ChatGPT

Campfire System User Journeys and Requirements

Administrator Role

Account Setup & Configuration

- **Explicit:** On first launch (when no user accounts exist), an **Administrator** creates the initial account via a special sign-up flow [1]. The first user becomes the account owner (with administrator role) and establishes the single-tenant **Account** record (including a unique join code for inviting others). After this **first-run** setup, the app redirects the admin to the main interface [2].
- **Explicit:** Administrators can edit global account settings (name and logo) via an account edit page [3] [4]. Updating triggers `AccountsController#update` to save the new account name or upload a new logo [5] [4]. The account logo is stored as an attachment and served to clients (with caching for performance) [6] [7]. Removing the logo restores a default icon [8] [9].
- **Explicit:** Admins can customize the chat's appearance by providing **custom CSS styles**. In the *Customize Styles* interface, the admin enters CSS which is saved in the account's `custom_styles` field [10] [11]. This CSS is injected into every page for branding or theming purposes [12] [13] (UX requirement: allow white-labeling the app's look and feel).
- **Explicit:** The system generates a **join link** (invitation URL) for the account. Administrators can retrieve or regenerate this invite link. Clicking "Regenerate join link" resets the account's `join_code` (12-character token split with dashes) [14] [15]. The new join code is applied immediately and shown to the admin [16] [17]. *(Security:* only admins can reset the join code [16]; a join code is required for any new user signup, preventing unauthorized registrations).
- **Inferred UX:** The **Invite Users** UI provides the shareable join URL (e.g. `.../join/ABCD-EFGH-IJKL`) for the admin to copy and send to team members [18]. A one-click "Copy Invite Link" button is available for convenience (likely implemented via a clipboard-copy control).
- **Explicit:** If an invitee visits the join URL, the system matches the token and allows them to fill out the new user form [19] [20]. If the token is invalid, the server returns 404 (preventing random sign-ups) [21]. After successful signup, the new user is automatically signed in and taken to the chat UI [22].
- **Performance:** The application is single-tenant (one account per deployment) but can serve many users and rooms. Public rooms ("open" rooms) are accessible to all users by default [23] [24]. For entirely isolated groups, separate deployments are required [25]. Admins should plan capacity accordingly (e.g. separate instances for different organizations).

User Management and Roles

- **Explicit:** Administrators have a dedicated **User Management** panel that lists all active non-bot users [26]. Up to 500 users are shown per page with pagination for large user bases [26]. This allows scanning and managing users easily (UX: sortable by name by default [26]).
- **Explicit:** Admins can **promote or demote** users between *member* and *administrator* roles. In the user list, an admin can change a user's role via a form (likely a dropdown or toggle). Submitting updates triggers `Accounts::UsersController#update`, which permits only `role` changes and

persists them [27] [28] . After updating, the admin is returned to the list with a success checkmark (✓) as feedback [5] .

- **Explicit:** Admins can **deactivate (remove) a user**. In the interface, an option like "Remove User" calls `Accounts::UsersController#destroy` for that user [29] . The controller calls `@user.deactivate`, which likely marks the user as inactive (preventing login) without fully deleting their data (so past messages remain attributed). The user disappears from the active list post-deactivation. *(Security:* Only admins can remove users [30] , preventing regular users from self-elevating privileges or removing others).
- **Explicit:** The system distinguishes **roles**: "member" (regular user), "administrator", and "bot" [31] . Administrator accounts have full permissions (e.g. can manage settings, rooms, and users). Regular members cannot access administrative endpoints (controllers enforce `ensure_can_administer` before admin-only actions [32] [30]).
- **Inferred UX:** The admin user list likely shows each user's name, email, and role, with controls (dropdown or buttons) to change role or deactivate. The UI might prevent the admin from demoting or removing themselves to avoid leaving the system without an admin (not explicitly shown in code, but a common UX safeguard).

Room Administration and Moderation

- **Explicit:** Administrators (or the creator of a room) can **delete any chat room**. In the room's settings, an "Delete Room" action triggers `RoomsController#destroy`. This checks that the current user can administer the room and then permanently removes the room and all its messages [33] [34] . A Turbo broadcast then removes the room from all users' sidebars in real-time [35] . The admin is redirected to the homepage after deletion [33] . *(Functional requirement:* ensure room deletion cleans up associated data like memberships and messages).
- **Explicit:** Admins can always administer (manage settings of or delete) any room, including those they didn't create [36] [37] . The permission check `Current.user.can_administer?(@room)` returns true for admins even if they are not a member of that room [36] . This means an admin could enter or remove a private room without being invited. However, the UI and controllers typically find rooms via the admin's membership list [38] . If an admin is not a member of a private room, attempting to access it by URL yields "Room not found or inaccessible" [39] unless the admin manually adds themselves. (This is a subtlety: the code's security check would allow it, but the room lookup uses membership, effectively hiding non-member closed rooms from admin. In practice, admins should add themselves via the membership management UI if they need to moderate a closed room.)
- **Explicit:** Admins can manage **room access controls** for private rooms. For any **Closed** room, an admin or the room's creator can edit its membership list. In `Rooms::ClosedsController#edit`, the system presents two lists: users currently in the room and all other users to potentially add [40] . The admin can check/uncheck users and save changes. On save, `#update` applies additions and removals: new users get memberships granted, and unchecked users are revoked (removed) [41] [42] . The UI updates in real-time: added members see the room appear in their sidebar, removed ones see it disappear (accomplished via Turbo Streams broadcast) [43] [44] .
- **Inferred UX:** The room settings page for an admin likely includes controls for:
- Changing the room name (with a live preview or after-save feedback).
- Toggling room type between **Open** (visible to all) and **Closed** (invite-only). A switch control is present in the room edit view to "Give everyone access to this room" (open) vs. "only some access" (closed) [45] [46] . If an admin toggles a closed room to open, the backend converts the model type

(`Rooms::Closed` -> `Rooms::Open`) and automatically grants access to all users [47]. Conversely, switching open to closed triggers conversion (`Rooms::Open` -> `Rooms::Closed`) and then allows selecting specific members.
- Deleting the room (with a confirmation prompt, given the destructive action).
- **Explicit (Performance):** Changing a room's open/closed status for a large user base is handled efficiently. When a room becomes **Open**, the system bulk-inserts memberships for *all* users via `memberships.grant_to(User.active)` inside a transaction [48]. This uses a set-based SQL insert rather than individual loops, which is important for performance with many users.
- **Explicit (Security):** All admin actions are protected behind authentication and authorization checks. Controllers that modify account, users, or rooms verify admin rights (`ensure_can_administer`) before proceeding [32] [30]. This ensures regular members cannot access these endpoints even if they guess the URLs.

