ChatGPT

End-to-End Onboarding Flow with QuickBooks Integration and AI Chat PWA

Goal

Enable new business owners to: - Register their business in a multi-tenant system
- Complete payment and plan selection (subscription via Stripe)
- Proceed through an onboarding wizard
- Connect their QuickBooks account **or** upload contacts via CSV
- Use an AI assistant via a mobile-friendly Progressive Web App (PWA)

Tasks and Subtasks

Registration and Account Model

- [ ] **Multi-tenant Data Models:** Ensure the data models support multi-tenancy. Set up `User`, `Business`, and a join model like `BusinessUser` (with roles) so a user can belong to multiple businesses and each business can have multiple users.
- [ ] **Business Entity on Sign-Up:** When a new user registers, create a corresponding `Business` record for their organization and associate the registering user as the owner/admin of that business.
- [ ] **Track Active Business:** Store the current `activeBusinessId` in the user's session or profile (e.g., so that API requests and UI components know which business context to use). Enforce that most operations require an active business to be selected (preventing context confusion if user belongs to multiple businesses).
- [ ] **Route Guards:** Implement frontend and backend route guards to enforce authentication, business membership, and subscription status. For example, protected routes should require a logged-in user *and* an active subscription for the active business, otherwise redirect to login or payment pages as appropriate.
- [ ] **Post-Registration Redirect:** After a successful registration, automatically redirect the user to the plan selection/billing page (e.g., navigate to `/billing/choose-plan`) to encourage setting up their subscription immediately.

Payment Portal and Subscription Setup

- [ ] **Plan Model & Listing:** Create a `Plan` model in the database to define available subscription plans (fields might include `stripePriceId`, name, billing interval, price, and feature flags). Seed the DB with the plans offered (e.g., Basic, Pro) so that the app can display plan options dynamically instead of hard-coding them.
- [ ] **Plan Selection Page:** Build a UI at `/billing/choose-plan` that fetches the plan list from the database and lets the user choose a plan. Display plan details (price, interval, key features) and a “Select” or “Upgrade” button for each.

- [ ] **Stripe Checkout Session:** Implement a backend endpoint `POST /api/billing/create-checkout-session` that creates a Stripe Checkout Session for the chosen plan. This should use Stripe's API to generate a session URL for the customer's email and selected `priceId`. Return the session URL so the frontend can redirect the user to Stripe's hosted checkout.
- [ ] **Stripe Webhook Endpoint:** Set up a secure webhook handler (e.g., `POST /api/webhooks/stripe`) to listen for Stripe events related to checkout and subscription lifecycle [1]. In particular:
- [ ] On a successful checkout completion (Stripe event `checkout.session.completed`), retrieve the subscription details and mark the user's subscription as **active** in our system (e.g., update a `Subscription` record or the Business status). This should include storing the Stripe customer ID and subscription ID, the plan info, status = active, and the current period end date [2] [1].
- [ ] On subscription payment failure or cancellation (e.g., `invoice.payment_failed` or a subscription updated event indicating status = past_due), update the subscription status in our database to **past_due** or **canceled**. Implement logic to gracefully restrict paid-only features if a subscription lapses. (Stripe will retry payments, but if ultimately failed, the app should limit functionality until resolved.)
- [ ] **Subscription Data Storage:** Record subscription metadata in the database. This includes the Stripe customer ID, Stripe subscription ID, current status (active/trial/past_due/canceled), start date, current period end, and plan details [2]. This will allow the app to quickly check if a given business (or user) has an active subscription and when it expires.
- [ ] **Post-Payment Onboarding Redirect:** After the Stripe checkout is completed successfully (either by listening for the webhook or via the return URL), forward the user into the onboarding flow. For example, redirect them to `/onboarding/start` to begin setting up their business profile and data now that their subscription is active.

Onboarding Wizard

- [ ] **Multi-Step Onboarding UI:** Develop an `OnboardingLayout` or wizard component that guides the user through several steps. This should be a multi-step form with progress indication (e.g., a stepper or progress bar). The steps include:
- **Step 1: Business Profile –** Form to collect business details such as business name, operating hours, time zone, address, etc. This populates the new Business's profile.
- **Step 2: Data Connection –** Let the user choose how to import their customer/contact data. Provide two options: **Connect QuickBooks** (OAuth flow to import customers) or **Upload CSV** (manual contact import). This step might branch: if they choose QuickBooks, show a "Connect QuickBooks" button; if CSV, show an upload interface (or skip to that in the next step).
- **Step 3: AI Assistant Setup –** Collect any info needed to personalize the AI assistant. For example, ask for an assistant name or nickname, and possibly set toggle options for what the assistant can do (permissions or scope, e.g., "Allow AI to access contact list"). This could just be informational or a few preference settings.
- **Step 4: Completion –** A confirmation screen that the onboarding is complete. This can include a summary of what was set up and a call-to-action to go to the Dashboard. Possibly also remind the user if there are any unfinished integrations (e.g., if they skipped QuickBooks connection or skipped the contact upload, prompt them to do it later).
- [ ] **Persist Onboarding Progress:** As the user moves through the onboarding steps, save their progress in case they drop off and return later. This could be a field on the User or Business (like `onboardingStepCompleted`) or a separate onboarding status table. The app should redirect a

user who hasn't finished onboarding back into the wizard at the right step if they sign in again. Once Step 4 is done, mark the onboarding as fully complete for that user/business.

