.ratingAvg   = 0
dish.ratingCount = 0
dish.timesOrdered = 0
dish.createdAt   = now()
dish.chefId      = Session.currentUserId()
return dish
```

DishManager.updateDishRecord

```
Input: existing: DishRecord, form: DishForm
Output: DishRecord (updated)
Main:
existing.name  = form.name
existing.description = form.description
existing.price = form.price
existing.allergens = form.allergens
```

existing.calories = form.calories

if form.imageFile != null:
  newUrl = ImageStore.save(form.imageFile)
  existing.imageUrl = newUrl

existing.updatedAt = now()
return existing

MenuDatabase.insert

Input: dish: DishRecord
Output: none
Main:
DB.insert("Menu", dish)

MenuDatabase.update
Input: dish: DishRecord
Output: none
Main:
DB.update("Menu", dish.id, dish)

MenuDatabase.findById
Input: dishId: string
Output: DishRecord | null
Main:
record = DB.findOne("Menu", { id: dishId })
return record or null

ImageStore.save (Assumed Utility)
Input: file
Output: url: string
Main:
path = FileSystem.upload("images/dishes/", file)
return path

13. Assign Delivery

Flow: Assign Delivery Main → OrderSystem.openForBidding → DeliveryBidding.collectBids → Manager.selectDeliveryPerson → DeliveryAssignment.finalize

OrderSystem.openForBidding
Input: orderId: string
Output: BiddingSession
Main:
session.orderId = orderId

```
session.status = "OPEN"
session.bids = []
session.openedAt = now()
session.deadline = now() + 15 minutes  // configurable timeout
DB.insert("BiddingSessions", session)
// notify all active delivery people
DeliveryNotification.broadcastBiddingOpportunity(orderId)
return session
```

DeliveryBidding.collectBids

```
Input: sessionId: string
Output: list<DeliveryBid>
Main:
session = DB.findOne("BiddingSessions", { id: sessionId })
if session == null or session.status != "OPEN":
   return []
bids = DB.findAll("DeliveryBids", { sessionId: sessionId })
// sort by bid amount ascending (lowest first)
sortedBids = sort(bids, by: "bidAmount", order: "ASC")
return sortedBids
```

DeliveryBidding.submitBid

```
Input: deliveryPersonId: string, sessionId: string, bidAmount: decimal
Output: "SUCCESS" | "SESSION_CLOSED" | "INVALID_AMOUNT"
Main:
session = DB.findOne("BiddingSessions", { id: sessionId })
if session == null or session.status != "OPEN":
   return "SESSION_CLOSED"
if bidAmount <= 0:
   return "INVALID_AMOUNT"

bid.id = UUID()
bid.sessionId = sessionId
bid.deliveryPersonId = deliveryPersonId
bid.bidAmount = bidAmount
bid.submittedAt = now()

DB.insert("DeliveryBids", bid)
return "SUCCESS"
```

Manager.selectDeliveryPerson

```
Input: sessionId: string, selectedDeliveryPersonId: string, memo: string | null
Output: "SUCCESS" | "REQUIRES_MEMO" | "SESSION_NOT_FOUND"
Main:
session = DB.findOne("BiddingSessions", { id: sessionId })
if session == null:
   return "SESSION_NOT_FOUND"
```

```
bids = DeliveryBidding.collectBids(sessionId)
lowestBid = bids[0]  // first element after sorting

selectedBid = findBid(bids, selectedDeliveryPersonId)

// if not selecting lowest bidder, memo is required
if selectedBid.deliveryPersonId != lowestBid.deliveryPersonId:
    if memo == null or trim(memo) == "":
        return "REQUIRES_MEMO"

return DeliveryAssignment.finalize(session.orderId, selectedBid, memo)
```

DeliveryAssignment.finalize

```
Input: orderId: string, selectedBid: DeliveryBid, memo: string | null
Output: "SUCCESS"
Main:
assignment.id = UUID()
assignment.orderId = orderId
assignment.deliveryPersonId = selectedBid.deliveryPersonId
assignment.deliveryFee = selectedBid.bidAmount
assignment.assignedAt = now()
assignment.status = "ASSIGNED"

if memo != null:
    assignment.overrideReason = memo
    assignment.wasLowestBid = false
else:
    assignment.overrideReason = null
    assignment.wasLowestBid = true

DB.insert("DeliveryAssignments", assignment)

// update order with delivery info
DB.update("Orders", { id: orderId }, {
    deliveryPersonId: selectedBid.deliveryPersonId,
    deliveryFee: selectedBid.bidAmount,
    status: "ASSIGNED_FOR_DELIVERY"
})

// close bidding session
DB.update("BiddingSessions", { id: selectedBid.sessionId }, {
    status: "CLOSED",
    closedAt: now()
})
```

```
// notify selected delivery person
DeliveryNotification.sendAssignmentConfirmation(selectedBid.deliveryPersonId, orderId)

return "SUCCESS"
```

Helper: findBid

```
Input: bids: list<DeliveryBid>, deliveryPersonId: string
Output: DeliveryBid | null
Main:
for each bid in bids:
   if bid.deliveryPersonId == deliveryPersonId:
      return bid
return null
```

14. Rate Food / Delivery

Flow: Rate Food/Delivery Main → RatingPage.collectRating → RatingSystem.submitRating → RatingSystem.updateAverages → ComplaintSystem.processComplaintOrCompliment (if applicable)

RatingPage.collectRating

```
Input: orderId: string, userId: string
Output: RatingForm
Main:
form.orderId = orderId
form.userId = userId
form.foodRating = UI.getValue("foodRating")  // 1-5 stars
form.deliveryRating = UI.getValue("deliveryRating")  // 1-5 stars
form.feedbackType = UI.getValue("feedbackType")  // "NONE" | "COMPLAINT" | "COMPLIMENT"
form.feedbackTarget = UI.getValue("feedbackTarget")  // "CHEF" | "DELIVERY_PERSON"
form.feedbackText = trim(UI.getValue("feedbackText"))
return form
```

RatingPage.handleSubmit

```
Input: form: RatingForm
Output: "SUCCESS" | "INVALID_RATING" | "ORDER_NOT_FOUND" | "ALREADY_RATED"
Main
validation = RatingPage.validateRating(form)
if validation.isValid == false:
   return "INVALID_RATING"

return RatingSystem.submitRating(form)
```

RatingPage.validateRating

```
Input: form: RatingForm
Output: ValidationResult { isValid: bool, errors: map }
Main:
```

```
result.isValid = true
if form.foodRating < 1 or form.foodRating > 5:
   result.isValid = false
   result.errors["foodRating"] = "must be 1-5 stars"
if form.deliveryRating < 1 or form.deliveryRating > 5:
   result.isValid = false
   result.errors["deliveryRating"] = "must be 1-5 stars"
if form.feedbackType != "NONE":
   if form.feedbackTarget == "":
      result.isValid = false
      result.errors["feedbackTarget"] = "must specify target"
   if form.feedbackText == "":
      result.isValid = false
      result.errors["feedbackText"] = "feedback text required"
return result
```

RatingSystem.submitRating

Input: form: RatingForm

Output: "SUCCESS" | "ORDER_NOT_FOUND" | "ALREADY_RATED"

Main:

```
// verify order exists and belongs to user
order = DB.findOne("Orders", { id: form.orderId, customerId: form.userId })
if order == null:
   return "ORDER_NOT_FOUND"

