WORKSPIRE

Software Design Specification

02.05.2025

Zeynep KURT 150121042

Damla KUNDAK 150121001

Ayşe Nisa ŞEN 150121040

Prepared for CSE3044 Software Engineering Term Project

Table of Contents

1. Introduction

1.1. Purpose

The purpose of this Software Requirements Specification (SRS) is to define the functional and non-functional requirements for the Team Collaboration and Productivity Web Application. The intended audience includes software engineers, developers, testers, project managers, and stakeholders involved in the project.

1.2. Statement of scope

This software aims to enhance team collaboration, task management, and productivity within companies. Key features include:

- Feature assignment and tracking

- Individual and team dashboards

- Performance evaluation with scoring and ranking systems

- Built-in communication for messaging

- Pomodoro timer integration for time management

- Mini-game for engagement and motivation

This system will be a web-based application accessible via browsers, ensuring openness, responsibility, and efficient work processes within teams.

1.3. Software context

The software is placed in a business or product line context. Strategic issues relevant to context are discussed. The intent is for the reader to understand the 'big picture'.

1.4. Major constraints

Any business or product line constraints that will impact the manner in which the software is to be specified, designed, implemented or tested are noted here.

1.5 Definitions, Acronyms, and Abbreviations

●**DSD**: Design Specification Document

● **UI**: User Interface

● **DBMS:** Database Management System

● **API:** Application Programming Interface

● **Pomodoro Timer**: A time management method based on 25-minute work intervals

● **Dashboard** : The main UI screen presented after login, containing key features and shortcuts

● **AI Task Prioritizer:** System feature that uses AI to suggest or sort tasks based on urgency or user behavior Software Requirements Specification Page 3

● **Ranking / Scoreboard**: A system that tracks users' performance based on completed assigned tasks

● Chat: A communication feature that allows team members to message each other in real-time.

● **Smart Meeting Scheduler**: AI-powered feature that finds optimal time slots for meetings based on calendar availability.

● **Authentication**: The process of verifying a user's identity before granting system access.

● **Employee**: A regular system user who can create personal tasks, communicate, and track progress.

● **Manager**: A user role with permissions to assign tasks to employees and view team activity.

● **System**: The Workspire platform, including backend services and AI engines.

1.6. References

- Lecture Slides – CSE3044 Software Engineering
- Agile Software Development Principles
- PostgreSQL, Node.js, and React.js Documentation.
- React & Node.js Official Docs

2. Design Consideration

This section outlines the foundational design principles, environment, assumptions, and constraints considered throughout the development of the WorkSpire platform.

2.1. Design Assumptions and Dependencies

- The system is designed as a browser-based web application accessible from modern devices.
- The primary users are employees and managers within an organization, with different permissions and access rights.
- The software relies on a PostgreSQL database, storing structured data for users, tasks, communications, and sessions.
- The application is developed using a full-stack JavaScript environment for core functionalities, supported by Python-based services where needed.
- Machine learning capabilities are used to enhance user productivity and streamline collaborative work routines.
- Real-time messaging is achieved using WebSockets, and state management is handled via component-based design on the frontend.
- End-users are expected to have consistent internet access to ensure real-time updates and API interactions.

2.2. General Constraints

- The system follows a modular architecture with strict separation between UI, backend services, and AI components.
- Communication between backend and auxiliary services is implemented through RESTful APIs.
- All services must comply with authentication protocols using JWT, and secure communication must be maintained via HTTPS.
- The platform is optimized for standard working hours and assumes user activity during weekdays.

- The backend is responsible for enforcing data validation, access restrictions, and business rules across all modules.
- Time-sensitive operations such as task updates, chat messages, and scheduling suggestions require low-latency processing.
- Client-side components should remain responsive under high task or message load, using caching or pagination strategies where necessary.

2.3. System Environment

- **Frontend**: React.js, using functional components and responsive layouts.
- **Backend**: Node.js with Express, exposing RESTful APIs.
- **Database**: PostgreSQL, managing structured data including tasks, messages, users, sessions, and schedules.
- **External Services**: Python services integrated through HTTP-based communication for processing complex logic.
- **Communication**: Socket.io for real-time messaging; WebSockets configured for persistent connections.
- **Development Stack**: Visual Studio Code, GitHub (for version control), Docker (containerization), pgAdmin (database inspection).

2.4. Development Methods

- The team adopts Agile development methodologies, working in iterations and conducting continuous integration and testing.
- Feature development follows a component-based approach, allowing reuse and easier maintainability.
- RESTful API design ensures scalability and interoperability between internal modules and external services.
- Testing is conducted at multiple levels, including unit, integration, and end-to-end, covering both frontend interactions and backend logic.
- All components are documented and reviewed regularly to ensure alignment with software requirements and stakeholder expectations.

3. Architectural and component-level design

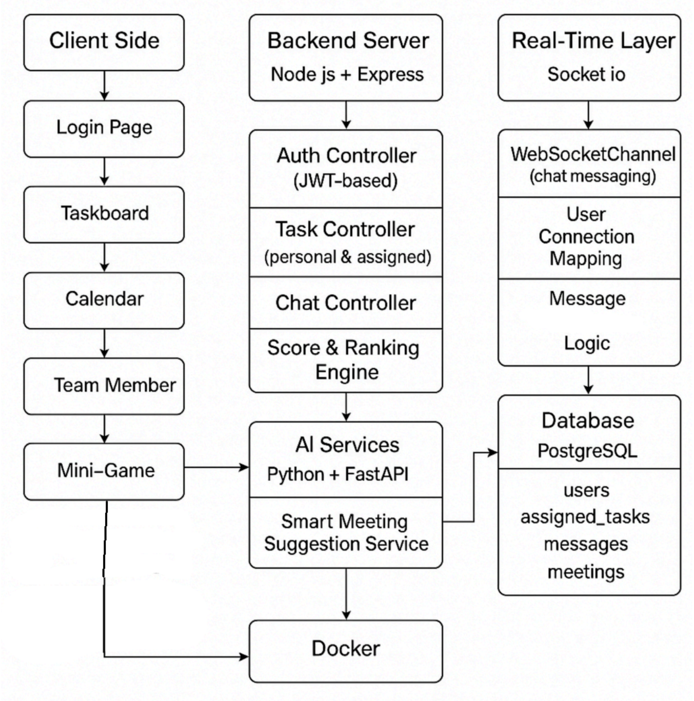
This section provides a comprehensive view of the system architecture and describes the components that make up the WorkSpire platform.

3.1. System Structure

WorkSpire is developed using a **modular client-server architecture** with clear separation of concerns between frontend, backend, real-time messaging, and external services. The system is divided into several core modules, each responsible for distinct functionality:

- **Frontend (React)**: Presents the UI components for login, dashboard, chat, task management, and productivity tools.
- **Backend (Node.js + Express)**: Handles authentication, task logic, user management, data access, and business rules.
- **Database (PostgreSQL)**: Stores persistent information such as users, tasks, messages, scores, and meeting schedules.
- **Real-Time Communication (Socket.io)**: Enables chat features and live updates across clients.
- **External Services (Python APIs)**: Delivers intelligent suggestions for meeting scheduling and task prioritization.

3.1.1. Architecture diagram



3.2. Description for Components

3.2.1 Description for Component User Login

3.2.1.1 Processing narrative (PSPEC) for component User Login

- The user navigates to the login page.
- The user enters an email and password.
- The form is validated (required fields check, email format check).
- A POST request is sent to the login API.
- The backend verifies the credentials against the database.
- If successful, a JWT token is generated and returned.
- The user is redirected to the Dashboard page.
- If unsuccessful, an error message is displayed.

3.2.1.2 Component User Login interface description

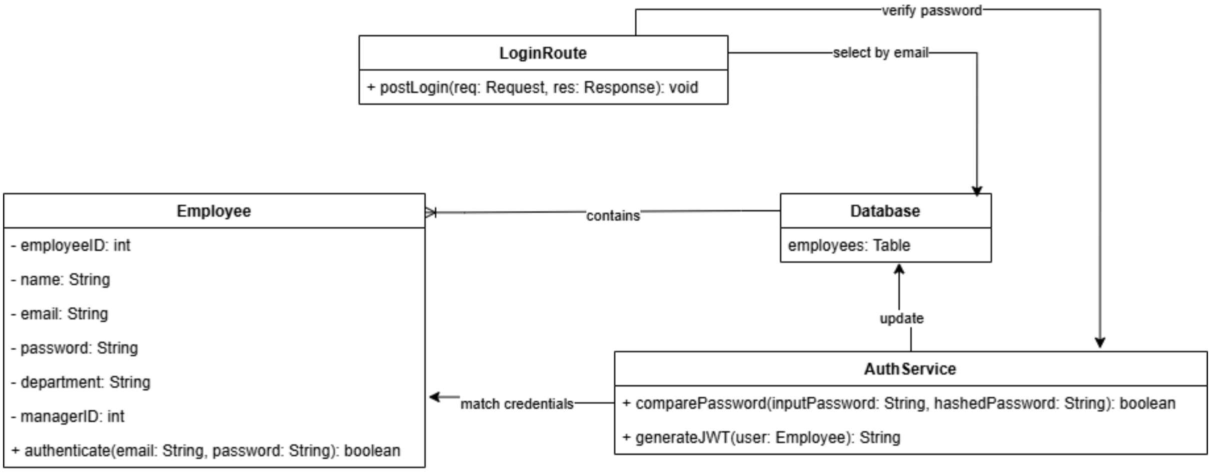
Input:

- email: String
- password: String

Output:

- Status: 200 OK with JWT token
- or Status: 401 Unauthorized with error message

3.2.1.3 Design Class hierarchy for component User Login



3.2.1.4 Restrictions/limitations for component User Login

- Passwords must be at least 8 characters.
- Email must be in a valid email format.
- No detailed error messages for incorrect login (security best practice).

3.2.1.5 Performance issues for component User Login

- High login attempts can cause server load; rate-limiting is recommended.
- JWT token generation must be efficient to avoid latency.

3.2.1.6 Design constraints for component User Login

- Authentication must use HTTPS.
- Passwords must be securely hashed (e.g., bcrypt).
- JWT tokens must have expiration times.

3.2.1.7 Processing Detail for Component User Login

1. User submits credentials via POST /login.
2. Backend queries employees table by email.

3. Password is validated (bcrypt).
4. On success, a JWT token is generated.
5. Token and user info returned with 200 OK.
6. On failure, returns 401 Unauthorized.

3.2.2 Description for Component Add Personal Task

3.2.2.1 Processing narrative (PSPEC) for component Add Personal Task

- The user logs in and navigates to the Dashboard.
- The user clicks the "Add Personal Task" button.
- A form appears where the user enters task title and optional description.
- After submission, the frontend validates required fields.
- A POST request is sent to the /todos endpoint.
- The backend saves the task into the database.
- A success notification is shown to the user.