QuickBooks Integration

- [ ] **Secure Credentials Storage:** Add configuration for QuickBooks Online API credentials. Store the Intuit **Client ID**, **Client Secret**, and any other required keys in environment variables or a secure config store (do **not** hard-code them). This ensures we can safely call the QuickBooks APIs for OAuth2 [3].
- [ ] **OAuth Authorization Flow:** Create an endpoint (e.g., `GET /api/integrations/qbo/authorize`) that begins the OAuth 2.0 flow with QuickBooks. This endpoint should construct the Intuit authorization URL (including client_id, redirect_uri, requested scopes, and state) and redirect the user's browser to Intuit's consent page [4] [5]. The user will log in to QuickBooks and approve access for our app.
- [ ] **OAuth Callback Endpoint:** Implement a callback route (e.g., `GET /api/integrations/qbo/callback`) which Intuit will redirect to after the user authorizes the app. This route will receive a temporary authorization `code` (and a `realmId` for the QuickBooks company). Use this code to request OAuth tokens from Intuit (exchange the auth code for an **access token** and **refresh token**) [6]. Store these tokens securely in the database, associated with the Business or integration record [6]. Also save the QuickBooks `realmId` (the company ID) since it's needed for API calls.
- [ ] **Token Refresh Logic:** Implement logic to refresh the QuickBooks access token using the refresh token when needed. QuickBooks access tokens expire after a short period, but the refresh token is long-lived. Before making QBO API calls (or via a scheduled job), check token expiry and use the refresh token to get a new access token if necessary [7]. Update the stored tokens accordingly. If the refresh token has expired or becomes invalid, flag the integration so that the user may need to reconnect (OAuth flow again) [7].
- [ ] **Customer Data Sync Job:** Once connected, kick off a background job to import contacts from QuickBooks. Use the QuickBooks API (with the stored tokens) to fetch the list of customers from the QuickBooks Online company. Map these to the app's internal `Contact` model and save them in the database. This might involve pulling customer name, email, phone, etc. **Deduplicate** or merge if some contacts already exist (to avoid duplicates if the user runs the sync twice).
- [ ] **Sync Status & UI Feedback:** In the dashboard (or during onboarding Step 2 completion), show the status of the QuickBooks sync. For example, display a message like "Importing X contacts from QuickBooks..." and update it when done ("Imported 45 contacts from QuickBooks"). Also indicate the integration status (connected or not). If the QuickBooks auth fails or tokens expire, surface that in the UI so the user can take action (reconnect).

Manual Contact Upload

- [ ] **CSV Upload Option:** If the user chooses to import contacts via CSV instead of QuickBooks, provide a file upload interface in the onboarding wizard (Step 2 for CSV path). Allow them to select a CSV file containing their contacts.
- [ ] **Template and Instructions:** Offer a downloadable CSV template with the required columns (e.g., Name, Email, Phone, etc.) so the user can prepare their contacts in the correct format. Also, show a brief description of the expected format and size limits (to prevent user frustration).
- [ ] **Backend Upload Endpoint:** Implement an API route (e.g., `POST /api/contacts/import`) that accepts the uploaded CSV file (multipart/form-data). Upon receiving a file, the server should parse the CSV asynchronously. Rather than blocking the request, enqueue a **background job** to handle the

import (this allows processing large files without timing out and gives the user immediate feedback that import has started).

- [ ] **Background Import Job:** In the background worker, read and parse the CSV file. Validate the data for each contact (e.g., proper email format, required fields present). Deduplicate entries (both within the file and against existing contacts for that business) to avoid creating duplicates. Insert the new contacts into the `Contact` model/table. If there are errors for certain rows (e.g., missing required info), collect those to report back.
- [ ] **Import Completion UI:** Once the job completes, update the UI with a success message. If the user is still on the onboarding completion step, show the number of contacts successfully imported (and perhaps list any errors or skipped rows). If the user has left the onboarding flow, the dashboard should surface an alert or notification about the import results (e.g., "Import finished: 50 contacts added."). Logging an event like `contacts_imported` could also trigger an email or alert if needed.