// check if already rated
existingRating = DB.findOne("Ratings", { orderId: form.orderId })
if existingRating != null:
   return "ALREADY_RATED"

// create rating record
rating.id = UUID()
rating.orderId = form.orderId
rating.customerId = form.userId
rating.chefId = order.chefId
rating.deliveryPersonId = order.deliveryPersonId
rating.dishId = order.dishId
rating.foodRating = form.foodRating
rating.deliveryRating = form.deliveryRating
rating.createdAt = now()
DB.insert("Ratings", rating)

// update averages for dish and delivery person
RatingSystem.updateDishRating(order.dishId, form.foodRating)
RatingSystem.updateDeliveryPersonRating(order.deliveryPersonId, form.deliveryRating)
RatingSystem.updateChefRating(order.chefId, form.foodRating)
```

```
// handle complaint or compliment if provided
if form.feedbackType != "NONE":
    ComplaintSystem.processComplaintOrCompliment(form, order)

return "SUCCESS"
```

RatingSystem.updateDishRating

```
Input: dishId: string, newRating: int
Output: none
Main:
dish = DB.findOne("Dishes", { id: dishId })
if dish == null:
    return

currentAvg = dish.averageRating
currentCount = dish.ratingCount
// calculate new average
newCount = currentCount + 1
newAvg = ((currentAvg * currentCount) + newRating) / newCount
DB.update("Dishes", { id: dishId }, {
    averageRating: newAvg,
    ratingCount: newCount
})
```

RatingSystem.updateDeliveryPersonRating

```
Input: deliveryPersonId: string, newRating: int
Output: none
Main:
deliveryPerson = DB.findOne("Employees", { id: deliveryPersonId, role: "DELIVERY_PERSON" })
if deliveryPerson == null:
    return

currentAvg = deliveryPerson.averageRating
currentCount = deliveryPerson.ratingCount
newCount = currentCount + 1
newAvg = ((currentAvg * currentCount) + newRating) / newCount
DB.update("Employees", { id: deliveryPersonId }, {
    averageRating: newAvg,
    ratingCount: newCount
})
```

RatingSystem.updateChefRating

```
Input: chefId: string, newRating: int
Output: none
Main:
```

```
chef = DB.findOne("Employees", { id: chefId, role: "CHEF" })
if chef == null:
    return

currentAvg = chef.averageRating
currentCount = chef.ratingCount
newCount = currentCount + 1
newAvg = ((currentAvg * currentCount) + newRating) / newCount
DB.update("Employees", { id: chefId }, {
    averageRating: newAvg,
    ratingCount: newCount
})

// check for automatic demotion or bonus triggers
if newAvg < 2 and newCount >= 10:  // minimum sample size
    HRSystem.flagForDemotion(chefId, "LOW_RATING")
else if newAvg > 4 and newCount >= 10:
    HRSystem.flagForBonus(chefId, "HIGH_RATING")
```

ComplaintSystem.processComplaintOrCompliment

```
Input: form: RatingForm, order: Order
Output: none
Main:
report.id = UUID()
report.reporterId = form.userId
report.orderId = form.orderId
report.type = form.feedbackType  // "COMPLAINT" | "COMPLIMENT"
report.feedbackText = form.feedbackText
report.status = "PENDING"
report.createdAt = now()

// determine target of feedback
if form.feedbackTarget == "CHEF":
    report.targetId = order.chefId
    report.targetType = "CHEF"
else if form.feedbackTarget == "DELIVERY_PERSON":
    report.targetId = order.deliveryPersonId
    report.targetType = "DELIVERY_PERSON"

// apply VIP weight multiplier
reporter = DB.findOne("Users", { id: form.userId })
if reporter.accountType == "VIP":
    report.weight = 2
else:
    report.weight = 1
```

```
DB.insert("Reports", report)

// notify manager of new report
ManagerNotification.sendReportAlert(report.id)
```

15. Start / Join Discussion

Flow: Discussion Main → DiscussionPage.selectTopic → DiscussionSystem.findOrCreateThread → DiscussionPage.submitMessage → DiscussionSystem.postMessage

DiscussionPage.selectTopic

```
Input: none
Output: TopicSelection
Main:
selection.topicType = UI.getValue("topicType")  // "CHEF" | "DISH" | "DELIVERY_PERSON"
selection.topicId = UI.getValue("topicId")  // id of selected chef/dish/delivery person
return selection
```

DiscussionSystem.findOrCreateThread

```
Input: topicType: string, topicId: string
Output: threadId: string
Main:
thread = DB.findOne("DiscussionThreads", { topicType: topicType, topicId: topicId })
if thread != null:
    return thread.id
// create new thread
newThread.id = UUID()
newThread.topicType = topicType
newThread.topicId = topicId
newThread.createdAt = now()
newThread.messageCount = 0
DB.insert("DiscussionThreads", newThread)
return newThread.id
```

DiscussionPage.submitMessage

```
Input: threadId: string, userId: string
Output: "SUCCESS" | "EMPTY_MESSAGE" | "THREAD_NOT_FOUND"
Main:
messageText = trim(UI.getValue("messageText"))
if messageText == "":
    return "EMPTY_MESSAGE"
return DiscussionSystem.postMessage(threadId, userId, messageText)
```

DiscussionSystem.postMessage

```
Input: threadId: string, userId: string, messageText: string
Output: "SUCCESS" | "THREAD_NOT_FOUND"
Main:
```

```
thread = DB.findOne("DiscussionThreads", { id: threadId })
if thread == null:
    return "THREAD_NOT_FOUND"

message.id = UUID()
message.threadId = threadId
message.userId = userId
message.messageText = messageText
message.postedAt = now()
message.flagged = false
DB.insert("DiscussionMessages", message)

// update thread message count
DB.update("DiscussionThreads", { id: threadId }, {
    messageCount: thread.messageCount + 1,
    lastActivityAt: now()
})

return "SUCCESS"
```

DiscussionSystem.getThreadMessages
Input: threadId: string
Output: list<Message>
Main:

```
messages = DB.findAll("DiscussionMessages", { threadId: threadId })
sortedMessages = sort(messages, by: "postedAt", order: "ASC")
return sortedMessages
```

DiscussionSystem.flagMessage
Input: messageId: string, reporterId: string, reason: string
Output: "SUCCESS"
Main:

```
flag.id = UUID()
flag.messageId = messageId
flag.reporterId = reporterId
flag.reason = reason
flag.createdAt = now()
flag.status = "PENDING"
DB.insert("MessageFlags", flag)

// update message flagged status
DB.update("DiscussionMessages", { id: messageId }, { flagged: true })

// notify manager for moderation
ManagerNotification.sendModerationAlert(messageId)
return "SUCCESS"
```

16. Hire Employee

Flow: Hire Employee Main → HiringPage.collectEmployeeInfo → HRSystem.checkBlacklist → HRSystem.createEmployeeAccount

HiringPage.collectEmployeeInfo

Input: none

Output: EmployeeForm

Main:

```
form.name = trim(UI.getValue("name"))
form.email = lowerCase(trim(UI.getValue("email")))
form.phone = trim(UI.getValue("phone"))
form.role = UI.getValue("role")  // "CHEF" | "DELIVERY_PERSON"
form.salary = UI.getValue("salary")
return form
```

HiringPage.handleSubmit

Input: form: EmployeeForm

Output: "SUCCESS" | "INCOMPLETE" | "BLACKLISTED" | "DUPLICATE"

Main:

```
validation = HiringPage.validateForm(form)
if validation.isValid == false:
   return "INCOMPLETE"
return HRSystem.processHiring(form)
```

HiringPage.validateForm

Input: form: EmployeeForm

Output: ValidationResult { isValid: bool, errors: map }

Main:

```
result.isValid = true
if form.name == "" or form.email == "" or form.phone == "":
   result.isValid = false
   result.errors["required"] = "all fields required"
if form.role != "CHEF" and form.role != "DELIVERY_PERSON":
   result.isValid = false
   result.errors["role"] = "invalid role"
if form.salary <= 0:
   result.isValid = false
   result.errors["salary"] = "must be positive"
return result
```

HRSystem.processHiring

Input: form: EmployeeForm

Output: "SUCCESS" | "BLACKLISTED" | "DUPLICATE"

Main:

```
// check blacklist
isBlacklisted = HRSystem.checkBlacklist(form.email)
if isBlacklisted:
   return "BLACKLISTED"

// check for duplicate employee
existing = DB.findOne("Employees", { email: form.email })
if existing != null:
   return "DUPLICATE"

HRSystem.createEmployeeAccount(form)
return "SUCCESS"
```

HRSystem.checkBlacklist

```
Input: email: string
Output: bool
Main:
blacklisted = DB.findOne("Blacklist", { email: email })
if blacklisted != null:
   return true
return false
```

HRSystem.createEmployeeAccount

```
Input: form: EmployeeForm
Output: employeeId: string
Main:
employee.id = UUID()
employee.name = form.name
employee.email = form.email
employee.phone = form.phone
employee.role = form.role
employee.salary = form.salary
employee.username = HRSystem.generateUsername(form.name)
employee.password = Security.generateTemporaryPassword()
employee.hiredAt = now()
employee.status = "ACTIVE"
employee.averageRating = 0
employee.ratingCount = 0
employee.warningCount = 0
employee.complimentCount = 0
employee.demotionCount = 0

DB.insert("Employees", employee)
// send credentials to new employee
EmployeeNotification.sendWelcomeEmail(employee.email, employee.username, employee.password)
```

```
return employee.id
```

HRSystem.generateUsername

```
Input: name: string
Output: username: string
Main:
base = lowerCase(replace(name, " ", ""))
counter = 1
username = base

// ensure uniqueness
while DB.findOne("Employees", { username: username }) != null:
    username = base + toString(counter)
    counter = counter + 1
return username
```

17. Grant Bonus

Flow: Grant Bonus Main → HRSystem.identifyEligibleEmployees → ManagerPage.selectEmployee → HRSystem.grantBonus

HRSystem.identifyEligibleEmployees

```
Input: none
Output: list<Employee>
Main:
eligibleList = []

// find employees with high ratings (>4 stars)
highRatedEmployees = DB.findAll("Employees", { averageRating: >4, ratingCount: >=10 })
eligibleList = eligibleList + highRatedEmployees

// find employees with 3 compliments
complimentedEmployees = DB.findAll("Employees", { complimentCount: >=3 })

eligibleList = eligibleList + complimentedEmployees
// remove duplicates
eligibleList = removeDuplicates(eligibleList)

return eligibleList
```

ManagerPage.selectEmployee

```
Input: none
Output: BonusForm
Main:
form.employeeId = UI.getValue("employeeId")
form.bonusAmount = UI.getValue("bonusAmount")
```

```
form.reason = trim(UI.getValue("reason"))
return form
```

ManagerPage.handleSubmit

```
Input: form: BonusForm
Output: "SUCCESS" | "INVALID_AMOUNT" | "EMPLOYEE_NOT_FOUND"
Main:
if form.bonusAmount <= 0:
   return "INVALID_AMOUNT"
return HRSystem.grantBonus(form)
```

HRSystem.grantBonus

```
Input: form: BonusForm
Output: "SUCCESS" | "EMPLOYEE_NOT_FOUND"
Main:
employee = DB.findOne("Employees", { id: form.employeeId })
if employee == null:
   return "EMPLOYEE_NOT_FOUND"

// create bonus record
bonus.id = UUID()
bonus.employeeId = form.employeeId
bonus.amount = form.bonusAmount
bonus.reason = form.reason
bonus.grantedAt = now()
bonus.grantedBy = currentManagerId  // assumed context
DB.insert("Bonuses", bonus)
// update payroll
PayrollSystem.addBonusToNextCycle(form.employeeId, form.bonusAmount)

// reset compliment count if bonus was for compliments
if employee.complimentCount >= 3:
   DB.update("Employees", { id: form.employeeId }, {
      complimentCount: 0
   })

// log HR action
HRSystem.logAction("GRANT_BONUS", form.employeeId, form.bonusAmount, form.reason)
// notify employee
EmployeeNotification.sendBonusNotification(employee.email, form.bonusAmount)
return "SUCCESS"
```

PayrollSystem.addBonusToNextCycle

```
Input: employeeId: string, bonusAmount: decimal
Output: none
Main:
```

```
currentCycle = PayrollSystem.getCurrentCycle()

payroll = DB.findOne("Payroll", { employeeId: employeeId, cycle: currentCycle })
if payroll == null:
   payroll = PayrollSystem.createPayrollEntry(employeeId, currentCycle)

DB.update("Payroll", { id: payroll.id }, {
   bonusAmount: payroll.bonusAmount + bonusAmount
})
```

HRSystem.logAction

Input: actionType: string, employeeId: string, amount: decimal, reason: string

Output: none

Main:

```
log.id = UUID()
log.actionType = actionType
log.employeeId = employeeId
log.amount = amount
log.reason = reason
log.performedBy = currentManagerId  // assumed context
log.performedAt = now()

DB.insert("HRActionLog", log)
```

18. Demote Employee

Flow: Demote Employee Main → HRSystem.identifyEmployeesForDemotion → ManagerPage.selectEmployeeForDemotion → HRSystem.demoteEmployee → (if second demotion) → HRSystem.terminateEmployee

HRSystem.identifyEmployeesForDemotion

Input: none

Output: list<Employee>

Main:

```
eligibleList = []

// find employees with low ratings (<2 stars)
lowRatedEmployees = DB.findAll("Employees", { averageRating: <2, ratingCount: >=10 })
eligibleList = eligibleList + lowRatedEmployees

// find employees with 3 warnings
warnedEmployees = DB.findAll("Employees", { warningCount: >=3 })
eligibleList = eligibleList + warnedEmployees
eligibleList = removeDuplicates(eligibleList)

return eligibleList
```

ManagerPage.selectEmployeeForDemotion
Input: none
Output: DemotionForm
Main:
```
form.employeeId = UI.getValue("employeeId")
form.reason = trim(UI.getValue("reason"))
form.newSalary = UI.getValue("newSalary")
return form
```

ManagerPage.handleDemotionSubmit
Input: form: DemotionForm
Output: "SUCCESS" | "EMPLOYEE_NOT_FOUND" | "INVALID_SALARY"
Main:
```
if form.newSalary <= 0:
   return "INVALID_SALARY"
return HRSystem.demoteEmployee(form)
```

HRSystem.demoteEmployee
Input: form: DemotionForm
Output: "SUCCESS" | "EMPLOYEE_NOT_FOUND" | "STAFFING_CONSTRAINT"
Main:
```
employee = DB.findOne("Employees", { id: form.employeeId })
if employee == null:
   return "EMPLOYEE_NOT_FOUND"