3.2.2.2 Component Add Personal Task interface description

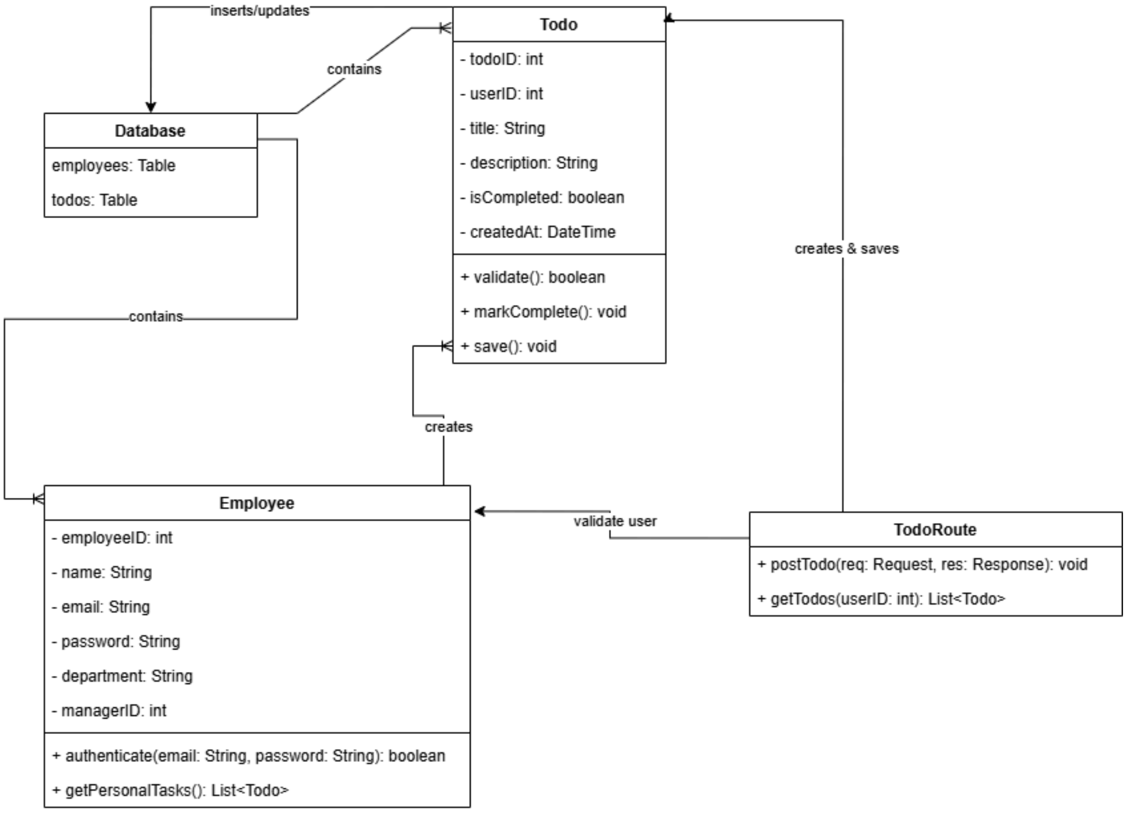
Input:

- user_id: int
- title: String
- description: String (optional)

Output:

- Status: 201 Created
- or Status: 400 Bad Request (validation failure)

3.2.2.3 Design Class hierarchy for component Add Personal Task



3.2.2.4 Restrictions/limitations for component Add Personal Task

- Task title cannot be empty.
- A user cannot have more than 100 active personal tasks.
- Description is optional.

3.2.2.5 Performance issues for component Add Personal Task

- Each task insertion involves a database write operation.
- Large numbers of tasks can slow down Dashboard loading; pagination is advised.

3.2.2.6 Design constraints for component Add Personal Task

- RESTful API with JSON payloads.
- JWT authentication is required.
- Task data must match the database schema (todos table).

3.2.2.7 Processing Detail for Component Add Personal Task

1. User opens the dashboard and clicks "Add Task".
2. Frontend shows modal and submits via POST /todos.
3. Backend validates input (title, user ID).
4. Task inserted into todos table.
5. Returns 201 Created; frontend refreshes task list.

3.2.3.Description for Component Assign Task

3.2.3.1. Processing narrative (PSPEC) for component Assign Task

Login & Navigation

- Manager authenticates and opens the "Assign Task" section.

Select Team Member

- Component fetches and displays the manager's direct reports.
- Manager picks one employee from the list.

Enter Task Details

- Manager fills in Task Name, Task Description (optional), Deadline and Score (optional).

Validation & Submission

- Component checks required fields and business rules.
- On success, it submits a request to create the task.

Persistence & Confirmation

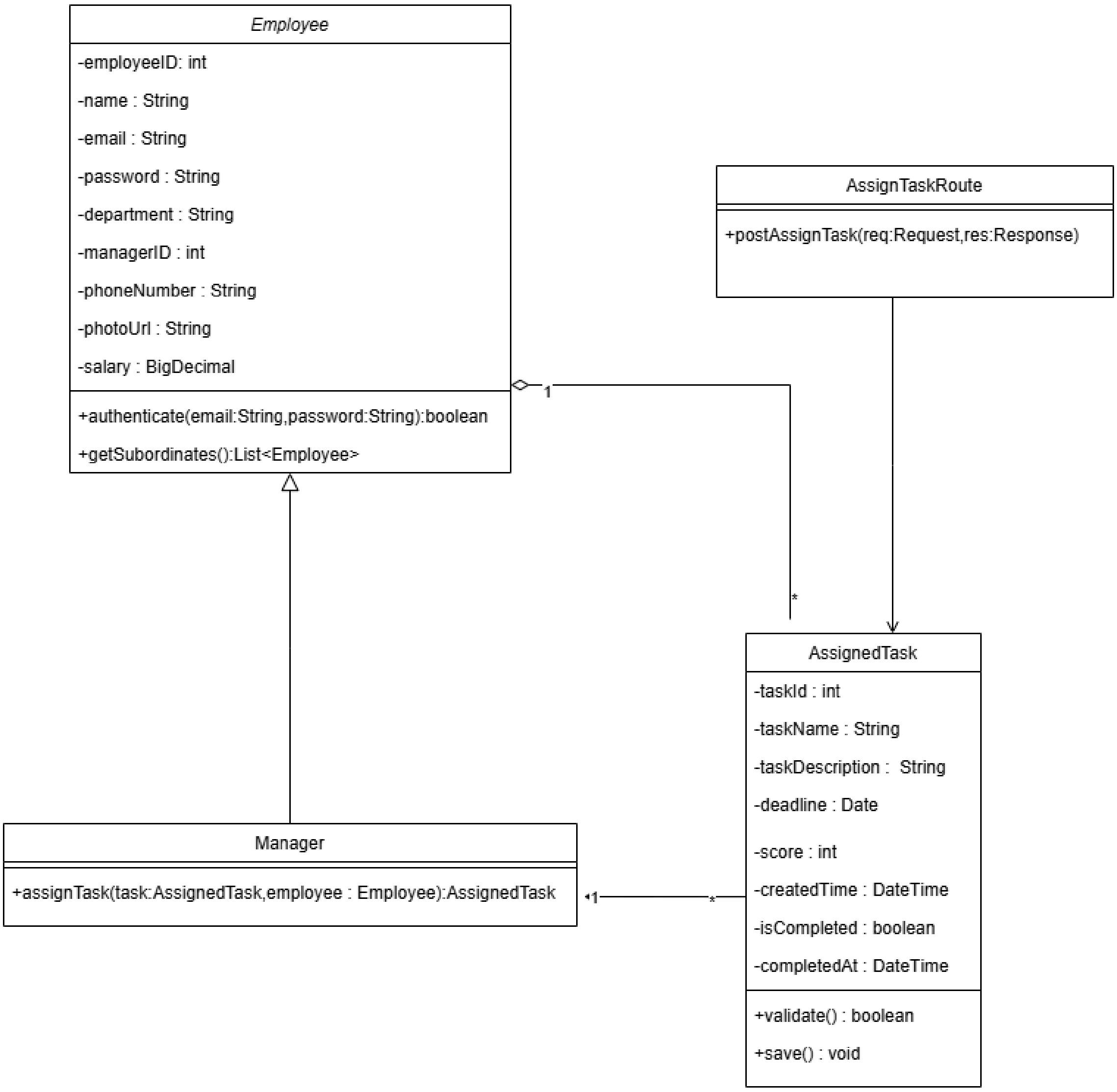
- Component saves the record, then displays a success message or error.

3.2.3.2. Component Assign Task interface description.

Input: employee_id, manager_id, task_name, task_description, deadline, score

Output: **Status**: 201 Created

3.2.3.3 Design Class hierarchy for component Assign Task



3.2.3.4 Restrictions/limitations for component Assign Task

- Role-based access: only managers can call this component.
- Required fields: taskName, deadline. Missing either → error.

3.2.3.5 Performance issues for component Assign Task

- No Batch Inserts: Single INSERT per task slows under load.
- Missing Indexes: Key columns (employee_id, manager_id, deadline) unindexed → slow queries.
- Connection Pool Limits: Default pool size can be exhausted by concurrent requests.
- Network Overhead: One HTTP round-trip per assignment adds latency.
- Full UI Refresh: Re-rendering the entire task list after each assign degrades responsiveness.

3.2.3.6 Design constraints for component Assign Task

Tech stack: Node.js/Express, PostgreSQL, React.

API style: RESTful, JSON payloads.

Security: HTTPS, JWT or session authentication.

Schema alignment: must match existing assigned_tasks table columns and types.

3.2.3.7 Processing detail for each operation of component Assign Task

- Manager logs in and opens the "Assign Task" page.
- System loads employees reporting to the manager.
- Manager selects an employee and fills in task details.
- System validates and saves the task to the assigned_tasks table.
- A confirmation is shown and the task appears in the employee's dashboard.

3.2.4. Description for Component Mark Task as Completed

3.2.4.1. Processing narrative (PSPEC) for component Mark Task as Completed

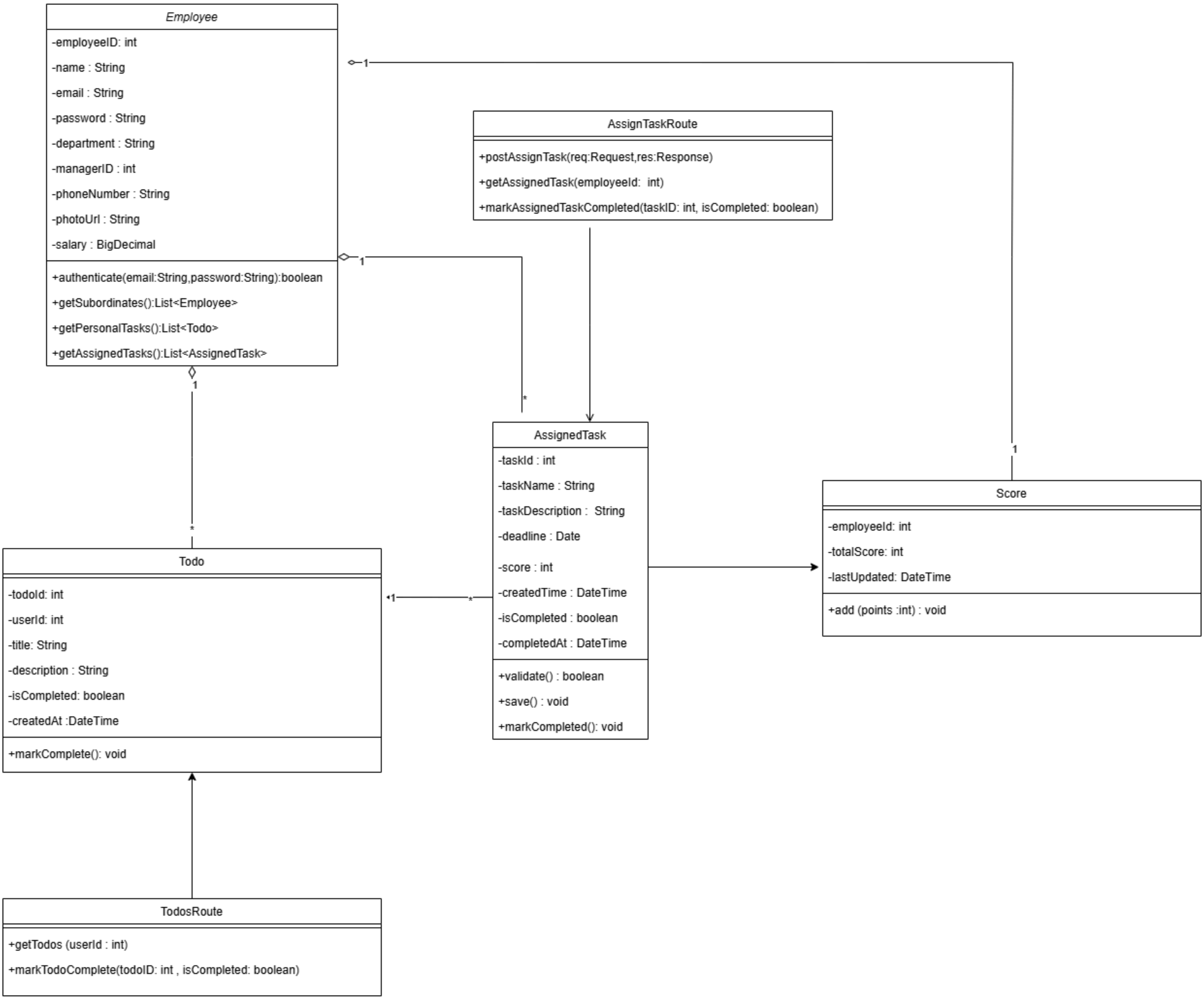
- Employee logs in and sees Personal and Assigned Tasks.
- Employee checks the "complete" box on a task.
- Frontend calls the appropriate API to update is_completed.
- Backend updates the task row (and, for assigned tasks, fires a trigger to adjust the score).
- UI shows the task as struck through and updates the scoreboard.

