```
METHOD registerCustomer(formData)
   // Input: formData (name, email, password, address, phone)
   // Output: customerId OR errorCode
   VALIDATE required fields
   IF email EXISTS IN UserTable THEN RETURN EMAIL_ALREADY_EXISTS
   HASH password
   CREATE Customer(name, email, hashedPassword, address, phone,
               role="Customer", balance=0, totalSpent=0,
               orderCount=0, status="PendingApproval",
               warningCount=0, isVIP=false)
   SAVE Customer
   RETURN Customer.id
END


METHOD loginUser(email, password)
   // Input: email, password
   // Output: sessionToken OR errorCode
   user = FIND User BY email
   IF user IS NULL THEN RETURN INVALID_CREDENTIALS
   IF user.isBlacklisted THEN RETURN BLACKLISTED
   IF HASH(password) != user.hashedPassword THEN RETURN INVALID_CREDENTIALS
   sessionToken = GENERATE_SECURE_TOKEN()
   STORE sessionToken WITH userId, expiration
   RETURN sessionToken
END


METHOD processCustomerRegistration(managerId, userId, decision, reason)
   // Input: managerId, userId, decision(APPROVE/REJECT), reason
   // Output: boolean
   VERIFY managerId.role = "Manager"
   user = FIND User BY userId
   IF user IS NULL OR user.status != "PendingApproval" THEN RETURN false
   IF decision = APPROVE THEN
      user.status = "Active"
   ELSE
      user.status = "Rejected"
      user.rejectionReason = reason
   ENDIF
```

```
    SAVE user
    LOG decision
    RETURN true
END


METHOD updateVIPStatus(customerId)
    // Input: customerId
    // Output: boolean (isNowVIP)
    customer = FIND Customer BY customerId
    IF customer IS NULL THEN RETURN false
    IF customer.warningCount > 0 THEN RETURN false
    IF customer.totalSpent > 100 OR customer.orderCount >= 3 THEN
        customer.isVIP = true
        SAVE customer
        RETURN true
    ENDIF
    RETURN false
END


METHOD blacklistUser(managerId, userId)
    // Input: managerId, userId
    // Output: boolean
    VERIFY managerId.role = "Manager"
    user = FIND User BY userId
    IF user IS NULL THEN RETURN false
    user.isBlacklisted = true
    user.status = "Blacklisted"
    SAVE user
    ADD user.email TO BlacklistTable
    RETURN true
END


METHOD getMenuForVisitor()
    // Input: none
    // Output: listOfDishes
    popular = QUERY dishes ORDER BY orderCount DESC LIMIT N
    topRated = QUERY dishes ORDER BY averageRating DESC LIMIT M
    combined = MERGE_AND_DEDUP(popular, topRated)
    RETURN combined
END


METHOD getMenuForCustomer(customerId)
    // Input: customerId
    // Output: {mostOrdered, highestRated, popular}
    orders = QUERY past orders BY customerId
    mostOrdered = TOP_DISHES_FROM(orders)
    highestRated = TOP_RATED_BY_CUSTOMER(customerId)
    popular = QUERY dishes ORDER BY globalPopularity DESC LIMIT K
    RETURN {mostOrdered, highestRated, popular}
END


METHOD createOrUpdateDish(chefId, dishData)
    // Input: chefId, dishData
    // Output: dishId
    VERIFY chefId.role = "Chef"
    IF dishData.id EXISTS THEN
        dish = FIND Dish BY dishData.id
        UPDATE dish FROM dishData
    ELSE
        dish = NEW Dish(dishData)
        dish.orderCount = 0
        dish.averageRating = 0
    ENDIF
    SAVE dish
    RETURN dish.id
END


METHOD depositMoney(customerId, amount)
    // Input: customerId, amount
    // Output: newBalance OR errorCode
    IF amount <= 0 THEN RETURN INVALID_AMOUNT
    customer = FIND Customer BY customerId
    IF customer IS NULL OR customer.status != "Active" THEN RETURN
INVALID_CUSTOMER
```

```
    customer.balance += amount
    SAVE customer
    LOG transaction("DEPOSIT", amount)
    RETURN customer.balance
END


METHOD createOrder(customerId, cartItems)
    // Input: customerId, cartItems[]
    // Output: orderId OR errorCode