// check if this is second demotion (would trigger firing)
if employee.demotionCount >= 1:
   // check minimum staffing constraint
   canFire = HRSystem.checkStaffingConstraint(employee.role)
   if canFire == false:
      return "STAFFING_CONSTRAINT"
   // proceed with termination
   return HRSystem.terminateEmployee(form.employeeId, "SECOND_DEMOTION")

// first demotion - reduce salary
DB.update("Employees", { id: form.employeeId }, {
   salary: form.newSalary,
   demotionCount: employee.demotionCount + 1
})

// reset warning count after demotion
if employee.warningCount >= 3:
   DB.update("Employees", { id: form.employeeId }, {
      warningCount: 0
   })
```

```
// log HR action
HRSystem.logAction("DEMOTE_EMPLOYEE", form.employeeId, form.newSalary, form.reason)
// notify employee
EmployeeNotification.sendDemotionNotification(employee.email, form.newSalary, form.reason)
return "SUCCESS"
```

HRSystem.checkStaffingConstraint

```
Input: role: string
Output: bool
Main:
// count active employees in the same role
activeCount = DB.count("Employees", { role: role, status: "ACTIVE" })

// must have more than 2 to allow firing
if activeCount > 2:
   return true
return false
```

19. Adjust Pay

Flow: Adjust Pay Main → HRSystem.identifyPayAdjustmentCandidates → ManagerPage.selectEmployeeForPayAdjust → HRSystem.adjustPay

HRSystem.identifyPayAdjustmentCandidates

```
Input: none
Output: list<Employee>
Main:
candidates = []

// employees flagged for performance-based adjustments
performanceCandidates = DB.findAll("Employees", { performanceReviewDue: true })
candidates = candidates + performanceCandidates

// employees with pending pay adjustment requests
requestedAdjustments = DB.findAll("PayAdjustmentRequests", { status: "PENDING" })
for each request in requestedAdjustments:
   employee = DB.findOne("Employees", { id: request.employeeId })
   candidates = candidates + [employee]
candidates = removeDuplicates(candidates)
return candidates
```

ManagerPage.selectEmployeeForPayAdjust

```
Input: none
```

Output: PayAdjustmentForm
Main:
form.employeeId = UI.getValue("employeeId")
form.newSalary = UI.getValue("newSalary")
form.adjustmentType = UI.getValue("adjustmentType") // "RAISE" | "CUT"
form.justification = trim(UI.getValue("justification"))
return form

ManagerPage.handlePayAdjustSubmit
Input: form: PayAdjustmentForm
Output: "SUCCESS" | "INVALID_AMOUNT" | "REQUIRES_HR_APPROVAL" | "EMPLOYEE_NOT_FOUND"
Main:
if form.newSalary <= 0:
  return "INVALID_AMOUNT"
return HRSystem.adjustPay(form)

HRSystem.adjustPay
Input: form: PayAdjustmentForm
Output: "SUCCESS" | "EMPLOYEE_NOT_FOUND" | "REQUIRES_HR_APPROVAL" | "BELOW_MINIMUM"
Main:
employee = DB.findOne("Employees", { id: form.employeeId })
if employee == null:
  return "EMPLOYEE_NOT_FOUND"

// validate new salary meets minimum wage
if form.newSalary < MINIMUM_WAGE: // assumed constant
  return "BELOW_MINIMUM"
// calculate percentage change
percentChange = ((form.newSalary - employee.salary) / employee.salary) * 100
// check if requires HR approval (increase > 10%)
if form.adjustmentType == "RAISE" and percentChange > 10:
  HRSystem.sendForHRApproval(form)
  return "REQUIRES_HR_APPROVAL"

// update employee salary
DB.update("Employees", { id: form.employeeId }, {
  salary: form.newSalary,
  lastPayAdjustment: now()
})

// update payroll for next cycle
PayrollSystem.updateSalaryForNextCycle(form.employeeId, form.newSalary)
// log HR action
HRSystem.logAction("ADJUST_PAY", form.employeeId, form.newSalary, form.justification)
// notify employee

EmployeeNotification.sendPayAdjustmentNotification(employee.email, form.newSalary, form.adjustmentType)
return "SUCCESS"

HRSystem.sendForHRApproval
Input: form: PayAdjustmentForm
Output: none
Main:
approval.id = UUID()
approval.employeeId = form.employeeId
approval.requestedSalary = form.newSalary
approval.justification = form.justification
approval.requestedBy = currentManagerId // assumed context
approval.requestedAt = now()
approval.status = "PENDING_HR_APPROVAL"
DB.insert("HRApprovalRequests", approval)

// notify HR department
HRNotification.sendApprovalRequest(approval.id)

PayrollSystem.updateSalaryForNextCycle
Input: employeeId: string, newSalary: decimal
Output: none
Main:
currentCycle = PayrollSystem.getCurrentCycle()
payroll = DB.findOne("Payroll", { employeeId: employeeId, cycle: currentCycle })
if payroll == null:
  payroll = PayrollSystem.createPayrollEntry(employeeId, currentCycle)
DB.update("Payroll", { id: payroll.id }, {
  baseSalary: newSalary
})

20. Terminate/Fire Employee

Flow: Terminate/Fire Employee Main → HRSystem.checkStaffingConstraint → ManagerPage.confirmTermination → HRSystem.terminateEmployee

HRSystem.identifyEmployeesForTermination
Input: none
Output: list<Employee>
Main:
candidates = []

// employees with 2 demotions (automatic termination trigger)
twoDemotions = DB.findAll("Employees", { demotionCount: >=2 })
candidates = candidates + twoDemotions
return candidates

ManagerPage.selectEmployeeForTermination
Input: none
Output: TerminationForm
Main:
form.employeeId = UI.getValue("employeeId")
form.reason = trim(UI.getValue("reason"))
return form

ManagerPage.handleTerminationSubmit
Input: form: TerminationForm
Output: "SUCCESS" | "EMPLOYEE_NOT_FOUND" | "STAFFING_CONSTRAINT"
Main:
return HRSystem.terminateEmployee(form.employeeId, form.reason)

HRSystem.terminateEmployee
Input: employeeId: string, reason: string
Output: "SUCCESS" | "EMPLOYEE_NOT_FOUND" | "STAFFING_CONSTRAINT"
Main:
employee = DB.findOne("Employees", { id: employeeId })
if employee == null:
  return "EMPLOYEE_NOT_FOUND"

// check minimum staffing constraint
canTerminate = HRSystem.checkStaffingConstraint(employee.role)
if canTerminate == false:
  return "STAFFING_CONSTRAINT"

// deactivate employee account
DB.update("Employees", { id: employeeId }, {
  status: "TERMINATED",
  terminatedAt: now(),
  terminationReason: reason
})

// add to blacklist
blacklist.id = UUID()
blacklist.email = employee.email
blacklist.name = employee.name
blacklist.addedAt = now()
blacklist.reason = "TERMINATED_EMPLOYEE"
DB.insert("Blacklist", blacklist)

// remove from current payroll
PayrollSystem.removeFromPayroll(employeeId)
// log HR action
HRSystem.logAction("TERMINATE_EMPLOYEE", employeeId, 0, reason)

```
// notify employee
EmployeeNotification.sendTerminationNotification(employee.email, reason)
return "SUCCESS"
```

PayrollSystem.removeFromPayroll

```
Input: employeeId: string
Output: none
Main:
currentCycle = PayrollSystem.getCurrentCycle()