Bot Account Management

- **Explicit:** Administrators can create and manage **Bot users** for integrations [49] [50]. In the *Bots* admin panel, an admin sees a list of existing bots (name and perhaps last activity) [51] and can click "New Bot". Creating a bot involves providing a bot name and optional avatar, plus an integration **Webhook URL** [52]. On submission, the backend generates a new `User` with role "bot" and a secure random token as its API key [53] [54]. If a webhook URL was provided, a Webhook record is associated with the bot [53].
- **Explicit:** The bot's unique credentials (the **bot key**) are available to the admin after creation. A bot key is a string composed of the bot's ID and token (e.g. `42-ABCDEF12...`) used for authenticating API calls [55] [56]. The UI likely displays this key with a copy button since it must be provided to the external integration.
- **Explicit:** Admins can edit a bot's properties. Via `Accounts::BotsController#edit` and `#update`, the admin may change the bot's name or avatar, or update its webhook URL [57] [58]. The system wraps updates in a transaction to ensure both the user record and webhook URL update atomically [59] [60]. Removing the webhook URL will disable incoming webhooks for that bot (the code deletes the Webhook record if URL is cleared) [61].
- **Explicit:** Admins can **regenerate a bot's API token** if needed (for example, if it was compromised). In the bot list, a "Reset API Key" action calls `Accounts::Bots::KeysController#update`, which finds the bot and generates a new random token [62] [63]. The bot's bot_key changes (since it includes the token) and the new key should be provided to the external service. The old key will no longer work for authentication (security requirement: allow credential rotation).
- **Explicit:** Admins can deactivate (delete) a bot user. Using `Accounts::BotsController#destroy`, the admin triggers the bot's `deactivate` method [64] [65]. Like regular users, this likely marks the bot as inactive; it will stop receiving messages or posting, but historical bot messages remain. The bot is removed from the active bots list in the UI.
- **Inferred UX:** The bots management screen likely highlights that bots are integrations. It may show the last few messages sent by each bot or their last active time to help the admin monitor activity. Each bot entry provides buttons: *Edit*, *Reset Key*, *Deactivate*. The admin should copy the bot key upon creation; if lost, resetting is the way to get a new one (the system might not show the token again for security reasons after initial creation).

Performance and Security Considerations (Admin)

- **Performance:** Admin operations that could affect many users (like making a room open to hundreds of users or sending push notifications to all) are optimized in the code (bulk inserts, background jobs, caching). For example, turning a room open grants memberships to all users in one query [66] [67]. Admins should still be mindful of system load when adding a large number of users or bots at once.
- **Security:** Admin actions are audited by virtue of role checks. Only administrators can invite users or promote roles [30] [28]. Passwords for all users are stored securely (the code uses Rails' `has_secure_password` via Devise or similar – implied but not shown in excerpt). Sessions are cookie-based with HTTP-only, SameSite protection [68] [69]. It's recommended that the site run under SSL (the Docker config supports Let's Encrypt via setting `SSL_DOMAIN` [70] [71] and Rails can be configured to force SSL in production).
- **Security:** The admin's presence is advertised in the UI for transparency. For instance, a help contact snippet shows the first admin's email (as "Contact support") [72] – assuming the first admin is considered the owner. This is more a UX/help feature, ensuring users know who the admin is.
- **Security:** The system defends against certain web attacks. Link unfurling is restricted from accessing internal IPs (to prevent SSRF) [73]. HTML content in messages is sanitized on rendering [74]. All file uploads are handled by ActiveStorage, which provides verified download URLs and content-type checking. These safeguards benefit all users but are crucial for admins to know as they maintain the system.

Regular User (Member) Role

Authentication & Sessions

- **Explicit:** Users log in with an email address and password. The **Login** form (`SessionsController#new`) is accessible to unauthenticated users [75]. On submission, `SessionsController#create` verifies credentials against an active user record (password authentication) [76]. If valid, a new session is created and the user is redirected to the chat interface [76]. If invalid, the login form renders an "unauthorized" error state (e.g. "Too many requests or unauthorized.") [76] [77].
- **Explicit:** The system implements **rate limiting** on login attempts to protect against brute force. It allows at most 10 login requests per 3 minutes for a given client, otherwise immediately responds with status 429 Too Many Requests [75] [78]. This is a security requirement to slow down password-guessing attacks.
- **Explicit:** Upon successful login, a persistent session cookie is issued. The `start_new_session_for(user)` helper creates a Session record and sets a signed, permanent cookie `session_token` [79] [68]. The cookie is HTTP-only and SameSite=Lax for security [68], and in production it would be secure (SSL) if force_ssl is enabled. This allows the user to remain logged in across browser restarts.
- **Explicit:** Users can **log out** via `SessionsController#destroy`. This action clears the session cookie and resets authentication state [80]. If the user had a registered push notification subscription, the logout process will remove that subscription server-side (so the device stops receiving chat notifications) [81]. The user is then redirected to a post-logout page (likely the login screen or marketing page).

- **Explicit:** Users can transfer a login session to another device using a **Session Transfer** feature. In their profile area, a user can generate a QR code or link containing a one-time **transfer token** [82] [83]. When scanned or visited on a new device, `Sessions::TransfersController#update` finds the user by the transfer token and logs that user in on the new device [82]. The token is time-limited (expires after 4 hours by default) [84] for security. This provides a smooth UX for logging into a mobile app or another browser without retyping credentials. *(Security:* The transfer token is single-use; after a successful login, it becomes invalid. An invalid or expired token yields a 400 Bad Request [82].)
- **Inferred UX:** The login page is simple, requiring email and password. On errors, the error message ("Unauthorized" or "Too many attempts") is shown inline. If it's the very first user ever, the login page will redirect to the **First Run** sign-up page [85], since no account exists yet. The session transfer likely appears as a QR code in the user's settings; the user scans it with their phone's camera to instantly log in on mobile. The presence of a `QrCodeController` that generates a QR image from a given URL [86] supports this flow (the controller decodes a token and renders an SVG QR code).

User Profile & Account Settings

- **Explicit:** Every user has a **Profile** page where they can view and edit their personal information. The profile is accessible via a stable URL (e.g. `/users/me/profile` as defined in routes [87]). `Users::ProfilesController#show` displays the user's details and perhaps an overview of their rooms (direct and group chats) [88]. `#update` allows changing profile fields [89].
- **Explicit:** Users can update their **display name**, **avatar (profile picture)**, **email address**, **password**, and a short **bio** from the profile settings [90] [91]. All these fields are optional except name and email. When a user submits changes, the server saves the new data and redirects back to the profile with a success notice [89] [92]. If an avatar image was uploaded, the notice informs the user that it may take up to 30 minutes to update everywhere [92] (because cached avatars might take time to expire across the system).
- **Explicit:** Users can remove their profile avatar. Hitting the **Remove Avatar** action calls `Users::AvatarsController#destroy`, which deletes the avatar attachment for the current user [93]. After removal, the UI will show either a default avatar (like initials or a placeholder graphic). Indeed, when avatars are fetched, if none is attached, the system either renders a default SVG with the user's initials or a stock image for bots [94] [95].
- **Explicit:** The application enforces a minimum browser version for full functionality. A concern `AllowBrowser` injects a before_action that checks the User-Agent against supported versions and renders an **"Incompatible Browser"** page if the browser is too old [96] [97]. This ensures a baseline UX (e.g. modern JS for Turbo and WebSockets). Users are expected to use recent Chrome, Firefox, Safari, etc.
- **Inferred UX:** The profile UI likely includes form fields for name, email, bio, and password (with password confirmation if changed). Avatar upload might be via a drag-and-drop or file picker with a preview (client-side preview is hinted by a Stimulus controller `upload-preview` in the form markup [98]). Changing the password probably requires entering the old password for security (though not shown explicitly, it's common practice). The **bio** might be shown to other users on hover or in a directory.
- **Security:** Changing email or password should re-trigger login verification (the app likely logs the user out or requires re-auth if the password is changed; not explicitly shown, but something to consider in a comprehensive design). Email uniqueness is enforced at DB level (attempting to create a user with an existing email triggers RecordNotUnique and redirects to login instead [99]).