3.2.4.2. Component Mark Task as Completed interface description.

Input is_completed: true

Output: **Status:** 200 OK

3.2.4.3 Design Class hierarchy for component Mark Task as Completed



3.2.4.4 Restrictions/limitations for component Mark Task as Completed

- Only the task owner (or the manager, for assigned tasks) may mark completion.
- Once marked completed, is_completed cannot be unset via API.

3.2.4.5 Performance issues for component Mark Task as Completed

- Each completion fires one UPDATE (and a trigger for assigned tasks).
- Triggered score calculation adds minimal overhead per request.

3.2.4.6 Design constraints for component Mark Task as Completed

- Uses existing Express routes and PostgreSQL trigger update_score.
- Must conform to current table schemas (todos, assigned_tasks, scores).

3.2.4.7 Processing detail for each operation of component Mark Task as Completed

- Employee marks the assigned task as completed.
- System updates assigned_tasks.
- Trigger updates score in the scores table.
- UI updates and score is reflected.

3.2.5 Description for Component Pomodoro Timer

3.2.5.1 Processing narrative (PSPEC) for component Pomodoro Timer

- The user navigates to the Pomodoro Timer page.
- The user clicks "Start" to begin a 25-minute session.
- A countdown timer is initiated and displayed.
- After 25 minutes, a break suggestion alert is shown.
- The user can start a break or restart a Pomodoro session.

3.2.5.2 Component Pomodoro Timer interface description

Input:

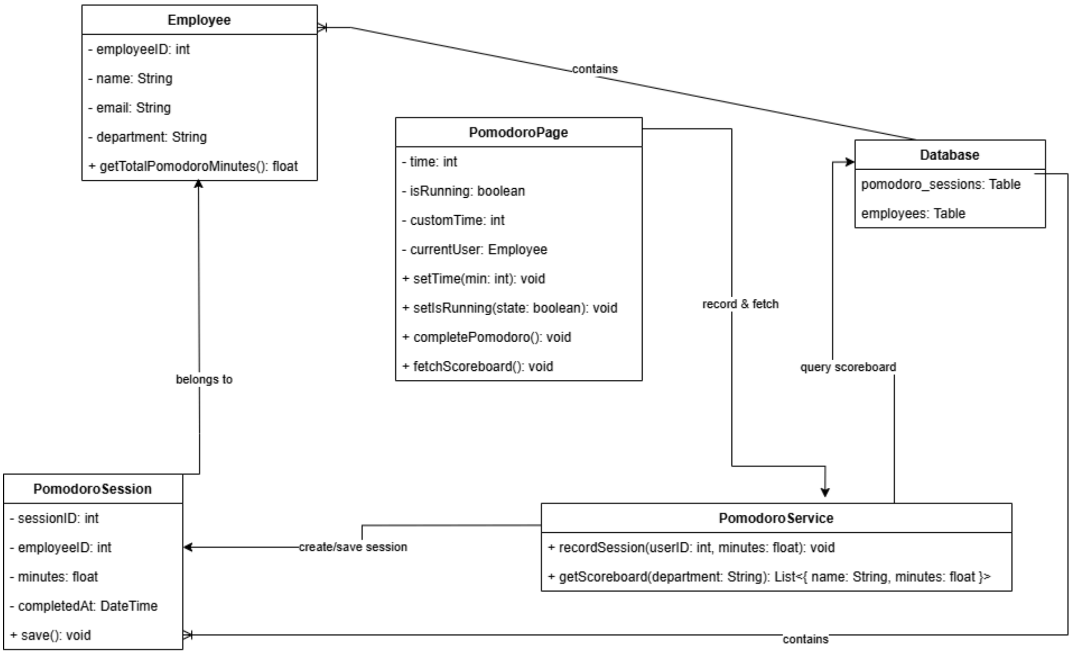
- Start Timer
- Break Start

Output:

- Session completion alert
- Break completion alert

(Pomodoro Timer operates mostly on the frontend without a backend call.)

3.2.5.3 Design Class hierarchy for component Pomodoro Timer



3.2.5.4 Restrictions/limitations for component Pomodoro Timer

- If the browser tab is closed, the timer resets unless localStorage persistence is used.
- Break duration is fixed (5 minutes).

3.2.5.5 Performance issues for component Pomodoro Timer

- setInterval function must be optimized to avoid performance bottlenecks.
- Unnecessary re-renders should be minimized.

3.2.5.6 Design constraints for component Pomodoro Timer

- Implemented using React components.
- Timer logic handled on the client side.
- Responsive design for different screen sizes.

3.2.5.7 Processing Detail for Component Pomodoro Timer

1. User navigates to the Pomodoro page and clicks "Start".
2. Timer starts on frontend using setInterval().
3. After 25 mins, POST /pomodoro/complete is sent.

4. Server inserts session into pomodoro_sessions table.
5. Leaderboard is refreshed using GET /pomodoro/scoreboard.
6. Frontend updates scoreboard view.

3.2.6. Description for Compnenent View Team and Company-wide Scoreboard

3.2.6.1 Processing narrative (PSPEC) for component View Team and Company-wide Scoreboard

- After logging in, the user navigates to the Scoreboard page from the dashboard.
- By default, the scoreboard displays company-wide scores.
- The user can toggle between "Team" and "Company-wide" views using the respective buttons.
- The scoreboard updates automatically whenever the user changes the view (team/company).

3.2.6.2 Component View Team and Company-wide Scoreboard interface description

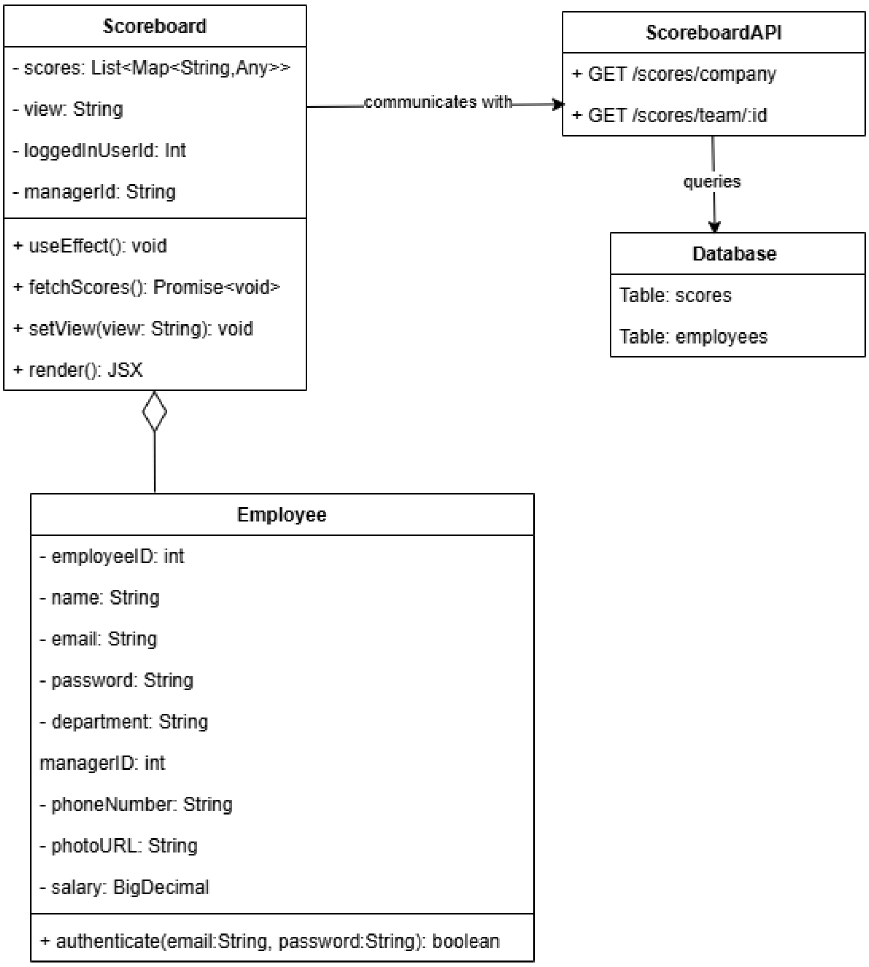
Input:

- managerId
- view (state)
- employeeId

Output:

- HTTP GET /scores/company
- HTTP GET /scores/team/{managerId}
- A visual scoreboard consisting of: Top 3 employees displayed with larger profile images and ranking.
- A scrollable table showing full rankings with: Rank, profile photo, employee name, total score, highlighted row for the currently logged-in user.

3.2.6.3 Design Class hierarchy for component View Team and Company-wide Scoreboard



3.2.6.4 Restrictions/limitations for component View Team and Company-wide Scoreboard

- **Manager-only filtering for team view**: The "Team" view relies on the presence of a valid manager_id. If a user has no manager (e.g., an admin or CEO), the team view may return empty or fail.
- **No pagination or infinite scroll**: The scoreboard retrieves and renders the full list of users. With a large number of employees, this could lead to UI lag or long initial load times.
- **Static top-3 logic**: The component assumes there will always be at least three users in the result. If there are fewer than three, the podium display may not render correctly.

3.2.6.5 Performance issues for component View Team and Company-wide Scoreboard

- **Full Data Fetch on Each View Switch:** Switching between "Company" and "Team" views triggers a complete re-fetch of score data from the backend, even if the data hasn’t changed. This results in redundant network calls and slower response times.
- **No Caching Mechanism:** There is no client-side or server-side caching implemented for scoreboard data. Frequent accesses can increase server load unnecessarily, especially during peak usage.
- **Inefficient Rendering for Large User Sets:** The entire scoreboard, including profile images and detailed rows, is rendered client-side without pagination or virtualization. This may lead to lag or UI unresponsiveness when dealing with large datasets.
- **Image Loading Overhead:** Each user's profile picture is fetched from a URL and rendered as a circular image. When the list is long, concurrent image loads may degrade performance and increase page load time.

3.2.6.6 Design constraints for component View Team and Company-wide Scoreboard

- **Frontend**: React.js (functional components, context API for dark mode and language).
- **Backend**: Node.js + Express.
- **API Communication**: RESTful GET endpoints (*/scores/company, /scores/team/:manager_id*).
- **Database**: PostgreSQL using *employees* and *scores* tables (joined by employee_id).
- **Authentication**: Relies on JWT-based login; user metadata (e.g., managerId, employeeId) accessed from *localStorage*.
- **Data Format**: JSON response with employee_id, name, photo_url, total_score.
- **Theming**: Supports dark/light mode via contextual state (*darkMode*).

3.2.6.7 Processing Detail for Component View Team and Company-wide Scoreboard

- The user logs in and navigates to the **Scoreboard** page from the dashboard.
- The frontend component reads user-specific data (employeeId, managerId, language, darkMode) from *localStorage*.
- By default, the view is set to **"Company-wide"**:
  - A GET request is sent to the backend endpoint: GET /scores/company
- The backend executes a SQL query joining scores and employees tables to retrieve:
  - employee_id, name, photo_url, total_score
  - Sorted in descending order by total_score
- The response is returned as a JSON array and stored in the frontend state (*scores*).
- The top 3 employees are displayed visually with profile pictures and ranking.
- The remaining scores are rendered in a table, with the current user's row highlighted.
- If the user switches to **"Team"** view:
  - A new GET request is sent to: GET /scores/team/:manager_id
- The backend filters the scores by employees with the specified *manager_id* and returns the sorted result.
- The UI re-renders the updated scoreboard based on the selected view.

3.2.7. Description for Component Real-Time Chat

3.2.7.1 Processing narrative (PSPEC) for component Real-Time Chat

- After successful login, the user navigates to the Chat section via the sidebar or interacts with the floating *ChatWidget*.
- The frontend initializes a WebSocket connection using *socket.io-client* and fetches historical messages via RESTful API.
- The client stores the current user's email and department from *localStorage* and uses them to:
  - Filter messages relevant to their department (for group chats).
- Display personal one-on-one conversations based on sender/receiver email matches.
- The chat interface displays previous messages using components like *ChatWindow, ChatSidebar*, and *MessageBubble*. Group and private chats are separated.

- The user types a message and clicks "Send" (or presses Enter):
- A message object is created including:
    - *username, department, content, timestamp, recipient_email, is_private.*
- All connected clients receive the message in real-time and append it to their local state.
- In private chats, messages are only shown to the sender and the intended recipient.
- In group chats, messages are visible only to users within the same department.
- The user can also initiate a new private chat using the *NewChatModal*, which lists employees except themselves.