// finalize any pending payments for current cycle
payroll = DB.findOne("Payroll", { employeeId: employeeId, cycle: currentCycle })
if payroll != null:
   DB.update("Payroll", { id: payroll.id }, {
      status: "FINALIZED",
      finalizedAt: now()
   })

// mark no future payroll entries should be created
DB.update("Employees", { id: employeeId }, {
   payrollActive: false
})
```

21. Close Account

Flow: Close Account Main → ManagerPage.viewPendingClosures → ManagerPage.selectAccountForClosure → AccountSystem.closeAccount

ManagerPage.viewPendingClosures

```
Input: none
Output: list<AccountClosureRequest>
Main:
pendingClosures = DB.findAll("AccountClosureRequests", { status: "PENDING" })
return pendingClosures
```

ManagerPage.selectAccountForClosure

```
Input: none
Output: ClosureForm
Main:
form.requestId = UI.getValue("requestId")
form.customerId = UI.getValue("customerId")
form.closureReason = UI.getValue("closureReason")  // "VOLUNTARY" | "FORCED_DEREGISTRATION"
return form
```

ManagerPage.handleClosureSubmit

```
Input: form: ClosureForm
Output: "SUCCESS" | "CUSTOMER_NOT_FOUND" | "REQUEST_NOT_FOUND"
Main:
```

```
return AccountSystem.closeAccount(form)
```

AccountSystem.closeAccount

```
Input: form: ClosureForm
Output: "SUCCESS" | "CUSTOMER_NOT_FOUND" | "REQUEST_NOT_FOUND"
Main:
customer = DB.findOne("Users", { id: form.customerId })
if customer == null:
   return "CUSTOMER_NOT_FOUND"
request = DB.findOne("AccountClosureRequests", { id: form.requestId })
if request == null:
   return "REQUEST_NOT_FOUND"

// clear deposit balance
AccountSystem.clearDeposit(form.customerId)
// deactivate account
DB.update("Users", { id: form.customerId }, {
   status: "CLOSED",
   closedAt: now(),
   closureReason: form.closureReason
})

// if forced deregistration, add to blacklist
if form.closureReason == "FORCED_DEREGISTRATION":
   AccountSystem.addToBlacklist(customer.email, customer.name)
// mark request as completed
DB.update("AccountClosureRequests", { id: form.requestId }, {
   status: "COMPLETED",
   completedAt: now(),
   completedBy: currentManagerId  // assumed context
})

// notify customer
CustomerNotification.sendAccountClosureConfirmation(customer.email, form.closureReason)
return "SUCCESS"
```

AccountSystem.clearDeposit

```
Input: customerId: string
Output: none
Main:
customer = DB.findOne("Users", { id: customerId })
if customer == null:
   return