AI Chat Assistant

- [ ] **Chat API Endpoint:** Create an API endpoint (e.g., `POST /api/chat`) that the client app (web or PWA) can call to send a message to the AI assistant and get a response. This endpoint should verify the user's authentication and which business they're querying for (use `activeBusinessId`). For responsiveness, consider implementing this as a streaming endpoint (using server-sent events or web sockets) so that the AI's response can be streamed back gradually.
- [ ] **Business Context in AI Prompt:** Integrate the business's data into the AI assistant's context. When a user sends a question or request, the server should assemble relevant info about the business to include in the prompt for the AI. For example, include the business name, operating hours, next appointment from the schedule, or a summary of the contact list if relevant. This ensures the AI's answers are tailored to the specific business. (For privacy and cost, be mindful to include only necessary data in prompts, perhaps summaries or counts rather than full raw data.)
- [ ] **Background Data Sync for AI:** To support the above, make sure the latest business data is readily available to the AI service. This could mean periodically caching key data (like an updated list of upcoming appointments or recent contacts) so that when the AI endpoint is called, it can quickly retrieve these without heavy DB queries. If data changes frequently, possibly maintain an in-memory cache or use a fast query to gather context.
- [ ] **Chat Session & Logging:** Implement a logging mechanism for the AI chat. Each chat message sent by the user (and the reply from the AI) should be saved to a database or logging system. This could be used to display past conversation history and is also important for monitoring usage and troubleshooting (and could help train future improvements). Log events like `chat_message_sent` with metadata (user, business, timestamp, maybe message length) for analytics. Ensure sensitive data is handled appropriately in logs.
- [ ] **Prompt Orchestration Design:** Structure the code so that the prompt generation and AI response handling are modular. For instance, have a dedicated function or class that builds the AI prompt from business data and user query, and another that calls the AI API (OpenAI or other) and streams/parses the response. This separation makes it easier to adjust the prompt or switch AI models later without altering the high-level logic. Also consider using a system message or few-shot examples to guide the AI's behavior as an assistant for this domain.

Owner Dashboard (Mobile + Desktop)

- [ ] **Dashboard Overview Page:** Develop a dashboard for business owners that is shown after onboarding (route like `/dashboard`). This page should summarize key information for the user's

business: for example, **upcoming schedule** (next few appointments or tasks), recent metrics (like number of bookings this week, total revenue if applicable), recent AI chat interactions or tips, and the status of their integrations (e.g., "QuickBooks: Connected" or "Contacts: 50 imported"). Each of these can be in a separate section/card for clarity.

- [ ] **Responsive Layout:** Use a mobile-first design for the dashboard and all new pages. On small screens (mobile devices ~320px wide), stack content in a single column of cards/sections that can be scrolled. Ensure the design scales up nicely for larger tablets or desktop by introducing responsive grid or two-column layouts at larger breakpoints. The UI should never require horizontal scrolling at 320px width; all content and tables should either reflow or be scrollable vertically if needed.
- [ ] **Mobile Navigation:** Implement an easy navigation for mobile users. Use a bottom navigation bar (since this is a PWA aimed at mobile usage) with 4 primary sections: Dashboard (home), Chat (AI assistant), Schedule (if scheduling feature exists), and Settings (or Account). Each icon/button in the bottom nav should be large enough to tap easily – aim for at least ~44px by 44px touch target for each (which aligns with accessibility guidelines for touch interfaces [8]). Highlight the current page's icon in the nav for clarity.
- [ ] **Desktop Considerations:** On desktop web, the bottom nav can be less prominent or replaced with a top nav or sidebar, but it should adapt accordingly. The dashboard page on desktop can show more info at a glance (e.g., a multi-column layout). Ensure all interactive elements are usable with mouse and touch, and use responsive components (tables, charts, etc.) that adapt or have scrollable containers on small screens rather than breaking layout.

PWA Support for AI Assistant

- [ ] **Web App Manifest:** Create an `app.webmanifest` file to enable installable PWA features. Include at minimum: the app name, short name, icons in required sizes, theme color, background color, and specify `start_url` as the chat page (e.g., `"/chat"` or whichever route loads the AI assistant). Set `display` to `standalone` so that when installed, it looks like a native app without browser UI.
- [ ] **Service Worker & Offline Mode:** Register a service worker in the app. The service worker should cache the essential assets (HTML, CSS, JS, icons) for offline use [9]. Implement a suitable caching strategy (e.g., Network-first for dynamic data, Cache-first for static assets) so that the app can load even with no connectivity. Also utilize the **Background Sync API** if possible – if the user tries to send a chat message offline, the service worker can queue the request and send it once connectivity is restored [10] [11]. This ensures the AI assistant feels responsive even with spotty internet (the user's message will eventually go through when online).
- [ ] **Full-Screen Chat UI:** Optimize the AI chat interface for mobile/PWA usage. When the PWA is launched (starting at the `/chat` route), it should open into a full-screen chat experience. Design the chat UI with a conversation view (messages list), a composer input at the bottom, and maybe quick action buttons (like suggested questions or common commands). Ensure that the layout works in both portrait and landscape on mobile, and that the text input and send button are easily tappable. Handle the on-screen keyboard properly (e.g., input stays above keyboard).
- [ ] **Reuse Auth Session:** Make sure the PWA does not require a separate login. It should share authentication with the web app (if using cookies or localStorage tokens). This might require that the auth token is stored in a way the service worker and PWA pages can access, or simply that when the user installs the PWA they are already logged in via the browser. Test that scenario: a user who is logged in on the mobile site and then "Add to Home Screen" can open the PWA and still be logged in to use chat immediately.