3.2.7.2 Component Real-Time Chat interface description

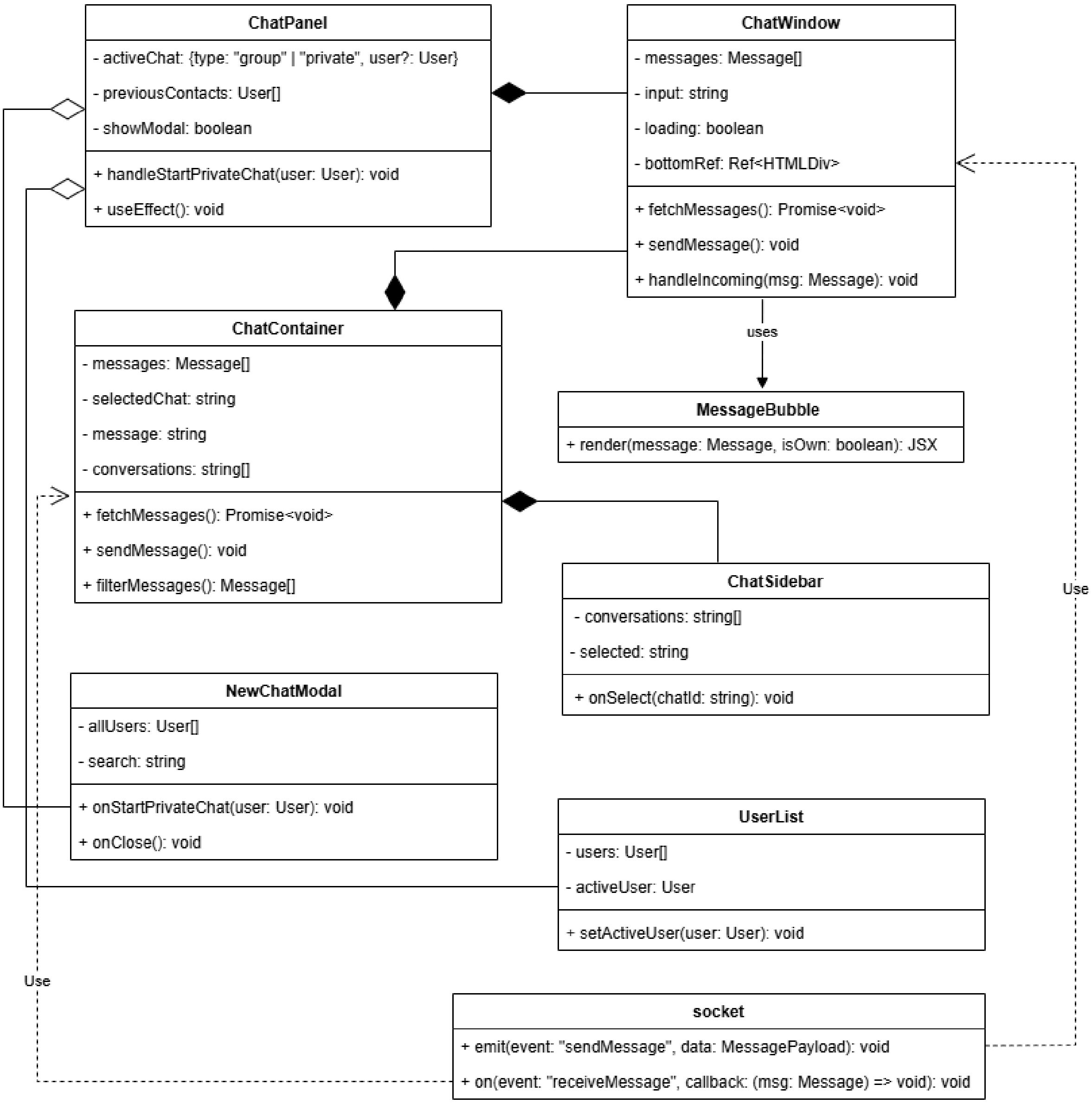
Input:

- username (String)
- department (String)
- content (String)
- recipient_email (String, optional)
- **Event:** sendMessage

Output:

- **Event:** receiveMessage
- username (String)
- content (String)
- timestamp (DateTime)
- department (String)
- recipient_email (String or null)
- is_private (Boolean)

3.2.7.3 Design Class hierarchy for component Real-Time Chat


ChatPanel
- activeChat: {type: "group" | "private", user?: User}
- previousContacts: User[]
- showModal: boolean
+ handleStartPrivateChat(user: User): void
+ useEffect(): void
ChatWindow
- messages: Message[]
- input: string
- loading: boolean
- bottomRef: Ref<HTMLDiv>
+ fetchMessages(): Promise<void>
+ sendMessage(): void
+ handleIncoming(msg: Message): void
uses
MessageBubble
+ render(message: Message, isOwn: boolean): JSX
ChatContainer
- messages: Message[]
- selectedChat: string
- message: string
- conversations: string[]
+ fetchMessages(): Promise<void>
+ sendMessage(): void
+ filterMessages(): Message[]
ChatSidebar
- conversations: string[]
- selected: string
+ onSelect(chatId: string): void
Use
NewChatModal
- allUsers: User[]
- search: string
+ onStartPrivateChat(user: User): void
+ onClose(): void
UserList
- users: User[]
- activeUser: User
+ setActiveUser(user: User): void
Use
socket
+ emit(event: "sendMessage", data: MessagePayload): void
+ on(event: "receiveMessage", callback: (msg: Message) => void): void

3.2.7.4 Restrictions/limitations for component Real-Time Chat

- **User Identification via LocalStorage**: The system depends on *localStorage* values (*userEmail, userDepartment*) for chat filtering. If these are missing or incorrect, message delivery and visibility may be compromised.
- **No End-to-End Encryption**: Messages are transmitted over *WebSocket* and stored in plaintext in the database. While transport is secured via *HTTPS* and *socket.io*, content confidentiality is not guaranteed.

- **Single Department Group Chat**: Group messages are limited to users within the same department. Cross-department communication is only possible via private messages.
- **No Offline Messaging**: The chat system does not support message queuing for offline users. Users must be online to receive messages in real-time.
- **No Message Deletion or Editing**: Once a message is sent, it cannot be deleted or edited by the user. This limits flexibility and message control.
- **No Typing Indicators or Read Receipts**: The system lacks real-time feedback features such as "user is typing" or message read/unread status.

3.2.7.5 Performance issues for component Real-Time Chat

- **No Message Pagination**: All messages are loaded at once. As the number of stored messages increases, UI responsiveness may degrade due to rendering delays and memory usage.
- **Concurrent Socket Connections**: Multiple users connected to the chat system via WebSocket can lead to high concurrency. Without socket clustering or load balancing, the server may experience performance bottlenecks under load.
- **Heavy Message Filtering on Client-Side**: Filtering logic for private vs. group messages is handled on the frontend. As the dataset grows, this increases processing time and may slow down the UI.

3.2.7.6 Design constraints for component Real-Time Chat

- **Frontend**: Implemented using React.js functional components with state and lifecycle management via hooks (*useState, useEffect*).
- **Real-Time Communication**: Utilizes *Socket.IO* for full-duplex *WebSocket* communication between clients and the Node.js server.
- **Backend**: *Node.js + Express* handles incoming messages and persists them into a *PostgreSQL* database.
- **Database**: Messages are stored in a *messages* table containing:
    - *username, content, timestamp, department, recipient_email, is_private*
- **Message Dispatch**: Messages are sent using the *sendMessage* socket event and received via *receiveMessage*.

- **UI Structure**: Componentized interface using *ChatWindow, ChatSidebar, ChatBox,* and MessageBubble, with dynamic filtering for private/group view.

3.2.7.7 Processing Detail for Component Real-Time Chat

- **User Entry**: After login, the user navigates to the Chat section or activates the floating chat widget.
- **Socket Initialization**: A WebSocket connection is established using Socket.IO (*socket.connect()*).
- **Message History Fetch**:
  - For group chat: *GET api/messages/group?department={userDepartment}*
  - For private chat:*GET/api/messages/private?user1={currentUser}&user2={recipient}*

- **Rendering**: The chat interface (*ChatWindow*) renders the conversation based on the selected mode (group/private).
- *ChatSidebar* or *UserList* shows past conversation contacts using *GET /api/messages* and *GET /employees.*
- **Message Sending**: User types a message and clicks "Send" or presses Enter.
- A message object is created: This object is emitted via the *sendMessage* WebSocket event.
- **Server-Side Handling**: The Node.js server receives the event and inserts the message into the *messages* table (with timestamp).
- It then broadcasts the message to all connected clients using the *receiveMessage* event.
- **Live Message Update**:nClients listen for *receiveMessage* and append new messages to local state in real-time.
- If the message is private, it is only shown to the sender and recipient. If public, it's shown to users in the same department.
- **Chat Interface Updates**: Messages are dynamically rendered using the *MessageBubble* component, with proper alignment and timestamp.

3.2.8. Description for Compnenent AI Task Prioritizer

3.2.8.1 Processing narrative (PSPEC) for component AI Task Prioritizer

- **Access Initiation:** The employee clicks on the "AI Task Prioritizer" button from the DashboardPage. The system navigates the user to the PrioritizerPage.

- **Task and Calendar Fetching:** The frontend sends a request to the PrioritizerAPI. The API queries the PostgreSQL database and returns the user's task list and calendar events to the frontend.
- **Task Analysis by AI:** The fetched tasks and calendar data are forwarded to the *AIPrioritizerEngine*, which performs:
  - Urgency analysis (e.g., due dates, deadlines, importance)
  - Conflict detection based on calendar availability
  - The engine returns a ranked list of tasks based on computed priorities.
- **Result Display:** The *PrioritizerPage* receives the ranked tasks and displays the prioritized task list to the employee.
- **User Decision:** The employee may either:
  - Accept the AI-suggested order
  - Manually adjust/reorder tasks.

  Any updates are sent back to the API to update task priorities in the database.

3.2.8.2 Component AI Task Prioritizer interface description

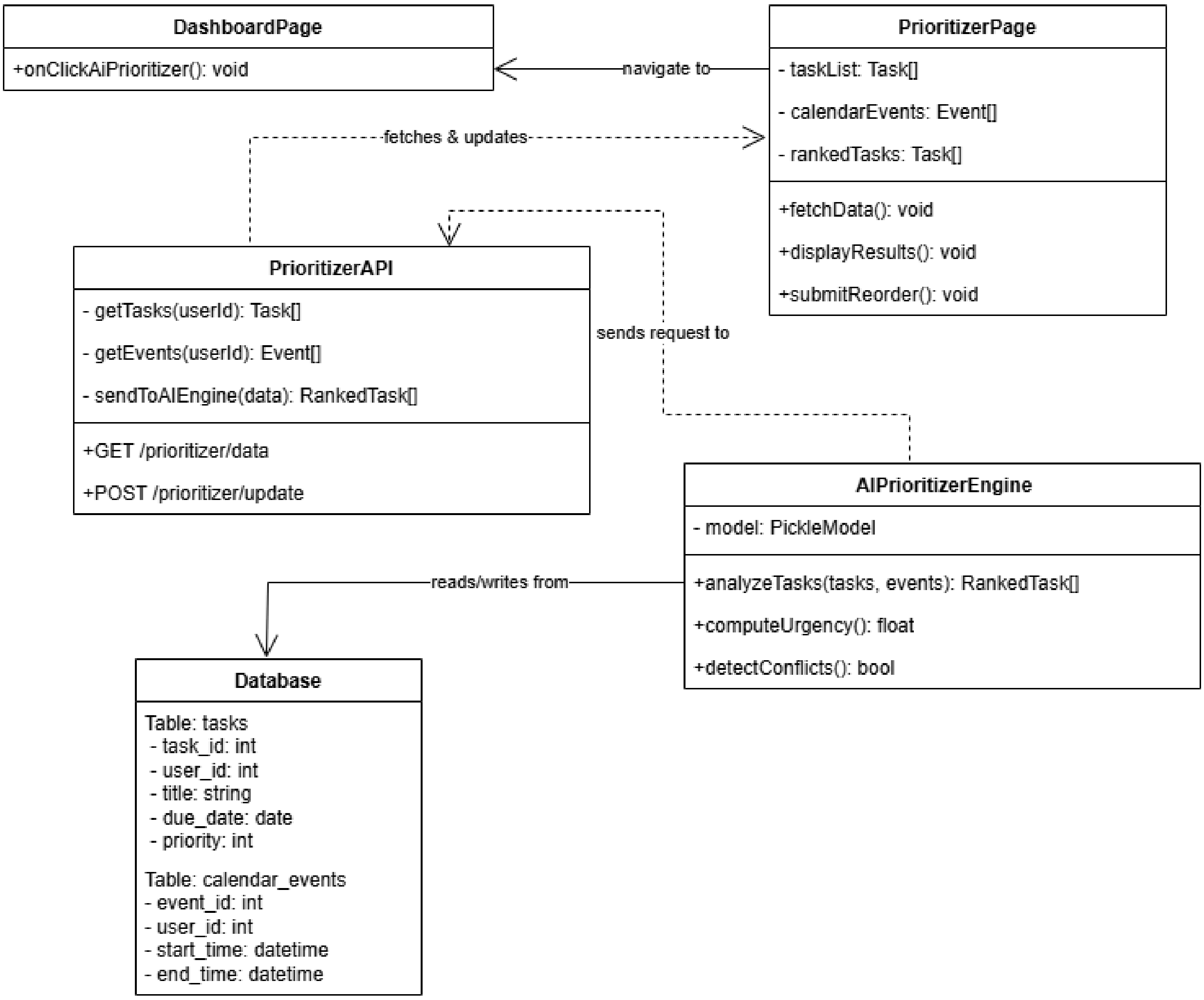
Input:

- Client-side Request
- User’s task list
- Calendar events

Output:

- Prioritized task list (from AI engine)
- If no tasks found:
  - message: "No tasks found. Please create tasks."
- If server error or timeout:
  - error: "Server unreachable. Try again later."
- If calendar is fully booked:
  - message: "Calendar conflict detected."

3.2.8.3 Design Class hierarchy for component AI Task Prioritizer



3.2.8.4 Restrictions/limitations for component AI Task Prioritizer

- **Data Availability Dependency**: The AI Task Prioritizer requires both task and calendar data to function. If either is missing or incomplete, the prioritization may be skipped or inaccurate.
- **User-Specific Scope**: The system only analyzes tasks and events for the currently logged-in user. It cannot prioritize shared or team-based tasks.
- **Time Constraints Only**: The conflict detection is based solely on calendar availability. It does not account for user energy levels, preferences, or external priorities.
- **Simplified Scoring**: Task priority scores are computed using predefined weights (e.g., due date proximity, importance flag). There's limited personalization or

context awareness.

3.2.8.5 Performance issues for component AI Task Prioritizer

- **Heavy Query Load**: Fetching both tasks and calendar events simultaneously from the database (via *JOIN* or separate queries) can result in increased latency, especially for users with many records.
- **Synchronous Processing Delay**: The analyze step (*analyzeTasks*) is a blocking operation. The system waits for the ranked task list before rendering the UI, which can slow down responsiveness.
- **No Async Task Queueing**: The prioritization logic is processed immediately upon request. There is no queuing or background processing, making the system vulnerable to overload under concurrent requests.

3.2.8.6 Design constraints for component AI Task Prioritizer

- **Backend**: Node.js + Express handles routing, authentication, and data fetching from PostgreSQL.
- **ML Service**: Implemented in Python using FastAPI, with task prioritization logic powered by a pre-trained model built using scikit-learn.
- **Model Format**: The AI model is stored locally on the Python service and is loaded at runtime.
- **Database**: PostgreSQL stores structured data including:
  - *tasks* table: user-defined tasks with due dates, priorities, and statuses.
  - *calendar_events* table: existing user schedules that help in detecting time conflicts.
- **Frontend**: Built with React.js; receives and displays prioritized tasks.

3.2.8.7 Processing Detail for Component AI Task Prioritizer

- The employee clicks the AI Task Prioritizer button from the dashboard. The system navigates to the *PrioritizerPage*.
- The frontend sends a request to the Node.js backend, including the authenticated user ID.
- The Node.js API:

    - Queries the *tasks* table to fetch all tasks for the user.
    - Queries the *calendar_events* table to get scheduled events.
    - Combines these into a structured JSON payload.
- The backend forwards the payload to the Python FastAPI service via HTTP.
- AI-Based Prioritization, The FastAPI service:
    - Loads the local model.
    - Extracts features such as task deadlines, importance levels, and schedule conflicts.
    - Computes a priority score for each task using the trained machine learning model.
- The Python service returns a list of tasks ranked by priority score.
- The backend sends the ranked task list back to the frontend. The React component renders the prioritized tasks visually.
- If the user changes the order:
    - A *POST /prioritizer/update* request is sent with new priorities.
    - The backend updates the priority field in the *tasks* table accordingly.
- If task or calendar data is missing → message: "No tasks to prioritize."
- If ML service is unavailable → message: "AI service is currently unreachable."

3.2.9. Description for Component Smart Meeting Suggestion

3.2.9.1. Processing narrative (PSPEC) for component Smart Meeting Suggestion

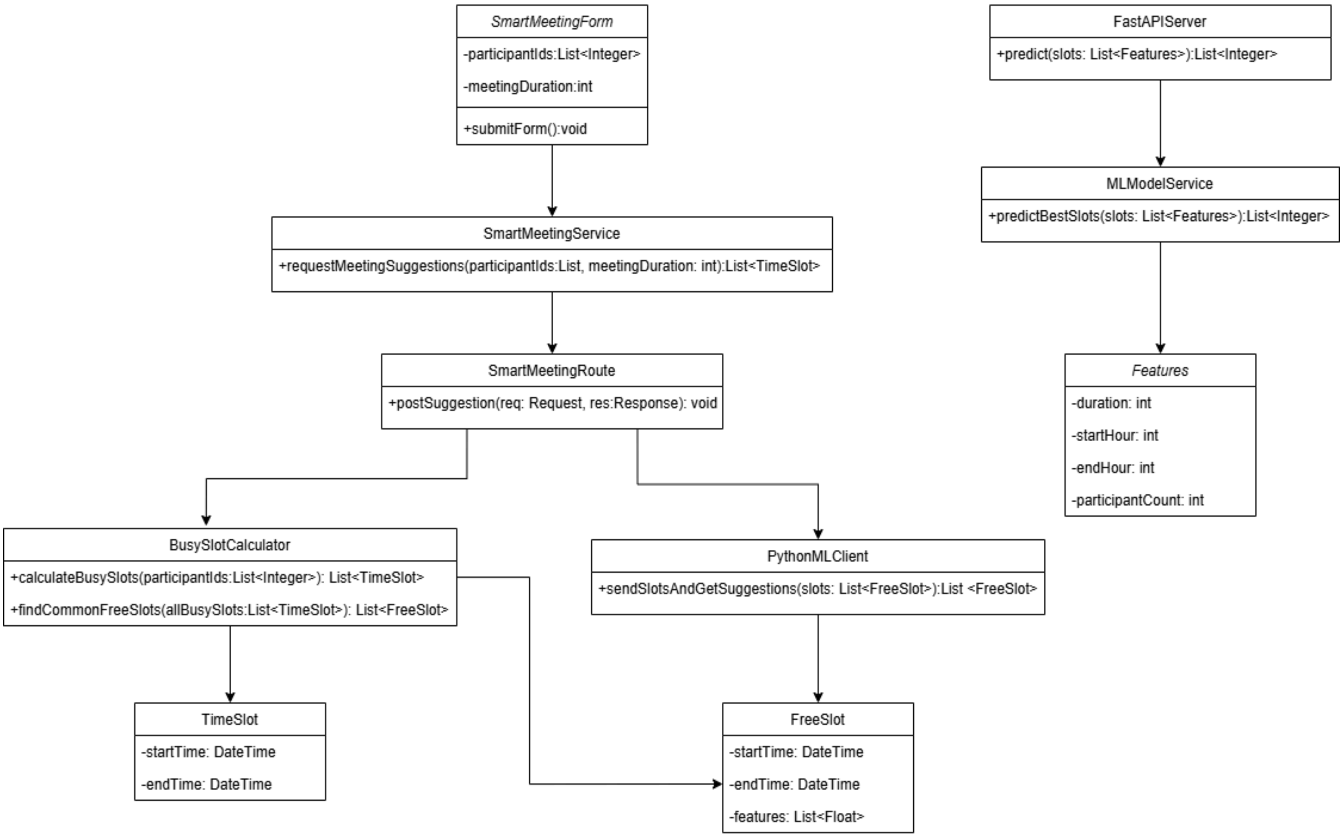
- Enables the user to get AI-based meeting time suggestions.
- Accepts a list of participant IDs and a desired meeting duration.
- Retrieves each participant's current task, todo, and meeting data from the database.
- Identifies each participant's busy time slots.
- Calculates common free time blocks among all selected users.
- Prepares feature data based on these slots (e.g. duration, time of day, number of participants).
- Sends the feature data to a Python-based ML service.
- ML model predicts and ranks the best meeting time options.
- Returns top suggested time slots to the frontend for user selection.

3.2.9.2. Component Smart Meeting Suggestion interface description

Inputs: participantIds , meetingDuration

Outputs: startTime , endTime

3.2.9.3 Design Class hierarchy for component Smart Meeting Suggestion



SmartMeetingForm
-participantIds:List<Integer>
-meetingDuration:int
+submitForm():void
FastAPIServer
+predict(slots: List<Features>):List<Integer>
MLModelService
+predictBestSlots(slots: List<Features>):List<Integer>
SmartMeetingService
+requestMeetingSuggestions(participantIds:List, meetingDuration: int):List<TimeSlot>
SmartMeetingRoute
+postSuggestion(req: Request, res:Response): void
Features
-duration: int
-startHour: int
-endHour: int
-participantCount: int
BusySlotCalculator
+calculateBusySlots(participantIds:List<Integer>): List<TimeSlot>
+findCommonFreeSlots(allBusySlots:List<TimeSlot>): List<FreeSlot>
PythonMLClient
+sendSlotsAndGetSuggestions(slots: List<FreeSlot>):List <FreeSlot>
TimeSlot
-startTime: DateTime
-endTime: DateTime
FreeSlot
-startTime: DateTime
-endTime: DateTime
-features: List<Float>

3.2.9.4 Restrictions/limitations for component Smart Meeting Suggestion

- Requires at least two participants.
- Suggests time slots only within standard working hours (e.g. 09:00–18:00).
- ML model accuracy depends on the quality of available busy/free time data.
- No overlapping events allowed during the suggested slot.

3.2.9.5 Performance issues for component Smart Meeting Suggestion

- Each request triggers fresh queries for all selected participants.
- As participant count increases, busy slot merging and common time finding can become slower.
- ML model prediction is fast (~1 second), but can be optimized further with caching.