depositAmount = customer.balance
```

depositLog.id = UUID()
depositLog.customerId = customerId
depositLog.amount = depositAmount
depositLog.action = "CLEARED"
depositLog.reason = "ACCOUNT_CLOSURE"
depositLog.processedAt = now()
DB.insert("DepositLog", depositLog)

// update customer balance to zero
DB.update("Users", { id: customerId }, {
  balance: 0
})

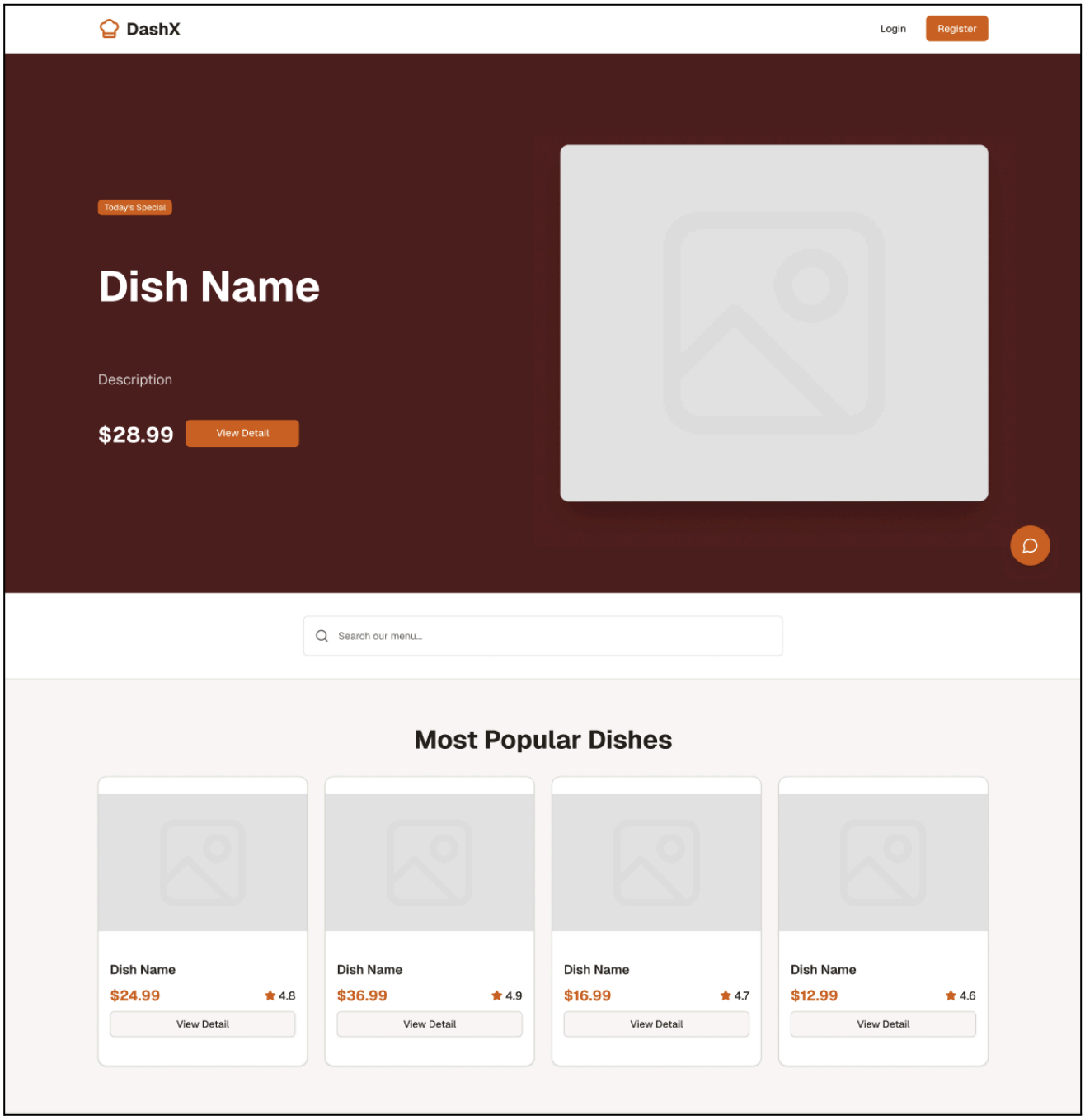
AccountSystem.addToBlacklist
Input: email: string, name: string
Output: none
Main:
blacklist.id = UUID()
blacklist.email = email
blacklist.name = name
blacklist.addedAt = now()
blacklist.reason = "FORCED_DEREGISTRATION_3_WARNINGS"
DB.insert("Blacklist", blacklist)

AccountSystem.createClosureRequest
Input: customerId: string, reason: string
Output: requestId: string
Main:
request.id = UUID()