- **PWA Support:** The system is configured as a Progressive Web App. There is a Web App Manifest (`pwa#manifest`) and a Service Worker (`pwa#service_worker`) served at known URLs [100]. This means users can **"install"** the chat as an app on their device. It also implies offline capability for caching static assets and handling push notifications through the service worker. *(UX:* The app likely prompts the user or provides instructions for adding to home screen, especially on mobile.)

Rooms & Channels (Open and Closed Rooms)

- **Explicit:** Campfire supports multiple **rooms** (group chat channels) which can be **Open** (everyone in the account has access) or **Closed** (invite-only subset of users) [23]. All users can browse and join open rooms freely. Closed rooms appear only if the user is invited (has membership).
- **Explicit:** Upon logging in, a user is directed to either a default/welcome page or their last visited room. The `WelcomeController#show` checks if the user has any room memberships; if yes, it redirects directly to the last room they visited [101] (persisting context), otherwise it shows a welcome screen (possibly prompting them to create or join a room) [101].
- **Explicit:** The **sidebar** displays the list of available rooms. This is dynamically constructed via `Users::SidebarsController#show`, which queries all visible memberships of the current user [102]. Rooms are grouped into "Direct Messages" and "Other Rooms" in the UI. The controller separates direct chat memberships from shared (group) memberships and sorts them (direct chats are sorted by most recent activity) [102] [103].
- The sidebar also shows some *placeholders* for direct messages: if there are other users with whom the current user has no direct chat yet, it lists a few as shortcuts to start new DMs [104]. This uses the `DIRECT_PLACEHOLDERS` (set to 20 max) logic to suggest people the user hasn't messaged directly [104].
- **Explicit:** Users can **create new rooms**. There are separate flows for each room type:
- **Open Room:** Any user can navigate to the "New Open Room" page (route `/rooms/opens/new`) to create a channel open to all [105] [106]. By default, a new room is given a placeholder name "New room" [107] which the user can change. Upon creation (`Rooms::OpensController#create`), the room is saved and the creator is assigned as a member (plus all other users are auto-added due to open status) [106] [108]. The system broadcasts this addition so every user's UI updates to show the new room in their "Rooms" list [109]. The creator (and all users) are then taken to the new room's chat view [106].
- **Closed Room:** Any user can create an invite-only room via `/rooms/closeds/new`. On the form, the creator can select which members to include (the form likely shows a list of all users with checkboxes) [110] [40]. Submitting calls `Rooms::ClosedsController#create`, which saves the room and grants membership to the selected users (plus the creator implicitly) [110] [108]. Immediately, those users will see the new room in their sidebar (via broadcast) [111]. Users not invited will have no access (if they somehow guess the URL, they'll get "inaccessible" message [39]).
- **Converting Rooms:** After creation, a room's type can be changed by an admin or its creator. Editing an open room to closed (or vice versa) triggers logic (`force_room_type`) to **migrate the Room object type** and adjust memberships accordingly [112] [113]. If an open room is made closed, the editor can then remove some users. If a closed room is made open, the system auto-adds all users (so nobody is left out) [48]. This conversion is broadcast so that all users update their room lists appropriately (some may gain or lose the room in their sidebar) [111].
- **Explicit:** Users can **rename rooms**. In the room settings UI (for creators/admins), changing the name and saving will update the `room.name` attribute [114]. A Turbo broadcast ensures all users see the new name without a page refresh (the partial for room list entries is replaced) [115] [44]. Regular

members who are not creators cannot directly rename a room (the edit form is likely read-only or hidden for them).

- **Explicit:** Users can **leave or hide a room** that they have access to. Instead of deleting a membership outright (except for closed rooms where an admin must remove them), Campfire uses an *involvement* setting:
- Each membership has an **involvement level**: `"everything"`, `"mentions"`, `"nothing"`, or `"invisible"` [116] [117]. Users can set this via the room's "Notification Settings" or "Leave Room" option.
- If a user chooses to leave/hide the room, the app likely sets their involvement to **invisible** for that room. This means the membership still exists but is marked hidden: the room will be removed from their sidebar (the `visible` scope excludes invisible memberships [118]). This is done in `Rooms::InvolvementsController#update` by setting involvement to "invisible" [119] [120]. A Turbo stream broadcast removes that room from the user's sidebar UI immediately [121] [120].
- If a previously hidden room is set to visible again (user re-joins via some UI), involvement would change from invisible back to another setting, and the room is prepended back into their sidebar [122] [120].
- **Explicit:** Users can adjust **notification preferences** per room using involvement settings:
- **Everything:** receive notifications for all messages.
- **Mentions:** only get notified (badge or push) when @mentioned.
- **Nothing (Muted):** no notifications, but room remains in sidebar for manual checking.
- **Invisible:** room hidden entirely (no notifications, effectively left).
  These options are presented in the room settings (e.g. "Notify me: All messages / Mentions only / Nothing" and a separate "Leave Room" or "Hide Room" action corresponding to invisible). Updating this triggers the involvement `update` action to save the new preference [119]. The backend uses the `Membership.involvement` enum to store this choice [116].
- **Inferred UX:** The main chat interface shows a list of rooms on the left and the selected room's messages on the right. Unread rooms might be bolded or show an unread count. The user can click a room to load it. If it's their first time in that room, or they are returning after some time, the last few hundred messages load by default (the `RoomsController#show` action fetches the latest page of messages) [123] [124]. For long histories, the user can scroll up and older messages will load automatically (via incremental pagination: `MessagesController#index` can fetch messages before a given message ID) [125] [126]. This provides an infinite scroll UX without overwhelming the browser (performance requirement: load messages in chunks, not all at once).
- **Inferred UX:** An open room likely has an indicator or toggle for "Make room private" (for creators/admins) and vice versa. A closed room shows the list of members somewhere (maybe in the room header or a side panel) with an option to invite more people if you created it. Possibly a "+" button or an *autocomplete user add* field is provided – the presence of an `Autocompletable::UsersController#index` that filters users by name suggests an invite UI where typing a name suggests users to add [127] [128].
- **Performance:** The app uses **Turbo Streams** (WebSockets via ActionCable) to propagate room changes instantly. When a new room is created or membership changes, the server broadcasts partial HTML updates to only the affected users rather than reloading the whole page [111] [129]. This real-time approach ensures a smooth UX even as the state changes. For example, if an admin adds you to a closed room, you'll see it appear live in your list.
- **Security:** Only members of a closed room can retrieve its data. The `RoomsController#set_room` only finds the room if `Current.user.rooms.find_by(id: ...)` returns it [38]. This prevents URL tampering (you cannot access a room you're not a member of). Even for open rooms, you must

be logged in; unauthenticated access is denied globally unless explicitly allowed for certain pages [130].