3.2.9.6 Design constraints for component Smart Meeting Suggestion

- Backend: Node.js + Express
- ML: Python (FastAPI + scikit-learn)
- Inter-process communication: REST API between Node.js and Python
- Database: PostgreSQL with assigned_tasks, todos, and meetings tables
- ML model stored locally as meeting_model.pkl

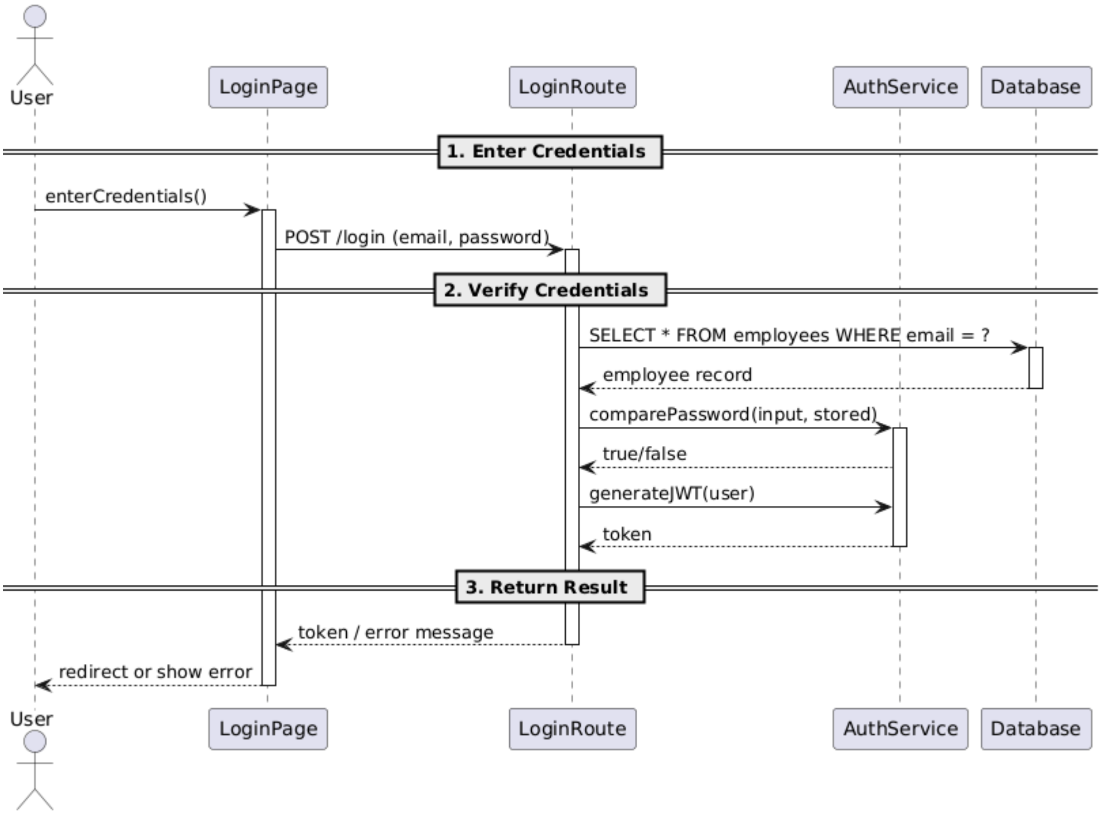
3.2.9.7 Processing detail for each operation of component Smart Meeting Suggestion

- User opens Smart Suggest and selects participants + duration.
- System fetches participants' busy times.
- Common free slots are calculated.
- Slots are sent to the Python ML model.
- Best meeting times are predicted and returned.
- Suggestions are displayed to the user.

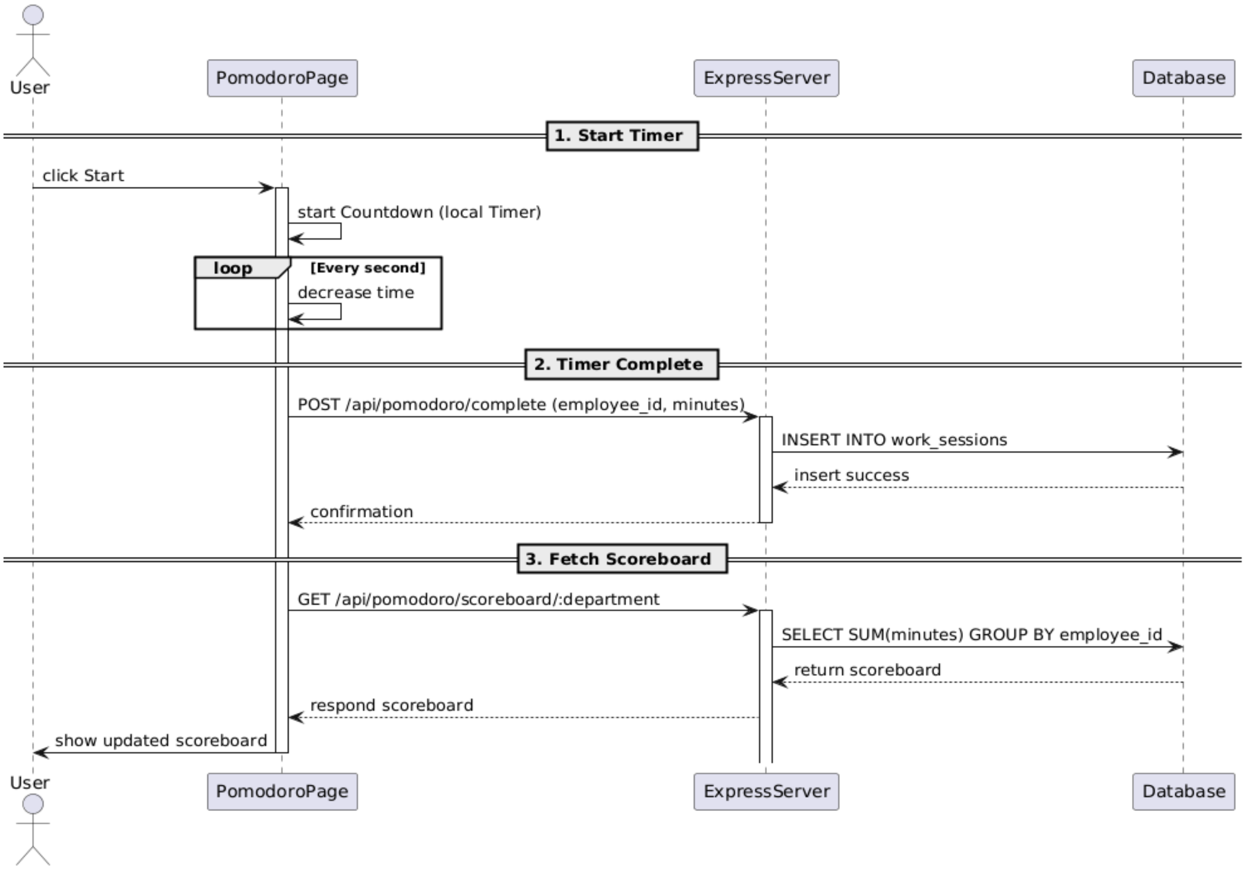
3.3. Dynamic Behavior for Components

3.3.1. Interaction Diagrams

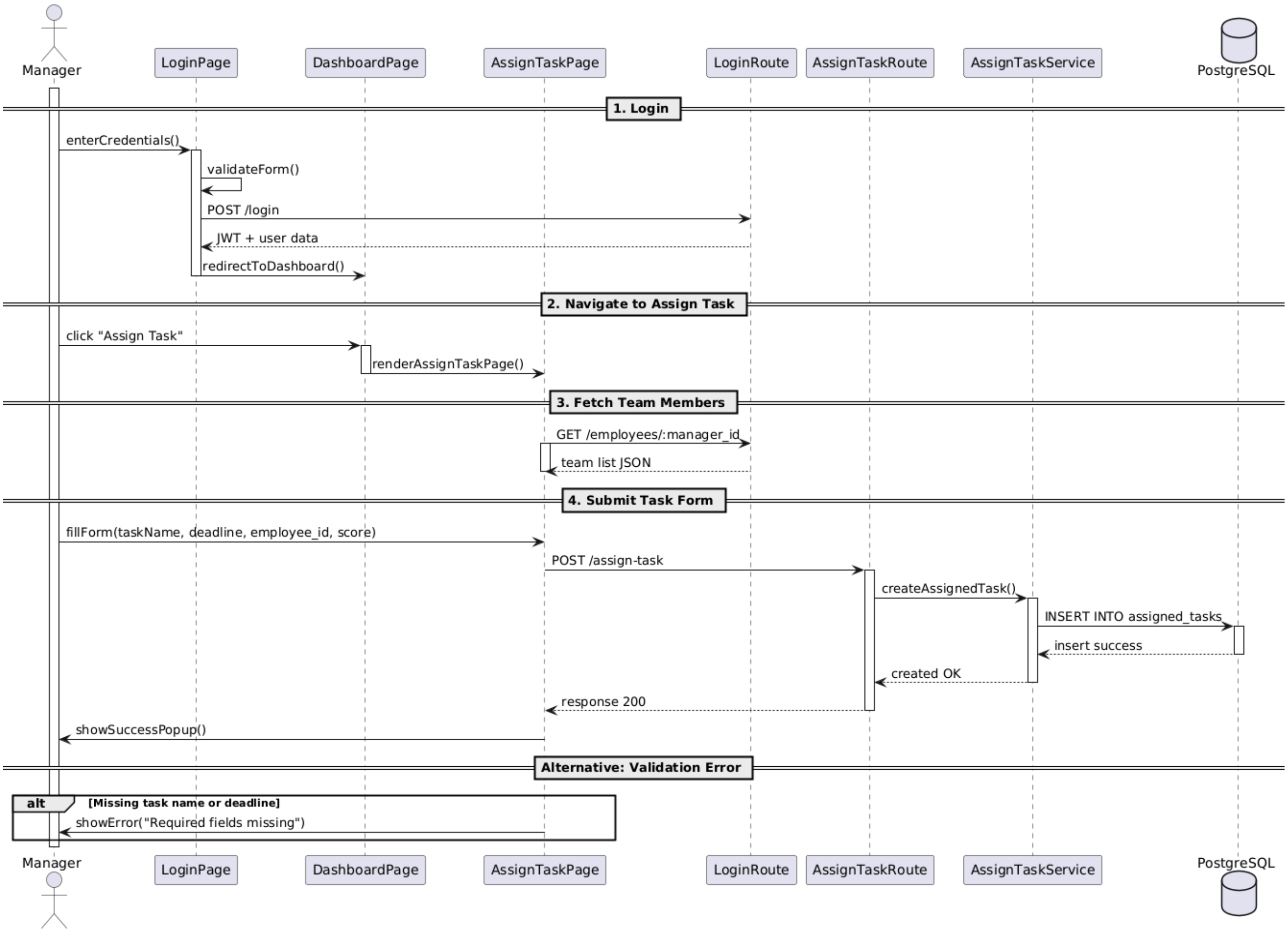
3.3.1.1. Sequence Diagram For **User Login**



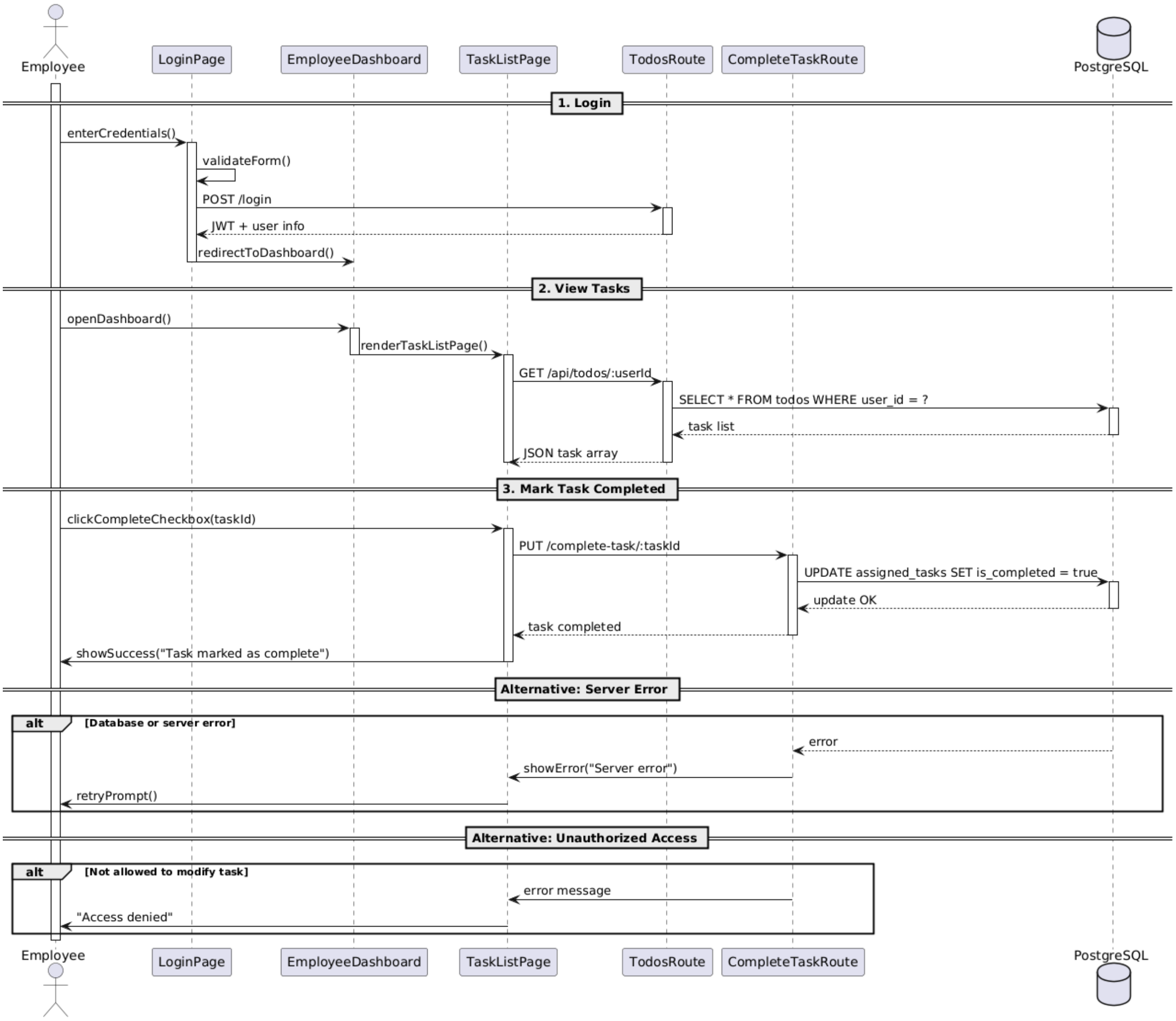
3.3.1.2. Sequence Diagram For **Pomodoro Timer**