request.customerId = customerId
request.reason = reason
request.status = "PENDING"
request.requestedAt = now()
DB.insert("AccountClosureRequests", request)

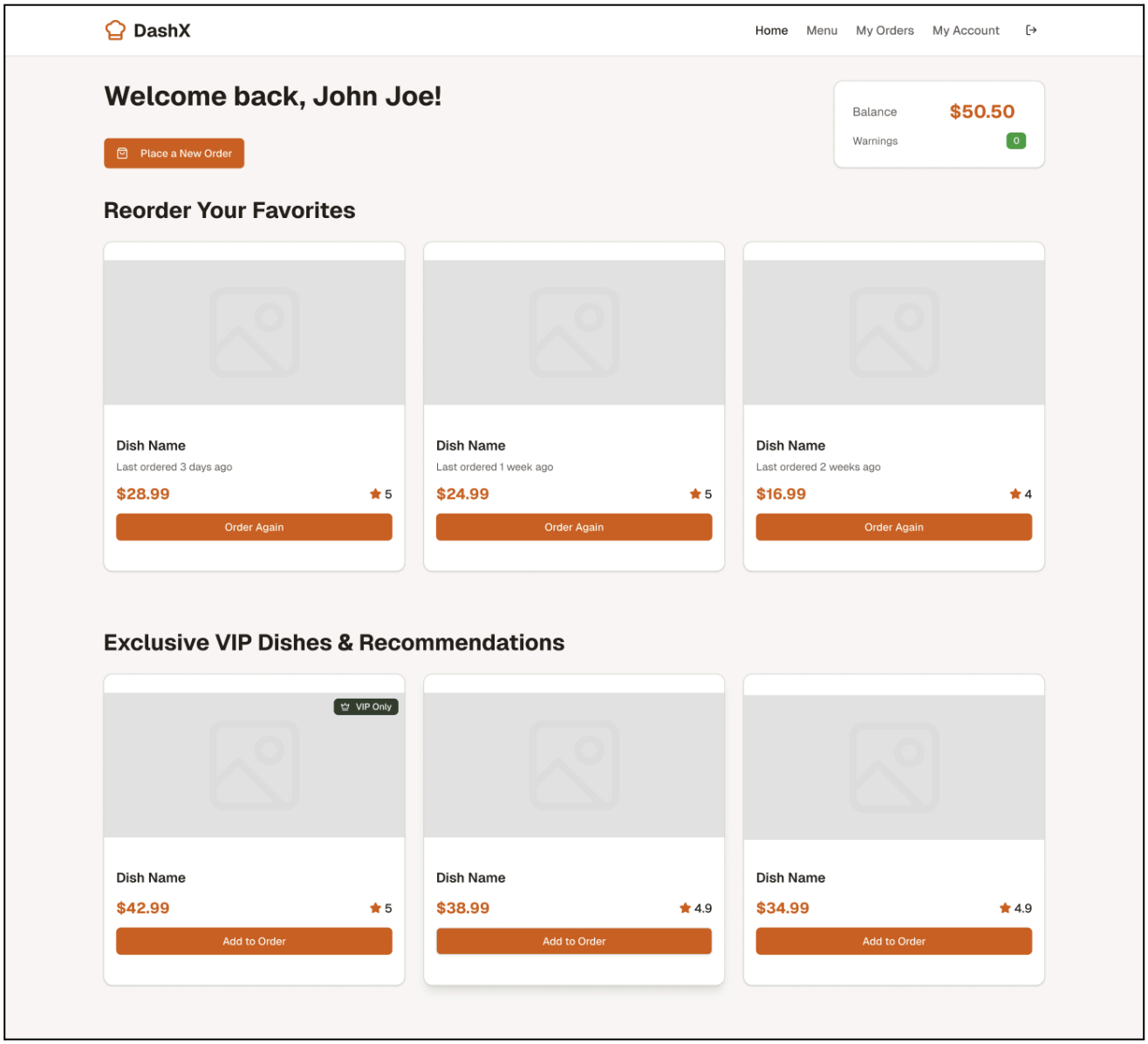
// notify manager of pending closure
ManagerNotification.sendClosureRequestAlert(request.id)
return request.id

5. System screens

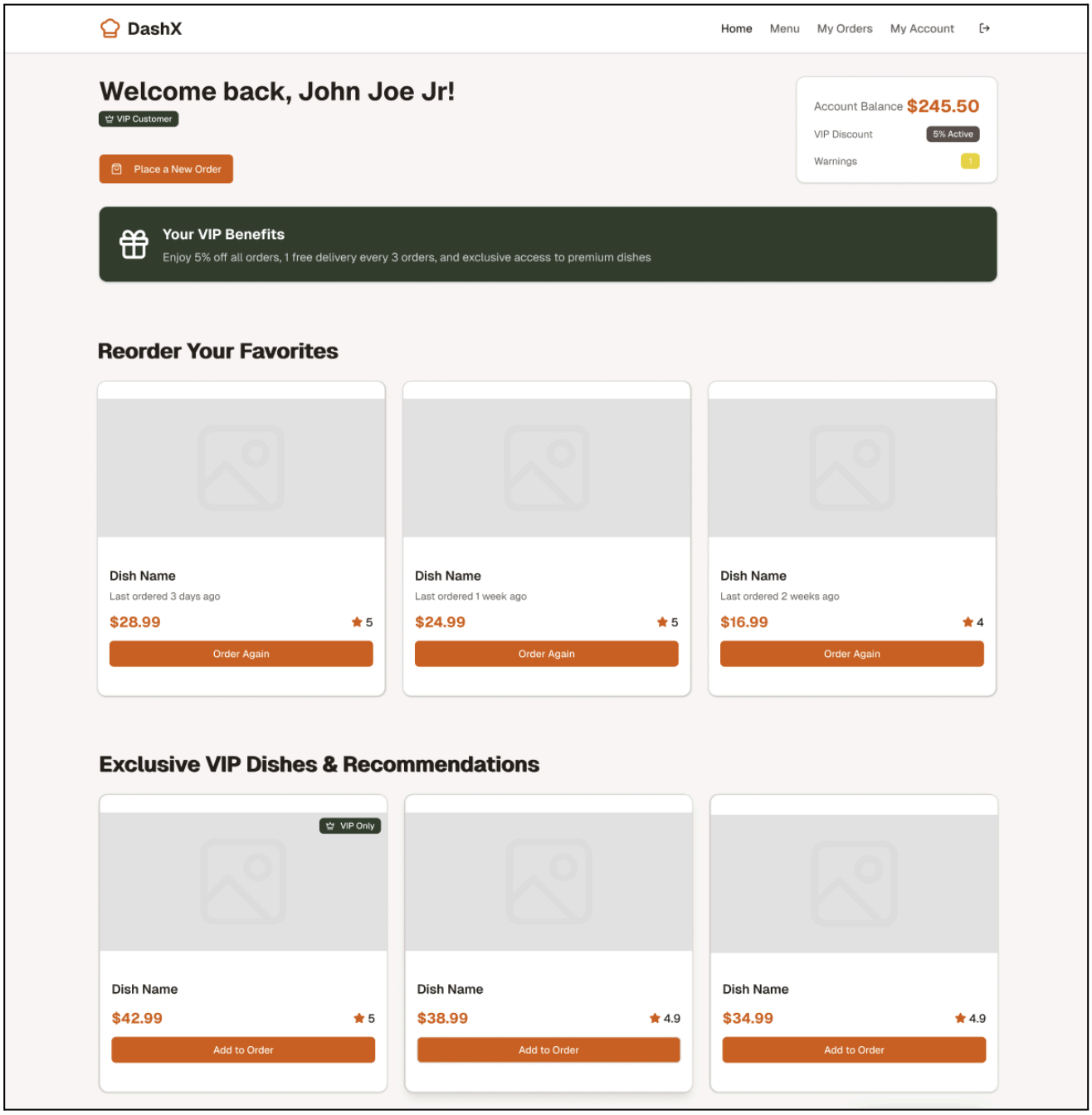
Visitor Screen



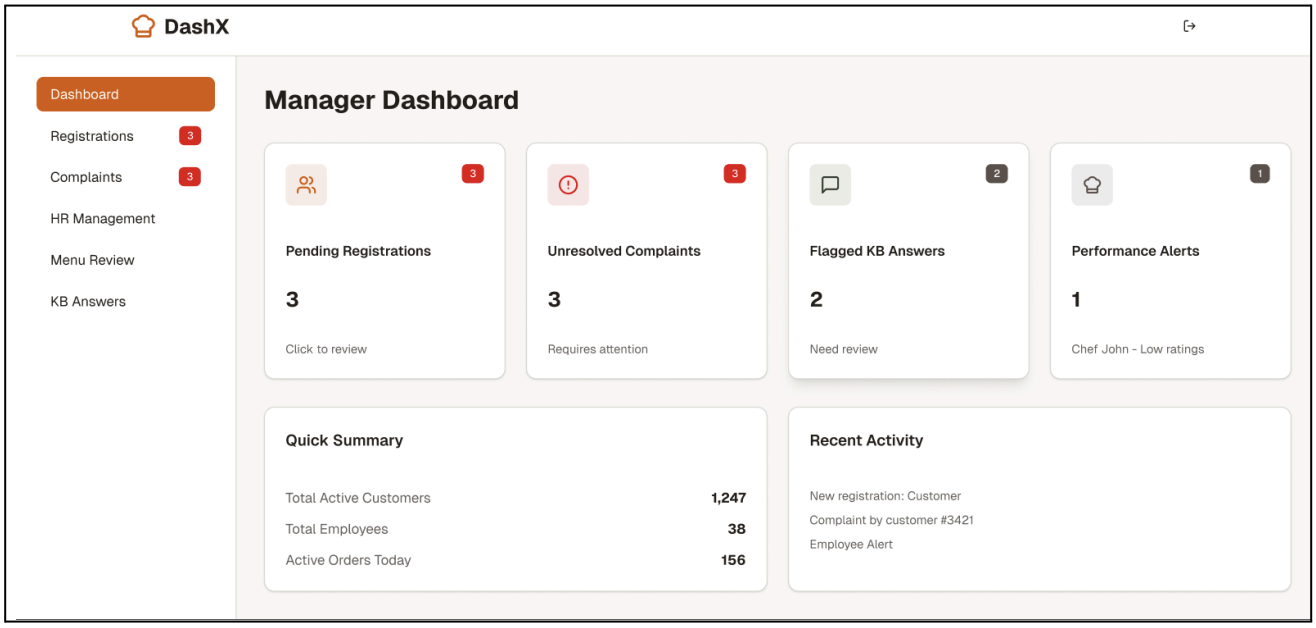
Customer Screen (Non-VIP)



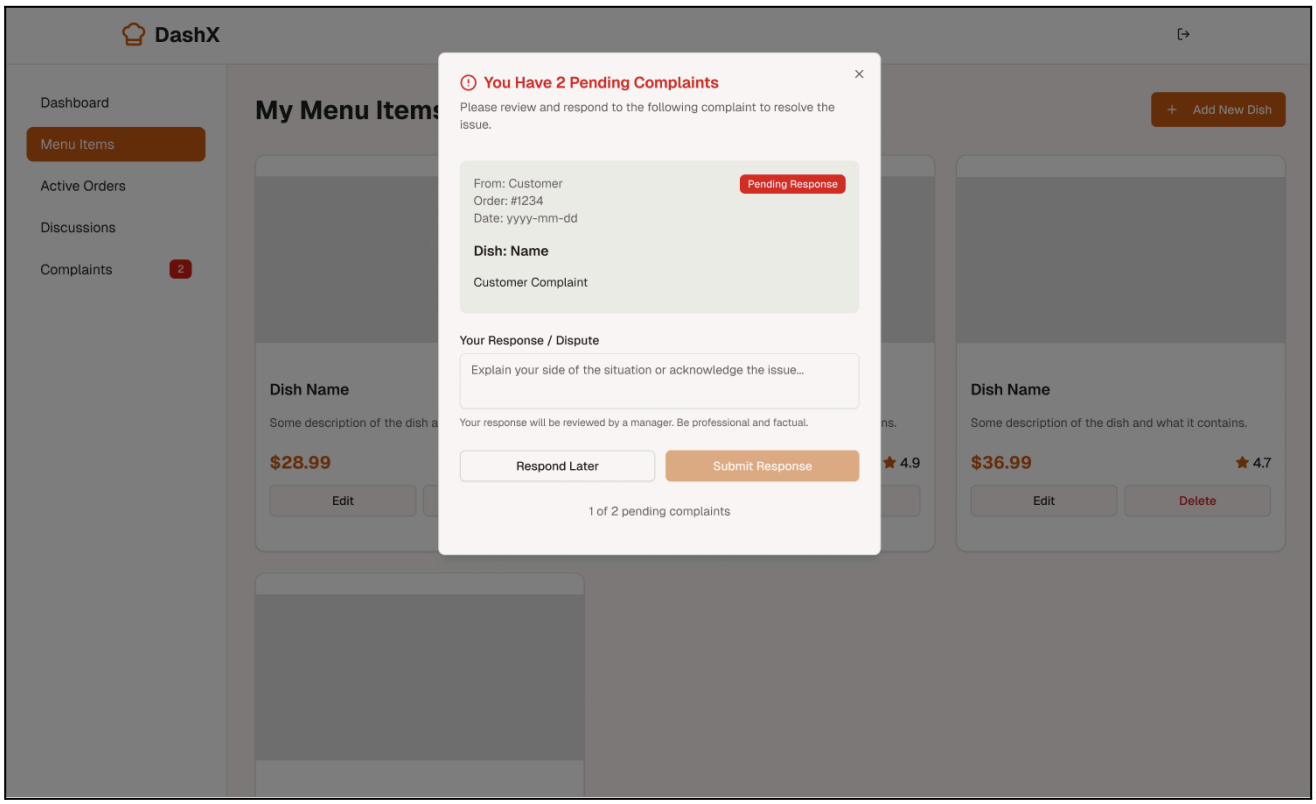
Customer Screen (VIP)

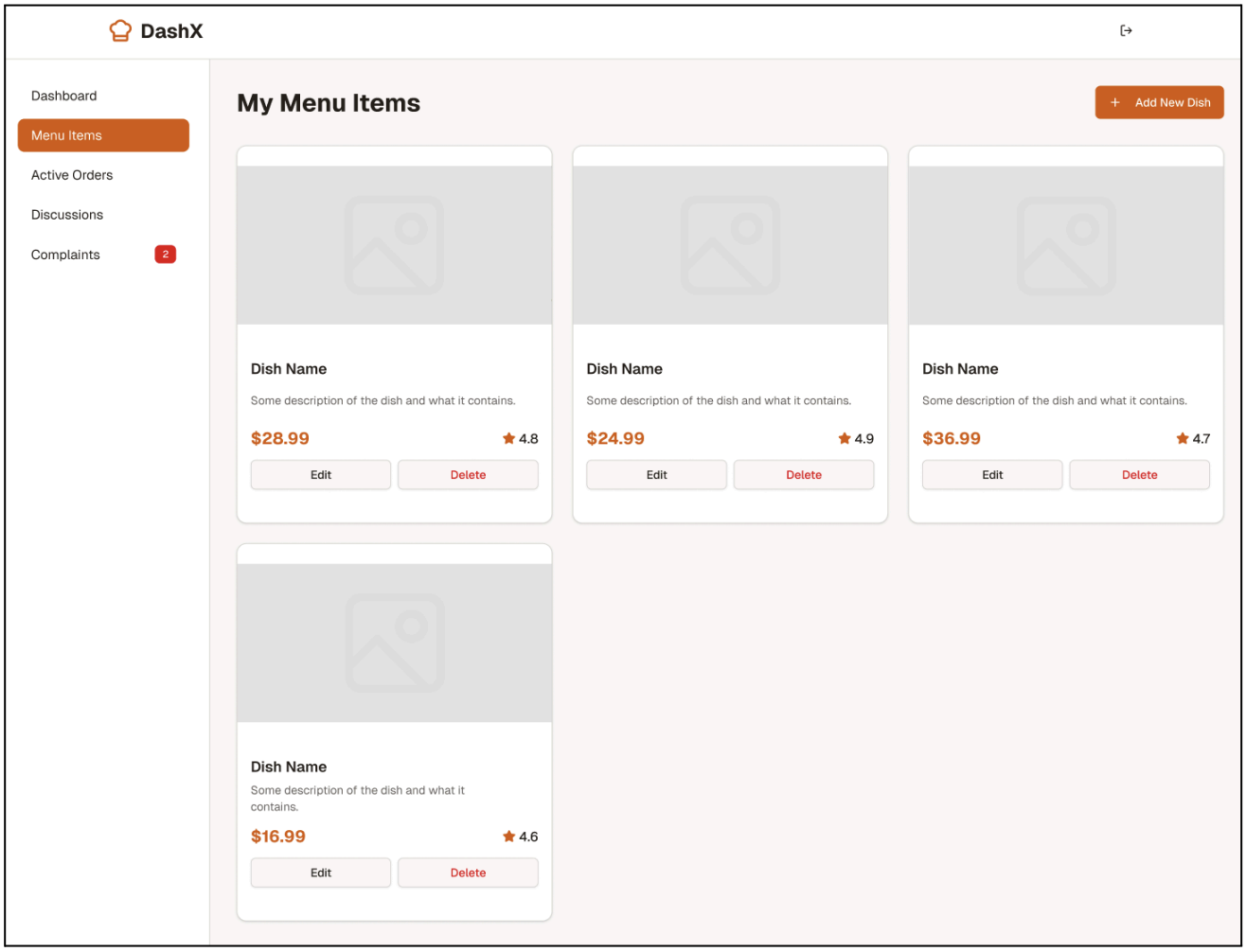


Manager Screen

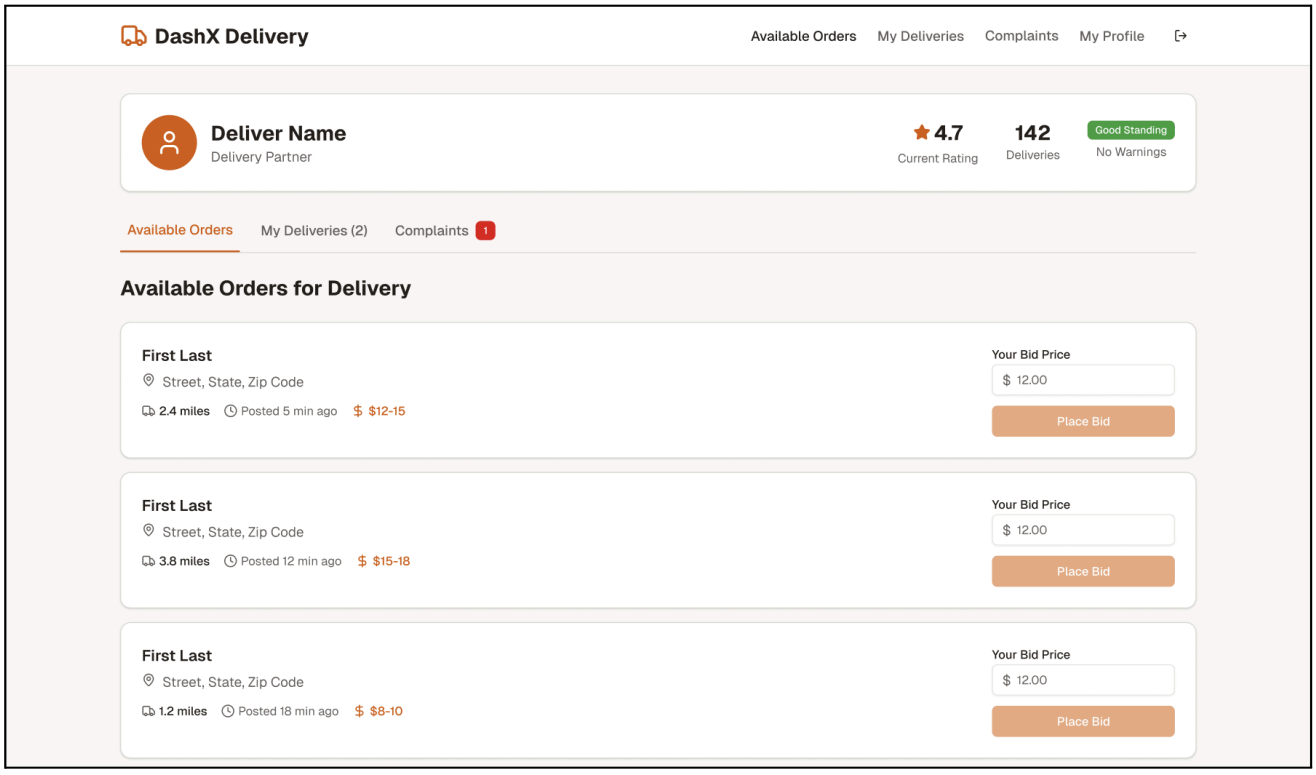


Chef Screen (w/ complaint prototype function)





Delivery Personnel Screen



6. Memos of Group Meetings

Meeting 11/8/2025

- Look through the requirements for report 2
- Jacob will be busy on upcoming weekends
- Joseph has important hackathons.
- Both will do less work on report, more work on implementation
- Assign sections for each group member
    - Bryan → Work on diagrams for parts 1-3 with help on part 2
    - Joseph and Jacob → Detailed Design
    - Yuki → GUI interface and setup Git Repo, help with part 2 on diagram
- Notify team member
- Will hold another meeting close to report 2 deadline

Meeting 11/15/2025

- Check in on how the work progress is for everyone
- Work is almost done, will need to review work
- Work on git repo and will upload all materials required.
- Will be ready to work on the implementation of the app once the report is finalized.

7. Address of the git repo (github, gitlab, bitbucket, etc)

GIT LINK: *https://github.com/yukil-30/DashX*