- **Security:** Room names are plain text and likely sanitized for HTML on output. The code uses standard Rails escaping in views (since no custom HTML appears in room names). This avoids any XSS in room titles or user-provided content.

Direct Messages (1:1 and Group DMs)

- **Explicit:** Users can send **Direct Messages** to one or more individuals. The interface might show a "+ Direct Message" button or list users not yet chatted with. When a user selects a person to message, the system uses or creates a **Direct room** that is private to those two users. `Rooms::DirectsController#create` calls `Rooms::Direct.find_or_create_for(selected_users)` [131] [132]. This will check if a DM room already exists for that exact set of users; if yes, it returns it, otherwise it creates a new direct room [133]. This ensures continuity of conversation (the same DM thread is reused).
- **Explicit:** Direct rooms are a special type: `Rooms::Direct`. They default to involvement "everything" for participants (meaning DMs are always considered high priority) [134]. All members of a direct room are treated as room administrators implicitly [135]. In practice, the controller overrides any admin check so that **any participant** can perform normally restricted actions in that DM (like renaming or even deleting the DM thread) [135]. *(Functional:* This is because a direct chat often doesn't have a single "owner" – all parties are equals.)
- **Explicit:** When a direct room is created, the system broadcasts it to the involved users' sidebars under the "Direct Messages" section [136]. The room might be labeled with the other person's name(s) rather than a custom title (often DMs are named by participants). In the partial `users/sidebars/rooms/direct`, likely the avatars or initials of the participants are shown [137].
- **Explicit:** If a user starts a DM with multiple people (i.e., a group DM), `find_or_create_for` will attempt to match the set. This means group DMs are possible (e.g. a direct room with 3 users). The lookup for an existing direct room uses a set comparison of user IDs [138], so if exactly the same 3 users have an old thread, it will reuse it. If not, a new one is created. *(Performance:* The code notes a potential scaling issue when there are 10k+ direct rooms, as the current lookup is not indexed and iterates all direct rooms [138]. This is a known trade-off; for moderate usage it's fine, but for very large user counts optimizations would be needed.)
- **Explicit:** In a direct room, all members can **delete the conversation** (which deletes the room for everyone). If any participant triggers `RoomsController#destroy` on a DM, the check `ensure_can_administer` will always return true (since overridden) [135]. The room and its messages would be removed for both parties [33]. (In practice, users might not commonly delete entire DM threads – a more user-friendly approach might be to *hide* the DM from your list without deleting it for the other, but currently the code's design allows actual deletion.)
- **Inferred UX:** Direct messages likely appear in the sidebar separately. Each direct conversation is listed by the name of the other user(s). If two people: it shows the other person's name; if a group DM, it might show a combined name or "[User1], [User2]...". New DMs could be initiated via an "+" button that opens a user autocomplete. Indeed, the presence of an autocomplete controller for users [127] suggests that when you start a direct message, you type a name to select the recipient.
- **Inferred UX:** When viewing a direct chat, the top bar might show the participants and an option to add others (for group DM) or leave the conversation. However, since any user can delete the DM, there might also be a "Delete Conversation" option. A safer approach would be "Archive Conversation" which for this app might correspond to setting involvement to invisible (hiding it). It's likely the UI for DMs includes a "Hide" or "Archive" action that simply removes it from your sidebar

(setting your membership invisible) instead of outright deletion. The code's allowance of delete by any participant is there, but the UX may choose not to expose a destructive delete easily.
- **Notifications:** DMs by default notify on every message (involvement "everything" ensures they are marked unread until read, and push notifications are sent if offline). This is logical as direct messages are typically high-priority.
- **Security:** Direct rooms are accessible only to their participants. Even an admin who is not a participant cannot see the messages unless they add themselves via backend or DB (there's no UI to join someone else's DM, and the membership lookup would prevent access).
- **Performance:** Since direct rooms often involve fewer users, broadcasting new messages or room events in DMs is cheap (only 2 users for one-to-one). The code uses a simple loop to broadcast a new DM room to each member's stream [136]. This is near-instant for a handful of users.

Messaging Features (Chat Messages, Attachments, Mentions)

- **Explicit:** Users can send **messages** containing text, file attachments, and special content. When a user types a message in the chat input and hits send, it triggers `MessagesController#create` via an AJAX (Turbo) request. The controller ensures the user is posting in a room they have access to (it uses `set_room` which finds the room from the URL and checks membership) [38]. Then it creates a new `Message` record in that room with the submitted content [139].
- The message body supports rich text (the model uses ActionText, so users can format text or paste images, etc.) [140]. The controller permits a `body` (text content), an `attachment` (single file upload), and a client-side `client_message_id` [141]. The `client_message_id` is a UUID generated on the client to help identify the message before the server responds (used to reconcile optimistic UI updates).
- If an attachment is included, the app uses a `create_with_attachment!` method to attach the file and save the message [139]. Images and other files are stored via ActiveStorage (with variants for thumbnails, e.g., images are automatically resized to a max of 1200x800 for previews [142] [143]).
- **Explicit (Real-time):** Upon saving a new message, the server **broadcasts** it to all subscribed clients in that room *before* sending the HTTP response. Specifically, `@message.broadcast_create` is called [144], which via Turbo Streams sends the new message HTML to every user in the room, appending it to their message list instantly. This ensures near real-time chat behavior (no manual refresh needed).
- **Explicit:** If the room is not found or the user lacks access (e.g., URL tampering), the create action rescues with a "room_not_found" response [145], preventing any message injection into unauthorized rooms.
- **Explicit:** Users can **edit** or **delete** messages they have sent. The UI likely shows an edit icon and a delete option on your own messages (and for admins, on any message):
- Editing triggers `MessagesController#update`. This action is restricted: it calls `ensure_can_administer` to allow only the message's creator or an admin to edit [146] [147]. The update saves the new content and then issues a Turbo broadcast to replace that message's HTML for all users [148]. The broadcast carries an attribute `maintain_scroll: true` [148] indicating the UI should not jump scroll position when the message updates (UX polish so that an edit doesn't jerk the viewport).
- Deleting triggers `MessagesController#destroy`, also gated by the same permission check [146] [149]. On destroy, the message is removed from the database and a Turbo stream `broadcast_remove` is sent to all clients to remove that message from the DOM [150]. This happens

in real-time; others see the message disappear (possibly replaced by a “Message deleted” notice, though the current implementation simply removes it entirely).

