ChatGPT

Designing a Scalable Authentication UI Architecture

Folder Structure & File Layout

A clear and modular folder structure makes the codebase easy to navigate and extend. We recommend grouping components by feature (following the pattern used in the Arcstrum database UI) with separate folders for the **Auth Panel** (list view), **Auth Editor** (detail view), and supporting elements like modals and sub-sections. For example:

```
/src/components/Auth/
├── AuthPanel/                # Listing panel for authentication (e.g. users
list)
│   ├── AuthPanel.tsx         # Main component for the auth list panel
│   ├── AuthPanelToolbar.tsx  # Toolbar for search, filters, add-user button,
etc.
│   ├── AuthList.tsx          # Component rendering the list of users (could be
a table or list view)
│   └── UserListItem.tsx      # Component for an individual user entry in the
list
├── AuthEditor/               # Detail editor for a selected user (or other
auth entity)
│   ├── AuthEditor.tsx        # Main component for the auth detail panel
│   ├── AuthEditorToolbar.tsx  # Toolbar for actions on the detail view (e.g.
save, delete, resend invite)
│   ├── AuthEditorTabs.tsx    # (Optional) Tab navigation for different
sections of user detail
│   ├── UserDetailsTab.tsx    # Tab for user's basic profile info (name, email,
etc.)
│   ├── UserSecurityTab.tsx   # Tab for security info (password reset, 2FA
status, active sessions)
│   ├── UserRolesTab.tsx      # Tab for role assignments (if RBAC is enabled)
│   └── UserActivityTab.tsx   # Tab for activity logs or sessions (login
history, devices)
├── AuthProviders/            # Components for managing auth providers (OAuth,
etc.)
│   ├── ProvidersList.tsx     # List of identity providers (Google, GitHub,
Email/Magic Link, etc.)
│   └── ProviderConfigEditor.tsx  # Form for configuring a selected provider
(client IDs, secrets)
├── AuthSettings/             # Components for global auth settings and
```

```
configurations
│   ├── AuthSettings.tsx       # Settings page (e.g. toggling features like
email/password, magic links, 2FA)
│   └── EmailTemplates.tsx     # (Optional) UI for editing email/SMS template
settings
├── AuthRoles/                 # (Planned) Role management interface for RBAC
│   ├── RolesPanel.tsx         # Panel listing roles and permissions
│   └── RoleEditor.tsx         # Detail view for creating/editing a role (name,
permissions, assigned users)
├── AuthModals/                # Reusable modals for auth-related actions
│   ├── AddUserModal.tsx       # Modal to invite/create a new user (email
invite, etc.)
│   ├── EditUserModal.tsx      # Modal for editing user info (if not using
inline editor)
│   ├── ConfirmDeleteModal.tsx # Confirmation dialog for deleting/removing a
user
│   └── ... (other modals like ResetPasswordModal, etc. as needed)
└── (Optional common components, e.g. AuthFormFields.tsx for shared form inputs)
```

Explanation: This structure separates concerns by feature and component type: - **AuthPanel** contains everything for the **listing view** of users (or other auth entities). This includes a toolbar (for search, filters, and actions) and the list of items. Each user entry is a small component (`UserListItem.tsx`) to keep the list rendering efficient and the code modular. - **AuthEditor** contains the **detail view** for a selected user. It's broken into sub-components like tabs so each aspect of the user's data is managed in isolation (profile info vs. security settings vs. roles, etc.). This keeps each file focused (for instance, `UserDetailsTab.tsx` only handles the profile form, while `UserSecurityTab.tsx` handles 2FA, password resets, session info, etc.). - **AuthProviders** covers configuration of OAuth providers and other login methods. For example, a list of providers with toggles or status and a detail editor for provider config (client IDs, secrets, redirect URIs). This anticipates adding new methods like magic links or SSO by simply adding new provider entries or settings. - **AuthSettings** is for global authentication settings that aren't tied to a single user. This could include enabling/disabling login methods (email/password, magic link), enforcing 2FA globally, password policy, redirect URL configurations, etc. In top platforms like Supabase and Firebase, such settings are often on separate pages or tabs in the console [1]. By isolating them, we ensure changes to config UI don't affect user-management UI. - **AuthRoles** is a placeholder for role-based access control interfaces. If implementing RBAC, you might have a roles list and an editor for role details (e.g., assigning permissions or users to roles). Keeping it separate means teams can work on roles management independently of user CRUD UI. (If RBAC isn't in the initial release, this section can be stubbed or omitted, but the structure leaves room for it.) - **AuthModals** contains all modal dialogs related to authentication UI. Using separate files for modals (e.g., adding a user, confirming deletion) keeps the panel/editor components simpler. The modals can be imported where needed (for example, the AddUserModal might be triggered from a button in the AuthPanelToolbar).