    IF cartItems IS EMPTY THEN RETURN EMPTY_CART
    customer = FIND Customer BY customerId
    IF customer IS NULL OR customer.status != "Active" THEN RETURN
INVALID_CUSTOMER
    order = NEW Order(customerId)
    totalPrice = 0
    FOR EACH item IN cartItems DO
        dish = FIND Dish BY item.dishId
        IF dish IS NULL OR NOT dish.isAvailable THEN RETURN DISH_NOT_AVAILABLE
        linePrice = dish.price * item.quantity
        ADD LINE(order, dish.id, item.quantity, dish.price)
        totalPrice += linePrice
    END FOR
    discount = CALCULATE_VIP_DISCOUNT(customer, totalPrice)
    order.originalPrice = totalPrice
    order.discountApplied = discount
    order.totalPrice = totalPrice - discount
    order.status = "PendingPayment"
    SAVE order
    RETURN order.id
END


METHOD CALCULATE_VIP_DISCOUNT(customer, totalPrice)
    // Input: customer, totalPrice
    // Output: discountAmount
    IF customer.isVIP THEN RETURN totalPrice * 0.05
    RETURN 0
END
```

```
METHOD confirmOrder(orderId)
   // Input: orderId
   // Output: true OR errorCode
   order = FIND Order BY orderId
   IF order IS NULL OR order.status != "PendingPayment" THEN RETURN
INVALID_ORDER
   customer = FIND Customer BY order.customerId
   IF customer.balance < order.totalPrice THEN
      APPLY_WARNING(customer.id, "Insufficient funds")
      order.status = "Rejected_Insufficient_Funds"
      SAVE order
      RETURN INSUFFICIENT_FUNDS
   ENDIF
   customer.balance -= order.totalPrice
   customer.totalSpent += order.totalPrice
   customer.orderCount += 1
   SAVE customer
   updateVIPStatus(customer.id)
   order.status = "Paid"
   SAVE order
   enqueueOrderForKitchen(order.id)
   startDeliveryBidding(order.id)
   RETURN true
END


METHOD enqueueOrderForKitchen(orderId)
   // Input: orderId
   // Output: boolean
   order = FIND Order BY orderId
   IF order IS NULL THEN RETURN false
   order.status = "Queued_For_Preparation"
   SAVE order
   ADD orderId TO KitchenQueue
   NOTIFY Chefs
   RETURN true
END
```

```
METHOD getPendingOrdersForChef(chefId)
   // Input: chefId
   // Output: listOfOrders
   VERIFY chefId.role = "Chef"
   pending = QUERY Orders WHERE status IN ("Queued_For_Preparation", "In_Preparation")
   RETURN pending
END


METHOD updateOrderPreparationStatus(chefId, orderId, newStatus)
   // Input: chefId, orderId, newStatus
   // Output: boolean
   VERIFY chefId.role = "Chef"
   order = FIND Order BY orderId
   IF order IS NULL THEN RETURN false
   IF newStatus NOT IN ("In_Preparation","Ready_For_Delivery","On_Hold") THEN RETURN
false
   order.status = newStatus
   SAVE order
   IF newStatus = "Ready_For_Delivery" THEN
      NOTIFY Manager AND DeliveryPeople
   ENDIF
   RETURN true
END


METHOD reportIngredientShortage(chefId, orderId, note)
   // Input: chefId, orderId, note
   // Output: boolean
   VERIFY chefId.role = "Chef"
   order = FIND Order BY orderId
   IF order IS NULL THEN RETURN false
   order.status = "On_Hold"
   order.shortageNote = note
   SAVE order
   NOTIFY Manager
   RETURN true
END
```

```
METHOD createDeliveryBid(deliveryPersonId, orderId, bidAmount)
  // Input: deliveryPersonId, orderId, bidAmount
  // Output: bidId OR errorCode
  VERIFY deliveryPersonId.role = "DeliveryPerson"
  order = FIND Order BY orderId
  IF order IS NULL OR order.status != "Ready_For_Delivery" THEN RETURN
INVALID_ORDER
  IF bidAmount <= 0 THEN RETURN INVALID_BID
  bid = NEW DeliveryBid(orderId, deliveryPersonId, bidAmount, "Pending")
  SAVE bid
  RETURN bid.id
END


METHOD assignDelivery(managerId, orderId, chosenBidId, justificationText)
  // Input: managerId, orderId, chosenBidId, justificationText
  // Output: deliveryId OR errorCode
  VERIFY managerId.role = "Manager"
  order = FIND Order BY orderId