- **Explicit:** The chat supports **file attachments** with inline previews. When an image or media file is attached to a message, the message content type is marked as "attachment" [151] . The UI will show a thumbnail or a download link. For images, the server generates a WebP or PNG variant for efficient display [6] [152] . Video/audio attachments might display with a player (ffmpeg is installed on the server for possible transcoding or preview generation [153]). Attachments are stored in `ActiveStorage` (which can use local disk or cloud storage as configured [154]) and served via secure URLs. Users downloading or viewing attachments hit `Accounts::LogosController#show` for logos or `Users::AvatarsController#show` for avatars, and presumably a similar streaming for message attachments (likely handled by Rails ActiveStorage routes).
- **Explicit:** Users can mention others with **@mentions**. The rich-text editor likely provides an autocomplete for `@Name`. Under the hood, the `User::Mentionable` concern integrates with ActionText attachments for mentions [155] . When you mention someone, it’s stored as an attachment in the message’s content (so that it can be rendered with a special highlight and link). The plain text representation of a mention is “@Name” [156] . Mentioning triggers two things:
- In the UI, the mentioned user’s name appears highlighted and is possibly clickable (leading to their profile).
- In the backend, after the message is saved, the system can detect *mentionees*. The `Message` model or controller identifies which users were mentioned in the message.
- **Explicit (Bots & Mentions):** If any mentioned user is a **bot**, the system will send that message to the bot’s webhook. `MessagesController#create` calls `deliver_webhooks_to_bots` after saving a message [157] [158] . The logic finds eligible bots: if the message is in a group room, it takes all bots mentioned in the message; if it’s a direct room with a bot, it will take that bot [159] . Each bot (excluding the message creator if the bot somehow posted it) gets a webhook delivery job enqueued [158] .
- **Explicit:** The system supports **reactions/emojis on messages**, called **“Boosts”**. Users can react to a message with an emoji (like 👍 ❤ , etc.). In the UI, this might appear as a small emoji picker or a set of quick reaction buttons.
- Adding a reaction triggers `Messages::BoostsController#create` [160] . This creates a new **Boost** record associated with that message and by the current user (the code ensures one user’s boost can be found via `Current.user.boosts.find` in deletion) [161] . The boost content is the emoji character [162] .
- The new reaction is broadcast via Turbo Streams to all users in the room so that the emoji appears under that message [163] . The broadcast targets an element like `"boosts_message_#{client_message_id}"` – meaning each message has a Boosts section that gets updated. This broadcast uses `append` to add the new boost icon for all to see [163] .
- Users can remove their reaction (un-boost) via `Messages::BoostsController#destroy` [161] . It finds the current user’s boost by ID and deletes it, then broadcasts a removal so the UI update reflects one less reaction [164] .
- **UX:** Only one boost per user per message is allowed (the code finds by current user and boost ID, implying uniqueness). The UI might toggle the boost state or allow multiple different emojis (the `Boost.content` could be any of a predefined set [165] , but likely limited to those in `EmojiHelper::REACTIONS` for consistent UI [166]). For example, multiple users boosting the same message with 👍 will show a count.
- **Explicit: Sound effects and commands:** Campfire includes a fun feature where typing `/play` followed by a sound name triggers a predefined sound or meme. The `Message#sound` method

checks if the message text begins with `/play ___` and matches a known sound name [167]. If so, it returns a `Sound` object representing that sound (with text or an image to display) [168] [169]. The frontend likely recognizes messages of `content_type` "sound" [151] and plays the corresponding sound file (there are `.mp3` files for sounds like `56k.mp3`, `drama.mp3`, etc., included in assets [170] [171]). This is an Easter egg-like feature (from the original Campfire) enabling messages like "User plays a rimshot" with actual sound feedback.

- **Inferred UX:** A user types `/play drama` and sends it. Others see a message "User: [some text or GIF indicating drama]" and hear a brief sound clip. Only recognized keywords produce sounds (the `Sound::BUILTIN` list covers many fun phrases) [168] [172]. If an unknown name is given, probably nothing happens beyond a normal message.
- **Read Receipts / Unread Indicators:**
- **Explicit:** The system tracks read/unread status per room for each user. Each membership has an `unread_at` timestamp which is set to the time of the latest message that the user hasn't seen [173] [174]. When a new message arrives in a room, the backend marks all *disconnected* users' memberships as unread at that timestamp [173] [174]. "Disconnected" here means the user is not currently online in that room (not present or possibly not having the window open).
- **Explicit:** The app uses a **Presence** mechanism to know who's online. When a user opens a room, their membership is marked connected (the `Membership::Connectable` concern updates `connected_at` and resets `unread_at` to nil) [175] [176]. If they leave (close browser or lose connection), after a short timeout (60 seconds default) they become "disconnected" [176] [177]. Thus, if messages come in while disconnected, `unread_at` is set, flagging those messages as unseen.
- **Explicit:** A background **Heartbeat** via WebSockets likely keeps the connection alive and refreshes presence. If the browser tab is hidden or connection lost, after a grace period the membership is considered offline and new messages will be counted for unread.
- **Inferred UX:** Unread rooms are shown with a badge or bold text in the sidebar. Possibly a count of unread messages is shown (though the code sets only a timestamp, not a count). The push notification payload uses `membership.unread.count` as a badge number [178], implying the system can count how many rooms or messages are unread for a user. Likely it counts unread rooms for the badge icon. Within a room, messages newer than your last read are visually distinguished (maybe with a "New Messages" separator).
- When a user reopens the app or switches to a room, `Rooms::RefreshesController#show` can be called periodically to fetch any new messages that arrived while they were offline or the tab was inactive [179]. This returns lists of new messages and updated messages (e.g. edited) since a given timestamp [179] [180]. The front-end likely calls this when a hidden tab becomes visible to catch up instantly (ensuring no missed content).
- **Search:**
- **Explicit:** Users can search chat history via a search box (often found in the UI header or sidebar). `SearchesController#index` handles showing results for a query `q` [181]. It queries `Current.user.reachable_messages.search(q)` [182]. "Reachable messages" likely means all messages in rooms the user has access to. The search implementation (perhaps using a full-text index via ActionText or simple LIKE queries) returns matching messages.
- The results are limited to the 100 most recent matches for performance [182]. The UI will display these with some context (e.g. which room, who sent it, snippet).
- The app also keeps track of recent searches. After performing a search, `SearchesController#create` records the query in a `Search` model for that user [183]. Only the latest 10 queries are retained (older ones are trimmed) [184]. This provides a quick history of searches in the UI for reuse.

- **Inferred UX:** There might be a dedicated search page or a modal. Typing in a search box and pressing enter likely navigates to `/searches?q=term`, showing results. Recent search terms could appear as suggestions or history below the search bar. Clearing search history is possible via a "Clear Recent Searches" button, which triggers `SearchesController#clear` to delete all past queries [185].