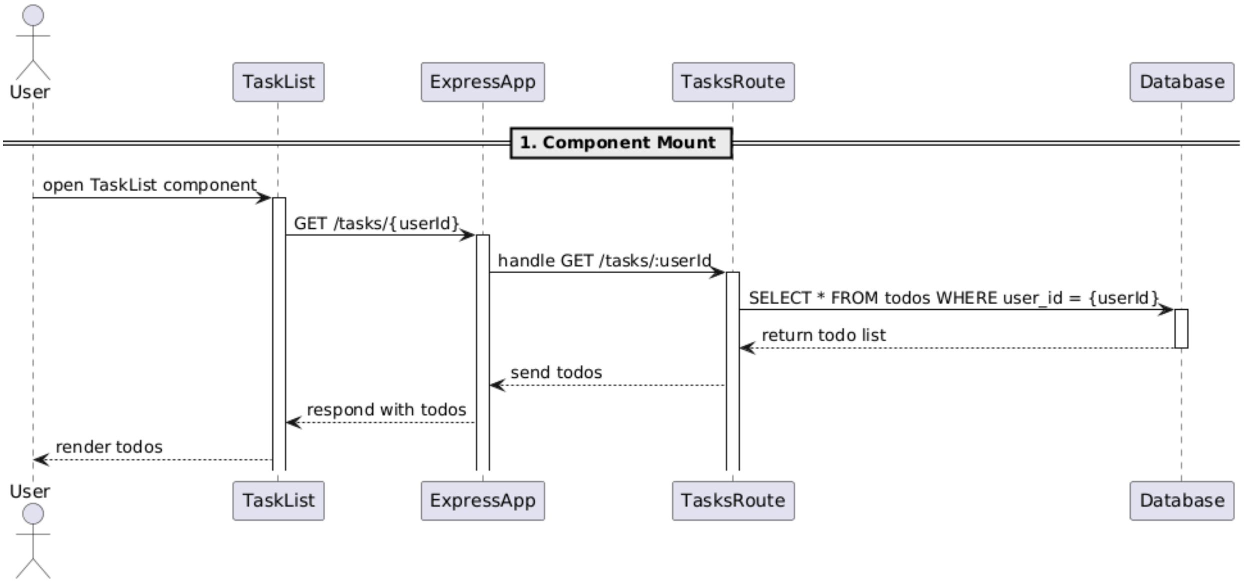
3.3.1.3. Sequence Diagram For **Assign Task**



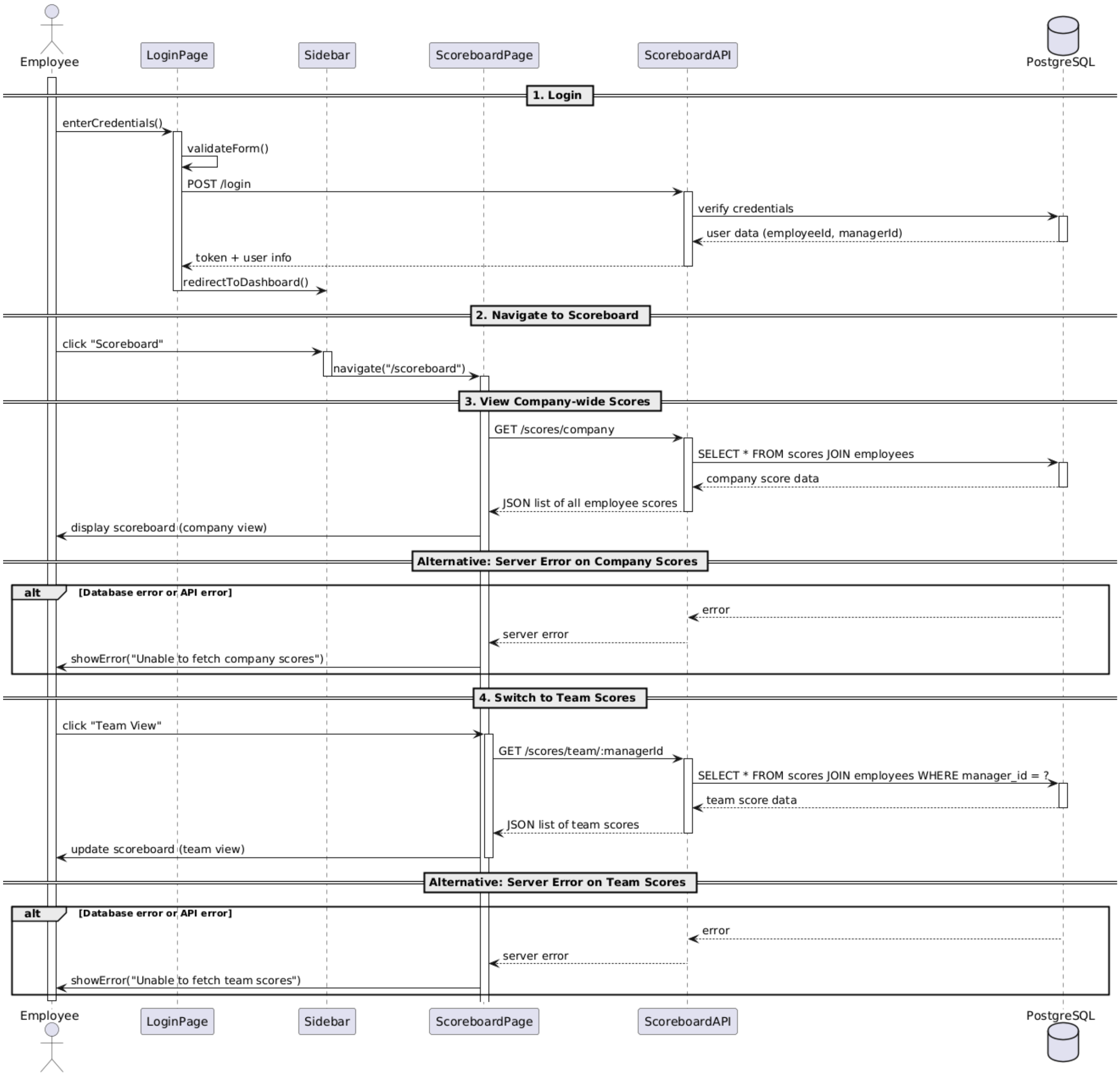
3.3.1.4. Sequence Diagram For **Mark Task as Completed**



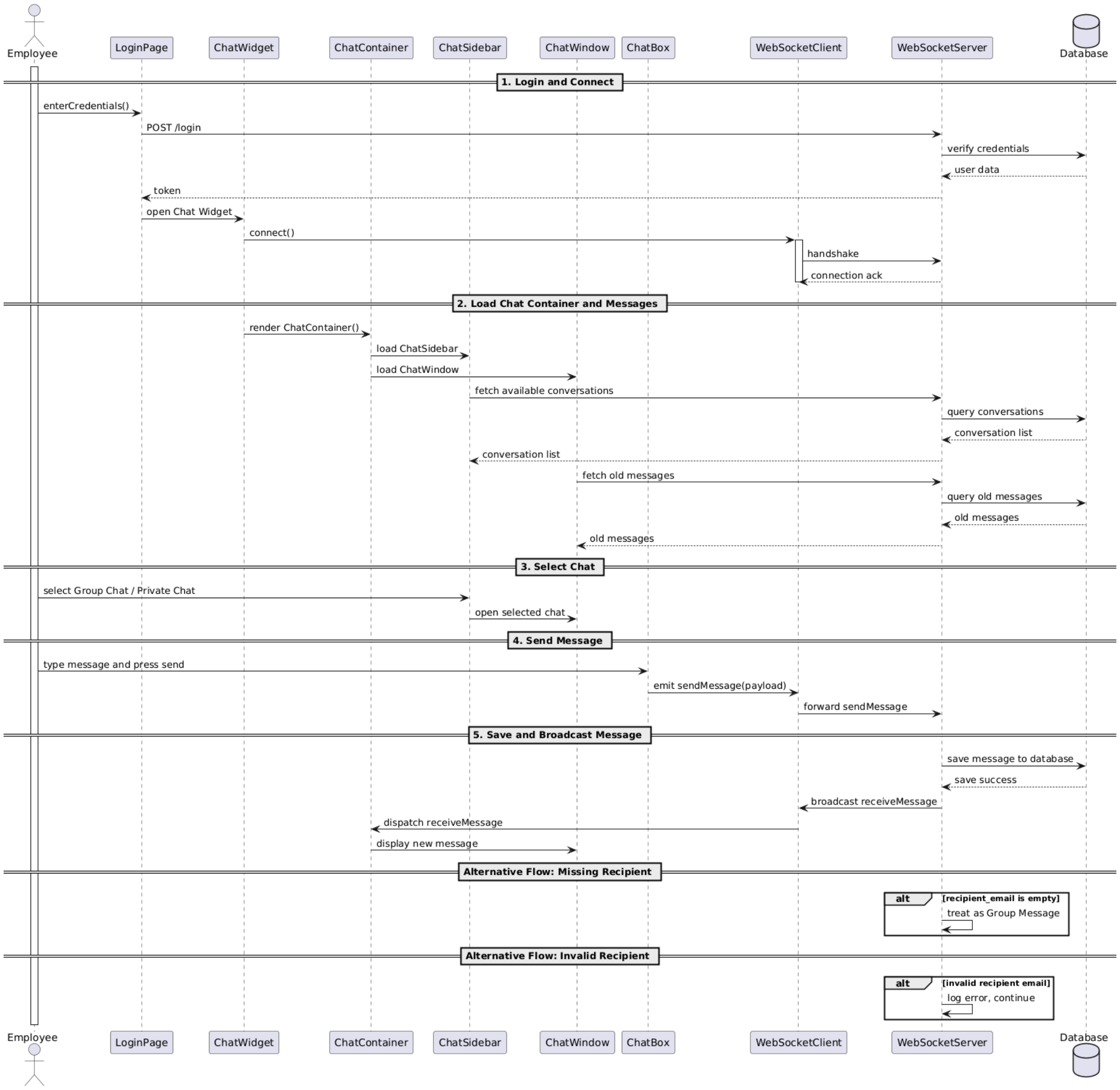
3.3.1.5. Sequence Diagram For **Add Personal Task**