  IF order IS NULL OR order.status != "Ready_For_Delivery" THEN RETURN
INVALID_ORDER
  bids = QUERY DeliveryBids WHERE orderId = orderId AND status = "Pending"
  chosen = FIND IN bids WHERE id = chosenBidId
  IF chosen IS NULL THEN RETURN INVALID_BID
  lowest = MIN_BY_AMOUNT(bids)
  IF chosen.id != lowest.id THEN
    IF justificationText IS EMPTY THEN RETURN JUSTIFICATION_REQUIRED
    LOG justification(managerId, orderId, chosenBidId, justificationText)
  ENDIF
  delivery = NEW Delivery(orderId, chosen.deliveryPersonId, chosen.amount, "Assigned")
  SAVE delivery
  order.status = "Awaiting_Pickup"
  SAVE order
  FOR EACH bid IN bids DO
    bid.status = (bid.id = chosenBidId ? "Accepted" : "Rejected")
    SAVE bid
  END FOR
  NOTIFY chosen.deliveryPersonId
  RETURN delivery.id
END
```

```
METHOD updateDeliveryStatus(deliveryPersonId, deliveryId, newStatus, note)
   // Input: deliveryPersonId, deliveryId, newStatus, note
   // Output: boolean
   VERIFY deliveryPersonId.role = "DeliveryPerson"
   delivery = FIND Delivery BY deliveryId
   IF delivery IS NULL OR delivery.deliveryPersonId != deliveryPersonId THEN RETURN
false
   delivery.status = newStatus
   delivery.note = note
   SAVE delivery
   order = FIND Order BY delivery.orderId
   IF newStatus = "Out_For_Delivery" THEN
      order.status = "Out_For_Delivery"
   ELSE IF newStatus = "Delivered" THEN
      order.status = "Completed"
      RELEASE_PAYMENT_TO_DELIVERY_PERSON(deliveryPersonId,
delivery.bidAmount)
      PROMPT_FEEDBACK(order.customerId, deliveryPersonId, order.id)
   ELSE IF newStatus = "Delivery_Failed" THEN
      order.status = "Delivery_Failed"
      NOTIFY Manager
   ENDIF
   SAVE order
   RETURN true
END
```

```
METHOD fileComplaintOrCompliment(fromUserId, toEntityId, entityType, isComplaint,
message)
   // Input: fromUserId, toEntityId, entityType, isComplaint, message
   // Output: recordId
   fromUser = FIND User BY fromUserId
   weight = 1
   IF fromUser.role = "Customer" AND fromUser.isVIP THEN weight = 2
   record = NEW ReputationRecord(fromUserId, toEntityId, entityType,
                          isComplaint, message, weight,
                          status="PendingReview")
   SAVE record
```

```
    NOTIFY Manager
    RETURN record.id
END


METHOD resolveComplaint(managerId, complaintId, outcome)
    // Input: managerId, complaintId, outcome
    // Output: boolean
    VERIFY managerId.role = "Manager"
    record = FIND ReputationRecord BY complaintId
    IF record IS NULL OR record.isComplaint = false THEN RETURN false
    record.status = outcome
    SAVE record
    IF outcome = "VALID" THEN
        APPLY_COMPLAINT_EFFECT(record.toEntityId, record.entityType, record.weight)
    ELSE IF outcome = "INVALID" THEN
        APPLY_WARNING(record.fromUserId, "Complaint without merit")
    ENDIF
    RETURN true
END


METHOD APPLY_WARNING(userId, reason)
    // Input: userId, reason
    // Output: none
    user = FIND User BY userId
    IF user IS NULL THEN RETURN
    user.warningCount += 1
    LOG warning(reason)
    IF user.role = "Customer" THEN
        IF user.isVIP THEN
            IF user.warningCount >= 2 THEN
                user.isVIP = false
                user.warningCount = 0
            ENDIF
        ELSE
            IF user.warningCount >= 3 THEN
                user.status = "Deregistered"
                blacklistUserForWarnings(user.id)
            ENDIF
        ENDIF
    ENDIF
    SAVE user
END


METHOD APPLY_COMPLAINT_EFFECT(employeeId, entityType, weight)
    // Input: employeeId, entityType, weight
    // Output: none
    employee = FIND User BY employeeId
    IF employee IS NULL THEN RETURN
    employee.netComplaints += weight
    SAVE employee
    IF entityType IN ("Chef","DeliveryPerson") AND employee.netComplaints >= 3 THEN