- **Performance:**

- The chat front-end uses **incremental loading** for messages and avoids heavy operations on the main thread. For instance, large message lists might be rendered server-side or via streams rather than building huge DOM all at once. The presence of `.last_page` and `.page_before/page_after` methods [126] indicates the UI only loads a screenful of messages initially, improving load time for very active rooms.
- Attachment uploads happen asynchronously via ActiveStorage direct uploads or form submissions, so sending a large file doesn't block the app. Previews are generated in the background (libvips for images ensures efficient resizing).
- The system leverages Redis and background jobs (Resque) for tasks like sending push notifications or webhooks, so that those do not delay the HTTP response that posts a message. For example, `Room::PushMessageJob` runs async to deliver web push notifications to devices after a message is created [186] [187].
- Caching: The app sets HTTP caching headers for static content. Avatars and logos have far-future expiry with cache-busting query params (the `fresh_user_avatar` and `fresh_account_logo` routes include a timestamp version) [188] [189]. This reduces bandwidth for assets. Turbo Streams minimize re-rendering by only updating the necessary parts of the DOM.
- **Security:**
- All user input in messages is sanitized or escaped. By default, ActionText will sanitize disallowed HTML tags in the message `body`. The app further applies custom content filters (like `SanitizeTags`) to ensure no malicious script or forbidden markup goes through [74]. This prevents XSS from users posting HTML/JS.
- Attachments are served in a secure way. For example, avatar URLs are signed [190] and file downloads likely require a valid blob signed ID so that only authorized users (with access to the blob's record) can retrieve them. The presence of `ActiveStorage::Blobs` and keys suggests that.
- Mentions and other dynamic content are safely encoded. The mention system displays `@user` but behind the scenes it references the user by ID in a way that prevents injection. The mention partial `users/mention` presumably produces a span with user data, which is safe.
- The **push notifications** and **webhooks** contain limited data (only what's needed) and are protected by unique tokens (VAPID keys for push, bot tokens for webhooks). The push payload includes a badge count and message snippet, but since it goes through the browser's Push API, it's encrypted and delivered only to authorized client service workers.
- The app likely enforces permission checks on every action (we saw this in controllers). For example, even adding a reaction (`BoostsController#create`) uses `set_message` which finds the message through `Current.user.reachable_messages` [191] – meaning if you try boosting a message not in your rooms, it won't find it. This pattern is consistent and ensures users can only interact with content they have rights to.

Notifications & Presence (Real-Time Feedback)

- **Explicit: In-App Notifications (Unread)** – As described, the system marks messages as unread for offline users. Conversely, when a user **reads** a room (opens it), the client will send a signal to mark all messages as read (likely by calling an ActionCable channel or an endpoint to clear `unread_at`). The `Membership.connect` logic does set `unread_at: nil` when a user comes online to a room [175] [192], effectively clearing the unread status. So simply being connected to the room marks it read.
- **Explicit: Web Push Notifications** – The app can send push notifications to a user's device when they are not actively looking at the chat. Users must opt-in by allowing notifications in the browser. Once they do, a **Push Subscription** is created:
- The client registers a push subscription (with endpoint and cryptographic keys) via the browser's Push API. This is sent to `Users::PushSubscriptionsController#create`, which stores it in the `Push::Subscription` model [193] [194]. If the same subscription already exists (same endpoint), it just updates the timestamp (to avoid duplicates) [193].
- Users can view and manage their push subscriptions in a settings page (likely "Notifications" settings). `Users::PushSubscriptionsController#index` would list all device subscriptions the user has (e.g., "Chrome on Windows, subscribed on Jan 1", "Firefox on Android, subscribed on Feb 3"). They can revoke any by clicking "Disable" which calls `#destroy` to remove that subscription [195] [196].
- When a message arrives and a user is offline (disconnected from that room), the `Room::PushMessageJob` is enqueued [186] [197]. This job finds all push subscriptions for users who have that room **visible** and unread, excluding the sender [198] [199]. It then sends a web push notification to each. The notification includes:
    - A title (e.g., the room name or "New message in Room X").
    - A body (likely the message text or "[User]: message snippet").
    - A `path` which is the URL to click (so clicking the notification opens the specific room).
    - A badge number indicating total unread count [178].
- The push sending is handled by a `WebPush::Pool` which batches and sends asynchronously [200] [201]. Any invalid subscriptions (e.g., user unsubscribed or token expired) are detected and removed from the DB to keep things clean [202] [203].
- **Explicit: Test Notifications** – There is a feature for users to test their push setup. In the UI, likely a "Send test notification" button exists in the Notifications settings. Pressing it calls `Users::PushSubscriptions::TestNotificationsController#create` [204] for a specific subscription. This simply sends a push with a generic title "Campfire Test" and a random UUID as the body [205]. The result lets the user confirm that notifications are working on their device.
- **Explicit: Presence (Online Status)** – The app uses several ActionCable channels to reflect user presence:
- `HeartbeatChannel`: likely used to regularly inform the server that the user is still connected and which room is active. The front-end code dispatches a JavaScript event if the heartbeat channel disconnects, after a 5-second timeout marking the user offline [206].
- `PresenceChannel`: possibly tracks global online status or per-room presence. There might be a list of online users displayed in each room (though the code for showing who's currently in the room is not explicitly given, it's a common chat feature).
- Under the hood, presence is managed by the `Membership::Connectable` module. When a user opens a room, the client likely subscribes to `RoomChannel` and triggers `Membership.present` or `connected` for that membership [207] [208]. On disconnect or closing,

`Membership.disconnected` is called after a timeout [209]. This data could be broadcast via `UnreadRoomsChannel` or similar to update the counts of online/offline.

- For example, `UnreadRoomsChannel` might broadcast changes in unread counts or a simple signal for the sidebar to reload unread state. The code snippet suggests that if the channel disconnects and reconnects, it triggers a sidebar reload [210].
- **Inferred UX:** Users likely see presence as a green dot next to avatars or a list of currently online users in a room. Typing indicators are another presence-related feature: indeed, a `TypingNotificationsChannel` exists, which presumably broadcasts "User X is typing..." events to others in the room. The front-end might throttle and send a "typing" event when the user is actively typing, and display a subtle "[User] is typing..." text for others. Though code for handling typing isn't shown, the existence of that channel indicates the functionality.
- **Inferred UX:** If someone @mentions you while you are online but not currently viewing that room, you might get a browser notification or at least an audible alert. The original Campfire had an option to play a sound for mentions. This implementation could reuse the `/play sound` system to play a "bell" sound on mention (the sound list includes a `bell` with text [168]).
- **Security:** Push notifications contain minimal data and rely on the browser's encryption (VAPID). The server never sends the actual content unencrypted over push; it uses `WebPush.payload_send` with encryption under the hood (not detailed here but standard for Web Push). This ensures only the user's device can decode the message content.
- **Security:** Presence information is only shared with authenticated users. The channels likely use `Current.user` context, and there's no way for an outsider to query who is online. Within the app, presence might be considered non-sensitive, but it's kept within the membership context.