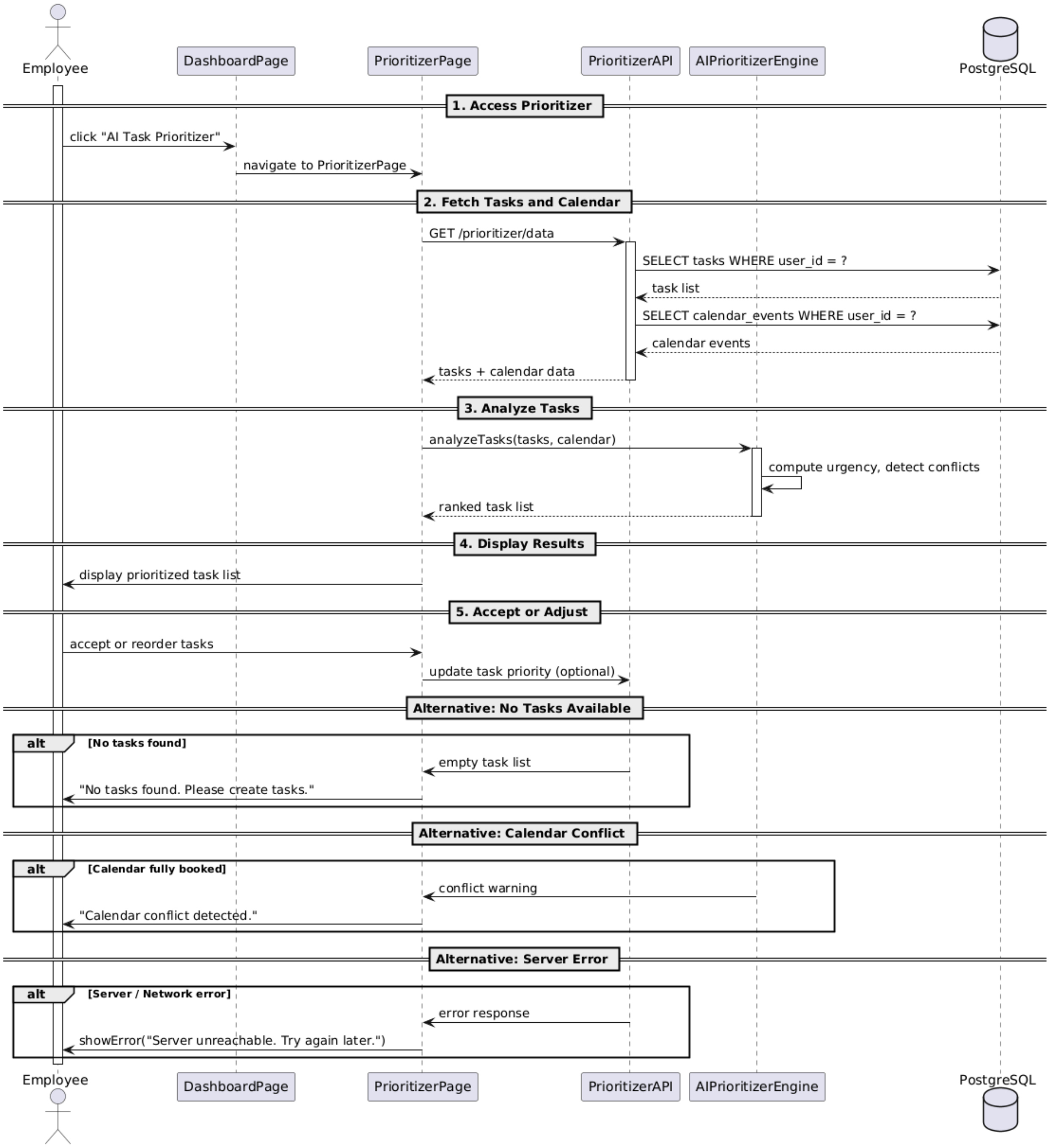
3.3.1.6.Sequence Diagram For **View Team and Company-wide Scoreboard**


Employee
LoginPage
Sidebar
ScoreboardPage
ScoreboardAPI
PostgreSQL
1. Login
enterCredentials()
validateForm()
POST /login
verify credentials
user data (employeeId, managerId)
token + user info
redirectToDashboard()
2. Navigate to Scoreboard
click "Scoreboard"
navigate("/scoreboard")
3. View Company-wide Scores
GET /scores/company
SELECT * FROM scores JOIN employees
company score data
JSON list of all employee scores
display scoreboard (company view)
Alternative: Server Error on Company Scores
alt
[Database error or API error]
error
server error
showError("Unable to fetch company scores")
4. Switch to Team Scores
click "Team View"
GET /scores/team/:managerId
SELECT * FROM scores JOIN employees WHERE manager_id = ?
team score data
JSON list of team scores
update scoreboard (team view)
Alternative: Server Error on Team Scores
alt
[Database error or API error]
error
server error
showError("Unable to fetch team scores")
Employee
LoginPage
Sidebar
ScoreboardPage
ScoreboardAPI
PostgreSQL

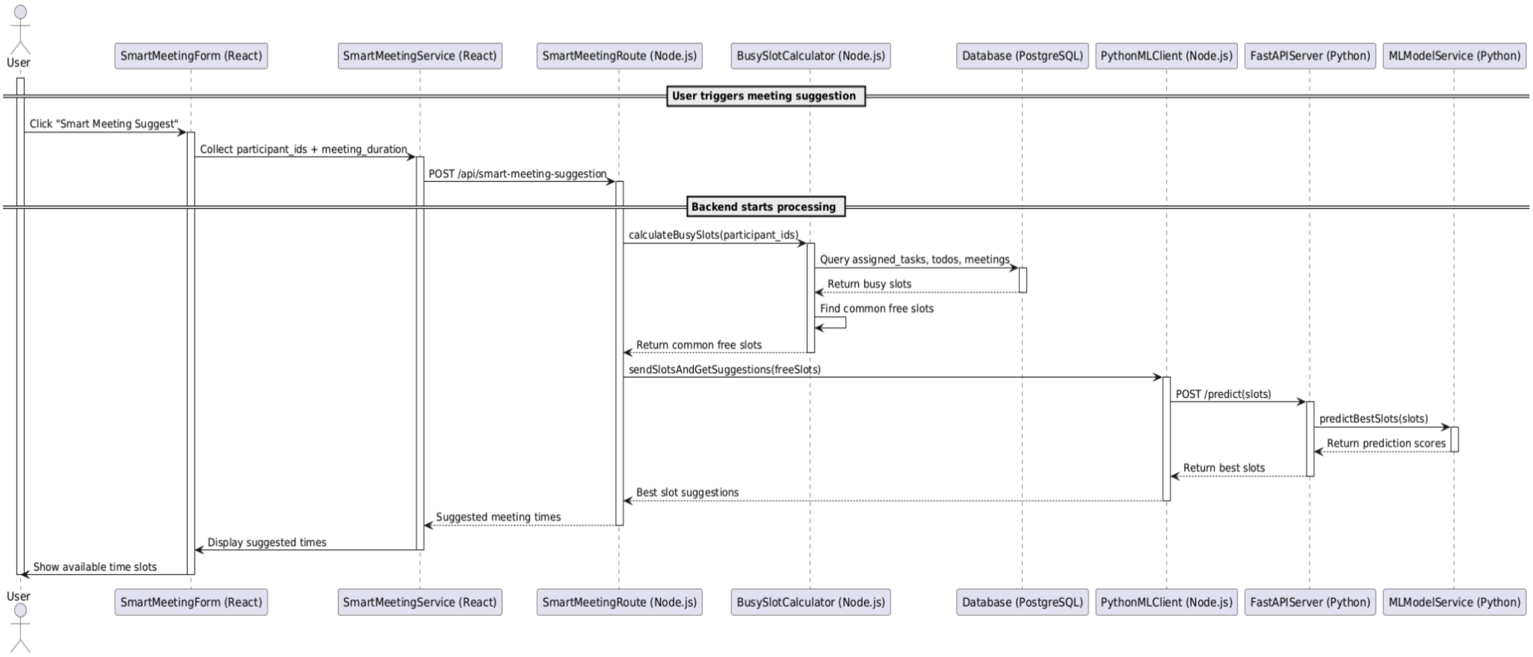
3.3.1.7. Sequence Diagram For **Real-Time Chat**



3.3.1.8. Sequence Diagram For **AI Task Prioritizer**


Employee
DashboardPage
PrioritizerPage
PrioritizerAPI
AIPrioritizerEngine
PostgreSQL
1. Access Prioritizer
click "AI Task Prioritizer"
navigate to PrioritizerPage
2. Fetch Tasks and Calendar
GET /prioritizer/data
SELECT tasks WHERE user_id = ?
task list
SELECT calendar_events WHERE user_id = ?
calendar events
tasks + calendar data
3. Analyze Tasks
analyzeTasks(tasks, calendar)
compute urgency, detect conflicts
ranked task list
4. Display Results
display prioritized task list
5. Accept or Adjust
accept or reorder tasks
update task priority (optional)
Alternative: No Tasks Available
alt
[No tasks found]
empty task list
"No tasks found. Please create tasks."
Alternative: Calendar Conflict
alt
[Calendar fully booked]
conflict warning
"Calendar conflict detected."
Alternative: Server Error
alt
[Server / Network error]
error response
showError("Server unreachable. Try again later.")
Employee
DashboardPage
PrioritizerPage
PrioritizerAPI
AIPrioritizerEngine
PostgreSQL

3.3.1.9. Sequence Diagram For **Smart Meeting Suggestion**


User
SmartMeetingForm (React)
SmartMeetingService (React)
SmartMeetingRoute (Node.js)
BusySlotCalculator (Node.js)
Database (PostgreSQL)
PythonMLClient (Node.js)
FastAPIServer (Python)
MLModelService (Python)
User triggers meeting suggestion
Click "Smart Meeting Suggest"
Collect participant_ids + meeting_duration
POST /api/smart-meeting-suggestion
Backend starts processing
calculateBusySlots(participant_ids)
Query assigned_tasks, todos, meetings
Return busy slots
Find common free slots
Return common free slots
sendSlotsAndGetSuggestions(freeSlots)
POST /predict(slots)
predictBestSlots(slots)
Return prediction scores
Return best slots
Best slot suggestions
Suggested meeting times
Display suggested times
Show available time slots
User
SmartMeetingForm (React)
SmartMeetingService (React)
SmartMeetingRoute (Node.js)
BusySlotCalculator (Node.js)
Database (PostgreSQL)
PythonMLClient (Node.js)
FastAPIServer (Python)
MLModelService (Python)

4. Restrictions, limitations, and constraints

This section outlines key limitations and design restrictions that influenced the development, scalability, and deployment of the WorkSpire system.

- **AI Service Dependency**: The Smart Meeting Suggestion and Task Prioritization modules depend on external Python services.
- **Model Accuracy**: The AI models are trained on limited historical data. Their accuracy may degrade if users' task or calendar usage patterns change significantly.
- **Scalability of Real-Time Chat**: The current Socket.io setup may encounter bottlenecks with hundreds of concurrent users. Load balancing or clustered socket infrastructure is not in place.
- **Security Boundaries**: Although JWT and HTTPS are implemented, multi-factor authentication and role-based API-level restrictions are still basic.
- **Database Constraints**: Queries related to prioritization and scheduling are complex joins over multiple tables (e.g., assigned_tasks, todos, meetings) and may lead to performance degradation under high load.

- **Frontend Limitations**: Real-time UI updates are limited to certain views (e.g., chat). Task updates and calendar changes require full refresh or polling due to lack of websocket binding.
- **Browser-Specific Behavior**: Some features like the Pomodoro Timer are not persistent if the tab is closed, unless localStorage is enabled.

5. Conclusion

WorkSpire is a modern, modular web application that combines traditional productivity tools with intelligent enhancements to improve team efficiency, collaboration, and time management. Through its task management system, performance tracking, Pomodoro timer, real-time chat, and AI-powered modules, it provides a unified platform for employee engagement.

The software architecture emphasizes separation of concerns, scalability, and extensibility — making use of a JavaScript-based full stack and integrating external Python microservices where appropriate. Although there are limitations tied to performance, dependency on ML services, and real-time scalability.

WorkSpire demonstrates a forward-looking solution to workplace collaboration challenges and is structured to evolve with emerging team needs and technological advances.

6.Work Distribution

● Purpose ● Statement of scope ● Software context ● Major constraints ● Definitions, Acronyms, and Abbreviations ● References	Zeynep Kurt Damla Kundak Ayşe Nisa Şen
● Design Assumptions and Dependencies ● General Constraints ● System Environment ● Development Methods ● Architecture diagram ● Description for Component User Login ● Description for Component Add Personal Task ● Description for Component Pomodoro Timer ● Sequence Diagram For User Login ● Sequence Diagram For Add Personal Task ● Sequence Diagram For Pomodoro Timer ● Restrictions, limitations, and constraints ● Conclusion	Zeynep Kurt
● Description for Component Assign Task ● Description for Component Mark Task as Completed ● Description for Component Smart Meeting Suggestion ● Sequence Diagram For Assign Task ● Sequence Diagram For Mark Task as Completed ● Sequence Diagram For Smart Meeting Suggestion	Ayşe Nisa Şen
● Description for Component AI Task Prioritizer ● Description for Component View Team and Company-wide Scoreboard ● Description for Component Real-Time Chat ● Sequence Diagram For AI Task Prioritizer ● Sequence Diagram For View Team and Company-wide Scoreboard ● Sequence Diagram For Real-Time Chat	Damla Kundak