        IF employee.demotionsCount = 0 THEN
            employee.role = "Demoted_" + entityType
            employee.demotionsCount += 1
        ELSE IF employee.demotionsCount = 1 THEN
            employee.status = "Terminated"
        ENDIF
        SAVE employee
    ENDIF
END


METHOD handleChatQuery(userId, queryText)
    // Input: userId (nullable), queryText
    // Output: (answerText, source, answerId)
    kbAnswer = queryKnowledgeBase(queryText)
    IF kbAnswer != NULL THEN
        answerText = kbAnswer.text
        source = "KB"
    ELSE
        llm = queryLLM(queryText)
        answerText = llm.text
        source = "LLM"
    ENDIF
    record = NEW ChatAnswer(userId, queryText, answerText, source,
                            rating=null, flagged=false)
    SAVE record
    RETURN (answerText, source, record.id)
END


METHOD queryKnowledgeBase(queryText)
    // Input: queryText
    // Output: kbAnswerOrNull
    matches = SEARCH KBArticles BY queryText
    IF matches EMPTY THEN RETURN NULL
    best = HIGHEST_SIMILARITY(matches)
    RETURN {text: best.answerText, articleId: best.id}
END


METHOD queryLLM(queryText)
    // Input: queryText
    // Output: llmAnswer
    resp = CALL_EXTERNAL_LLM_API({prompt: queryText})
    RETURN {text: resp.text}
END


METHOD rateAnswer(userId, answerId, rating)
    // Input: userId, answerId, rating(0–5)
    // Output: boolean
    answer = FIND ChatAnswer BY answerId
    IF answer IS NULL THEN RETURN false
    answer.rating = rating
    SAVE answer
    IF rating = 0 THEN
        answer.flagged = true
        SAVE answer
        NOTIFY Manager
    ENDIF
    RETURN true
END


METHOD startVoiceRecording(clientState)
    // Input: clientState
    // Output: updatedClientState
    clientState.mode = "Listening"
    INITIATE_AUDIO_CAPTURE()
    RETURN clientState
END


METHOD stopAndSendRecording(audioFile, userId)
    // Input: audioFile, userId
    // Output: responseAudioFile OR errorCode
    FINALIZE_AUDIO_CAPTURE(audioFile)
    resp = HTTP_POST("/api/voice-query", {audioFile, userId})
    IF resp.status != 200 THEN RETURN VOICE_PROCESSING_ERROR
    RETURN resp.audioFile
END


METHOD processVoiceRequest(audioFile, userId)
    // Input: audioFile, userId
    // Output: responseAudioFile
    transcript = callElevenLabsSTT(audioFile)
    (answerText, source, answerId) = handleChatQuery(userId, transcript)
    responseAudio = callElevenLabsTTS(answerText)
    RETURN responseAudio
END


METHOD callElevenLabsSTT(audioFile)
    // Input: audioFile
    // Output: transcriptText
    resp = CALL_ELEVENLABS_STT_API({audio: audioFile})
    RETURN resp.transcript
END


METHOD callElevenLabsTTS(text)
    // Input: text
    // Output: audioFile
    resp = CALL_ELEVENLABS_TTS_API({text, voiceId: DEFAULT_VOICE})
    RETURN resp.audioFile
```

```
END


METHOD getAdminDashboardData(managerId)
   // Input: managerId
   // Output: dashboardData
   VERIFY managerId.role = "Manager"
   pending = QUERY Users WHERE status = "PendingApproval"
   openComplaints = QUERY ReputationRecords WHERE status="PendingReview" AND
isComplaint=true
   flagged = QUERY ChatAnswers WHERE flagged=true
   unresolvedCount = COUNT openComplaints
   RETURN {
      pendingRegistrations: pending,
      openComplaints: openComplaints,
      flaggedAIAnswers: flagged,
      unresolvedComplaintsCount: unresolvedCount
   }
END


METHOD handleSystemAlertForComplaints(managerId)
   // Input: managerId
   // Output: none
   VERIFY managerId.role = "Manager"
   unresolved = QUERY ReputationRecords WHERE status="PendingReview" AND
isComplaint=true
   IF COUNT(unresolved) > THRESHOLD THEN
      SEND_PRIORITY_ALERT(managerId, "Multiple unresolved complaints")
   ENDIF
END
```