In summary, **Regular Users** can authenticate, manage their profile, participate in multiple chat rooms or direct conversations, send rich messages with files and mentions, react to messages, search archives, and receive notifications – all in real-time. The system explicitly defines these flows in the code, and we infer additional UX behavior (like how things are displayed or minor interactions) from standard practice and the code structure. The combination of functional requirements (message sending, room creation, etc.), UX requirements (instant updates, clear indicators, ease of use), performance considerations (pagination, background jobs, broadcasts), and security measures (auth checks, sanitization, encryption) provide a comprehensive picture of Campfire's capabilities and design.

Bot Integration Role

Receiving Messages via Webhooks (Bot as Listener)

- **Explicit:** Campfire provides a first-class integration for **Bots**, allowing external services to participate in chats. When a bot user is present in a room (either invited to a closed room or implicitly in an open room since all users including bots get added, or in a direct chat with someone), that bot can **listen** for certain messages:
- If the bot is directly addressed (e.g., in a DM) or mentioned in a group, the system triggers a webhook call to the bot's configured URL [158]. This is handled by a background job `Bot::WebhookJob` that calls the bot's webhook endpoint asynchronously [211] [212] to avoid slowing down the chat.
- The payload of the webhook is a JSON with details about the message and its context: the user who sent it, the room info, and the message content (both HTML and plain text versions) [213] [214]. It also

includes helpful links: e.g., `room.bot_messages_path` – an API endpoint the bot can POST to in order to reply in the same room [213] [215].

- The webhook delivery is resilient: it has a timeout of 7 seconds for the external service to respond [216] [217]. If the bot's endpoint doesn't respond in time or at all, Campfire catches the timeout and inserts a message in the chat from the bot saying it failed to respond [217] [218]. This feedback is important for UX: participants know the bot didn't act.
- Bots can be mentioned in messages by @name just like users; such mentions are what trigger the webhook in group rooms. In direct rooms with a bot, every message is effectively addressed to the bot (so likely every message triggers the webhook unless the bot itself sent it).
- **Explicit:** When the bot's webhook receives the JSON, it can decide to respond. The design allows **bot responses directly via the webhook HTTP response** for simplicity:
- If the bot's HTTP response has a content-type of text (plain or HTML) and a 200 status, Campfire interprets the response body as a **text reply**. It will create a new message in the room from the bot user with that text [219] [220]. This means a bot can just return "Sure, here's info on that..." and the users will see the bot's message appear.
- If the bot's response is some other content-type (e.g., an image or file), Campfire will treat it as an **attachment reply**. For example, if the bot returns an image (content-type image/png) in the response body, Campfire will create an ActiveStorage blob from that data and post it as a message attachment from the bot [221] [222]. This allows bots to send images or files without a separate API call.
- The webhook response handling is robust: if the bot returns text and an attachment (unlikely in one response, but if it did HTML with an image?), the code checks for text first, then attachment. Typically, bots will return one or the other.
- **Explicit:** The bot might also choose not to respond or to respond with a non-200 status for certain messages (which would result in no automatic reply posted). If a non-200 code is returned and no text, Campfire does nothing (no error unless it's a timeout or specific failure).
- **Security:** Outgoing webhooks are made to the URL set by the admin. The system prevents internal network abuse by disallowing private IP targets [73]. The bot's URL should ideally be HTTPS (the code supports both, but if not HTTPS, `http.use_ssl` will be false [223]).
- The webhook request includes only the necessary information. It does not include the bot's token or any secret; authentication of the request to the external service could be done by the bot's URL containing a token or the service whitelisting Campfire's IP.
- The bot's external service must handle the JSON and decide what to do. Any heavy lifting (e.g., calling third-party APIs or databases) is done on that external side, keeping Campfire responsive.
- **Inferred UX/Functionality:** The presence of bots could be indicated in the UI (maybe a robot icon next to their name). Users might summon bots with commands or triggers. For instance, a bot named "WeatherBot" might respond whenever someone mentions "weather" or specifically @mentions it: "@WeatherBot what's the forecast?". The bot's external logic would parse that and reply with a weather summary.
- **Performance:** The webhook calls and bot responses happen asynchronously. The user sending a message won't wait for the bot reply (they'll see their message immediately). When the bot responds (whether immediately via webhook response or later via API call), that message appears in chat. The design using the HTTP response for quick replies is actually faster and saves the bot from having to issue a separate HTTP request back.
- If a bot needs to do a longer operation, it could immediately return 202 Accepted or some short acknowledgment to avoid the timeout, and later use the API to post the result.

- The system can handle multiple bots: each bot with a webhook will get called if they are mentioned. The code ensures a bot doesn't webhook itself (excluding the message creator if that is a bot) [159] to avoid infinite loops.

Sending Messages via API (Bot as Speaker)

- **Explicit:** Bots can also send messages proactively (not just via webhook replies). To do this, a bot (external service) uses the **Campfire API** endpoint: `POST /rooms/:room_id/:bot_key/messages` [224] [225].
- The bot must know the `room_id` (which could be gleaned from a webhook payload or via configuration) and use its `bot_key` (e.g. `42-abcdef123456`). The bot key authenticates the request; it's passed in the URL and the system identifies the bot by this token [226] [227].
- Authentication: The `Authentication` concern allows requests with a `bot_key` parameter to skip the normal login (`deny_bots` is disabled in `Messages::ByBotsController#create` specifically) [228]. If the bot_key matches a valid active bot, `Current.user` is set to that bot for the scope of the request [226] [229]. All further operations run as the bot user.
- Bots post via a specialized controller `Messages::ByBotsController`, which inherits from the normal `MessagesController` but tweaks parameters handling [230]. Bots don't submit a form with multipart data in the same way as a human interface, so this controller:
  - Accepts either a raw text in the request body *or* a file attachment in a param. If an `attachment` param is present (probably as raw bytes or a file upload), it permits that [231]. If not, it reads the raw request body as the message text [232]. This flexibility means a bot can POST text directly or use an API client to send a file.
  - After creating the message (via the normal create logic in the parent), it returns an HTTP 201 Created with the `Location` header set to the URL of the created message [233]. The content of the message is broadcast to the room as usual, so users see it instantly.
- Using this API, a bot could inject messages into any room it's a member of, on its own schedule or trigger (e.g., a daily report, or responding to an external event).
- **Explicit:** A bot's message posted via API triggers the same post-create hooks as any other message: it will mark users' unread, possibly notify others, etc. One exception: such a message likely won't trigger a webhook back to the same bot (the code excludes the message's creator from bot webhooks [159], and here the creator **is** the bot, so it won't webhook itself).
- **Security:** The bot_key is essentially the API key. It must be kept secret on the external service side. If it's compromised, someone could impersonate the bot to post messages. This is why admins can reset bot keys. All bot API actions are restricted to what a normal user can do in that context, with the identity of the bot:
- A bot cannot create or destroy rooms via API (there are no such endpoints with bot access).
- The `deny_bots` filter prevents bots from accessing web UI controllers; only the specifically allowed `messages/by_bots#create` is open to them [228]. So a bot key cannot be used to, say, fetch messages or search (there's no implemented GET API for messages for bots).
- Rate limiting for bot messages isn't explicitly shown, but it might rely on external control. If needed, the admin could implement a simple throttle on the bot's side or the API can be extended with a rate limiter.
- **Inferred Use Cases:** Bots can be information providers, bridges to other services, or simple fun responders. For example:
- A **SupportBot** could listen in a room and whenever someone says "help me with X", the webhook could create a support ticket and respond "I've created ticket #123".