- [ ] **PWA Compliance Check:** Test the application with Lighthouse or similar PWA audit tools. Ensure it meets baseline PWA installability criteria: it must be served over HTTPS, have a valid manifest and service worker, respond with a 200 when offline (for the start URL), and use an icon of proper size. Aim to pass all relevant Lighthouse PWA checks (e.g., "PWA is installable", "offline support", etc.). Additionally, verify things like the splash screen and icon when launching the installed app on a device.

Testing and Monitoring

- [ ] **Unit Tests:** Write unit tests for critical pieces of this flow. This includes testing the Stripe webhook handler logic (simulate events for checkout success and failure to ensure the subscription status updates correctly), the QuickBooks OAuth callback logic (given a mock auth code, does it exchange for tokens and store them?), and the CSV import parsing (test that various CSV inputs result in correct contact records or proper error handling). Also, test smaller units like the service worker's message queuing logic (if possible in a controlled environment).
- [ ] **Integration Tests:** Develop integration tests that cover multi-step interactions between components. For example, using a testing framework, simulate a user signing up via the API, ensure a Business is created and `activeBusinessId` is set, then simulate hitting the billing endpoint and a webhook call from Stripe, and finally check that the onboarding steps can be retrieved/continued. These tests ensure that the backend pieces work together as expected (from authentication to subscription activation to data import).
- [ ] **End-to-End (E2E) Tests:** Use an end-to-end testing tool (like Cypress) to run through the entire user journey in a browser context. Create tests for both desktop and mobile viewports. Steps to automate: user opens the app, registers a new account, is redirected to choose a plan, completes a dummy payment (you may use Stripe test mode and stub webhooks), goes through onboarding steps, connects QuickBooks (this part might be tricky in automated tests; alternatively, skip QBO and do CSV upload in the test), and finally sends a message in the chat. Validate through the UI that each step behaves correctly (e.g., after payment, the onboarding starts; after onboarding, the dashboard is shown; the chat returns an AI message). These E2E tests will catch any integration issues in the full flow.
- [ ] **Event Logging:** Instrument the application with logging for key events. For auditing and analytics, log events such as `user_registered` (with user and business ID), `subscription_active` (with plan info), `qbo_connected` (when QuickBooks integration is successful), `contacts_imported` (with counts of records), and `chat_message_sent` (when the user sends a question to the AI). These logs can help in debugging issues (e.g., if a user claims they did X but something failed, we can trace it) and also in understanding user engagement with features.
- [ ] **Monitoring & Alerts:** Set up monitoring for the critical background processes and webhooks. For example, integrate with a monitoring service or at least add error tracking – if the Stripe webhook fails (non-200 response or exception), or the QuickBooks sync job throws an error, those should trigger alerts (email or Slack notifications to the dev team). Likewise, monitor the AI chat service for failures or high latency. Define alerts for conditions like "Webhook failed to process after X retries" or "Contact import job failed" so that the team can respond quickly to any issues in the user onboarding flow.

Each of these tasks should be tracked with a checkbox above. As development progresses, update and tick off each item. This will ensure nothing is missed in delivering the full end-to-end experience for new business owners. Once all subtasks are completed and tested, new users should be able to seamlessly go

from sign-up to a fully onboarded state with an active subscription, their data loaded (from QuickBooks or CSV), and access to their AI assistant on any device.

---

[1] [2] plan.md
https://github.com/jamesjmcconnell/PodPace/blob/c65f173f1a5bd0b54a56db8e2a8c0789dc2cdb87/plan.md

[3] [6] [7] developer.intuit.com
https://developer.intuit.com/app/developer/qbo/docs/develop/authentication-and-authorization/oauth-2.0

[4] [5] Step-by-Step Guide to Integrating QuickBooks Online with NodeJS
https://glinteco.com/en/post/step-by-step-guide-to-integrating-quickbooks-online-with-nodejs/

[8] Accessible Target Sizes Cheatsheet — Smashing Magazine
https://www.smashingmagazine.com/2023/04/accessible-tap-target-sizes-rage-taps-clicks/

[9] [10] [11] Offline and background operation - Progressive web apps | MDN
https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Offline_and_background_operation