- A **NotificationBot** could not use webhooks at all, but instead be fed from an external system: e.g., a CI server posts build status into a Campfire room via the API using the bot's key.
- Because bots can also be invited to closed rooms, teams can have private interactions with a bot.
- **Performance:** Bot messages use the same broadcast mechanism, so they won't overwhelm the system beyond normal messages. The main overhead is webhook calls. The code handles webhook sequentially per message (iterating through eligible bots) [158] – if many bots are mentioned, they each get a call. This is typically fine (few bots per room).
- The WebPush and Webhook deliveries run in background threads (via a thread pool `WebPush::Pool` and the job queue for webhooks), so they don't block the main app thread. This ensures the chat stays responsive even if a bot's endpoint is slow.
- **UX for Bots:** Regular users will see bots as participants in the room. Bots have an identifier (maybe "[Bot]" label) and can have avatars (perhaps a robot icon if none set). Their messages appear just like a user's messages. If a bot posts an attachment (image/file), it shows up with the bot's name.
- If a bot fails to respond or errors out, the inserted "Failed to respond" message lets users know the bot didn't complete the request, which is a good UX detail to avoid confusion.

- Users might not need to know the technical detail, but they should know how to invoke bots (usually by mention or command).

- **Security (Additional):** The system ensures bot accounts cannot be used interactively by humans. There's no login form for bots (they don't have passwords). If a bot tried to access the normal UI with its bot_key, it's forbidden by the `deny_bots` filter on all controllers except the allowed one [130] [234]. This prevents misuse of bot credentials to spy on chat via web endpoints. Essentially, bots are write-only (through the API) and read via webhooks; they do not have a web session or UI presence beyond appearing in the chat.

In conclusion, **Bot integrations** allow extending Campfire's functionality by reacting to messages and posting automated responses or notifications. The code explicitly defines how bots are created, authenticated, and triggered (webhook calls on mention, API endpoint for posting). The system balances flexibility (bots can post anything users can, including attachments) with security (isolated token auth, limited access) and provides a straightforward UX (mention a bot to get a response, see bot messages inline).

Table: Summary of Roles, Features, and Requirements

Role	Feature	Key Requirements & Behaviors
Administrator	Account Setup	First user creates account (first-run) [1]; can edit account name/ logo [3]; configure custom CSS [10]; share/regenerate join invite link [17] [16].
Administrator	User Management	View user list [26]; promote/demote roles [28]; deactivate users [29]; ensure only admins can do these [30].
Administrator	Room Moderation	Can delete any room [33]; change room open/closed access for any room [235] [112]; add/remove members in closed rooms [41]; delete others' messages (via admin role) [236].

Role	Feature	Key Requirements & Behaviors
Administrator	Bot Management	Create bots with name, avatar, webhook [50] ; obtain bot API key [55] ; reset bot keys [62] ; deactivate bots [65] .
Regular User	Authentication	Log in with email/password [76] (rate-limited [75]); stay logged in via secure cookie [68] ; log out (clear session, optionally unregister push) [81] ; transfer session via QR code [82] .
Regular User	Profile & Settings	Edit profile (name, avatar, bio, email, password) [90] ; remove avatar [93] ; browser compatibility checks [96] ; install PWA (manifest/service worker provided) [100] ; manage notification subscriptions [193] [196] .
Regular User	Rooms – Open/ Closed	Browse open rooms (auto-joined) [23] ; create open rooms (all-users access) [106] ; create closed rooms (invite specific users) [108] ; invite or remove members if room owner [41] ; rename rooms [237] ; leave/hide rooms (set involvement invisible) [122] [120] .
Regular User	Direct Messaging	Start 1:1 or group DM (system finds or creates room) [133] ; reuse existing threads for same users [138] ; all participants equal (can administer DM) [135] ; hide or delete DM thread (either by involvement or full delete) [121] [135] .
Regular User	Sending Messages	Post messages with text and/or file attachments [139] ; attachments stored & previewed (images/videos via ActiveStorage) [151] [142] ; messages broadcast in real-time to others [144] ; handle errors (e.g., no permission -> alert) [238] .
Regular User	Editing/ Deleting Msgs	Edit own messages (or any if admin) [146] [148] ; broadcast edited content to update UI [148] ; delete own messages (or admin any) [147] [150] ; broadcast removal to all clients [150] .
Regular User	Reactions (Boosts)	React to messages with emoji (👍, ❤, etc.) [239] ; one boost per user/message; new boost saved and broadcast to update reaction count [163] ; remove boost (un-react) possible [161] [164] .
Regular User	Mentions	Mention users with @ (auto-complete user list) [127] ; mentioned user's name highlighted in chat; if bot mentioned, triggers webhook [158] ; mention may override "mentions-only" notification preference to notify that user.
Regular User	Search	Search messages across accessible rooms [182] (full-text); show up to 100 results [182] ; store recent searches and allow clearing history [184] [185] for convenience.
Regular User	Notifications (Push)	Opt-in to browser push notifications; store subscription (endpoint & keys) [193] ; on new message, send push to offline users in that room [199] with message snippet and unread count badge [178] ; allow user to test notification [205] and remove subscriptions [195] .

Role	Feature	Key Requirements & Behaviors
Regular User	Notifications (In-App)	See unread rooms highlighted (memberships with `unread_at` set) [173] ; unread count badge possibly shown (e.g., on favicon or app icon) [178] ; real-time updates of unread when others send messages; read on entering room (clears unread_at via presence) [175] [240] .
Regular User	Presence & Typing	See who's online (maybe avatar indicators or list); presence updated via heartbeat (5s offline delay) [206] [241] ; typing indicator shown when others are typing (via TypingNotificationsChannel); own status goes offline shortly after disconnect (60s) [176] [207] .
Bot	Webhook Triggers	Receive webhook POST when mentioned or DM'd [158] ; JSON payload includes user, room, message details [213] ; respond within 7s to auto-post reply [242] [220] (text or attachment); timeout or error yields a failure message in chat [217] .
Bot	Posting Messages	Post messages via API using `bot_key` (authentication token) [55] ; endpoint `POST /rooms/:id/<bot_key>/messages` available [224] ; can send text (in request body) or file (as attachment param) [231] ; results in a normal message from bot broadcast to room [233] .
Bot	Permissions & Limits	Bot cannot log in via UI (no password, UI controllers forbid bots) [130] [234] ; bot actions limited to sending messages (no direct control of users/rooms); bot's messages and attachments are subject to the same sanitization and file type rules as others.
Bot	Management	Created and configured by admins (see Admin Bot Management); each bot identified by name and avatar in chats; bots can be deactivated to revoke access (no messages or webhooks after deactivation).

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 basecamp-once-campfire-8a5edab282632443.txt

file://file-AmJjTy57dBPho4hsZ3Wwzb