This layout mirrors patterns used in Arcstrum's existing UI (like grouping `AuthPanel` and `AuthEditor`), making it consistent for developers. It also resembles the structure of many modern web consoles, where each major sub-feature has its own folder and internal structure. For instance, Auth0's dashboard separates user management, roles, and settings into distinct sections [2] [3], and our file layout would reflect a similar separation of concerns in code.

Key UI Components and Responsibilities

With the above structure, each UI component has a clear responsibility:

- **AuthPanel**: The main panel that lists users (or other primary auth records). It is often the left-side section in a two-pane layout. This panel fetches and displays a paginated list of users with basic info like name, email, status, and possibly an icon for their auth provider. It allows selecting a user to view/edit in the detail panel. It should support **searching and filtering** (e.g. by name, email, or by status like "enabled/disabled"), which is facilitated by the toolbar component. If multi-tenancy is supported, the AuthPanel can show only users for the current organization/project context (with an org switcher to change context).

- **AuthPanelToolbar**: A sub-component rendered at the top of the AuthPanel, containing controls for the list. Common elements include:

- A search input to filter users by name/email.
- Filter or dropdown controls (e.g. by user status or by role).
- An "Add User" button to open the **AddUserModal** (for inviting a new user or creating one manually).
- (If applicable) an **Organization Switcher** dropdown to select which organization/tenant's users to view, if the user has access to multiple orgs. This follows best practices for multi-tenant UIs – e.g., Clerk.dev allows users to seamlessly switch between organizations in the interface [4] .

- The toolbar component keeps these controls separate from the list logic, adhering to separation of concerns.

- **AuthList / UserList**: The component responsible for rendering the list of users, often as a scrollable list or table. It maps through user data and renders a **UserListItem** for each. It should handle empty states ("No users found"), loading states, and possibly infinite scroll or pagination controls if the user count is large. For scalability, if expecting thousands of users, implementing virtualization (rendering only visible items) or pagination is important to keep the UI responsive.

- **UserListItem**: A small presentational component for a single user entry in the list. It might display the user's avatar or icon, name, email, and maybe badges or indicators (e.g., "Google" or "Email" to indicate sign-up method, a lock icon if 2FA is enabled, etc.). It could also highlight disabled or blocked users differently. Having a dedicated item component makes it easy to update the styling or info displayed for each user without touching the list logic. Clicking a UserListItem selects that user (e.g., highlights it and loads the detail in AuthEditor).

- **AuthEditor**: The detail view shown when a user (or other item) is selected from the list. This is the right-hand pane in a two-pane layout. The AuthEditor acts as a container that could include a header (title, maybe user's name and ID, and high-level actions) and a set of **tabs** or sections to organize user details. Using a tabbed interface is a common pattern in complex consoles (for example, clicking a user in Auth0 opens a detailed profile with tabs for user info, metadata, logs, etc.). The AuthEditor can manage state for the selected user and coordinate saves/updates.

- **AuthEditorToolbar**: Appears at the top of the editor (often as part of the header). This can include actions like:

- “Edit” or “Save” (if the detail fields aren’t auto-saving).
- “Delete User” (with a confirmation flow).
- Other actions such as “Disable/Enable User” (to toggle access), or “Send Password Reset Email”.

- If the platform supports it, an action to **impersonate** or log in as the user (common in some admin tools for testing). These controls are isolated in their own component to keep the layout of the editor clean, and they can be conditionally rendered based on user status (e.g., show "Enable" if user is disabled).

- **AuthEditorTabs**: A sub-component that renders the navigation for detail sections (e.g., a tab list or side navigation within the editor). This is only needed if the user detail is split into multiple tabs; otherwise, a single scrollable panel could suffice. A tabbed approach is useful for separating **Profile Info**, **Security**, **Activity**, **Settings** for each user, etc. Each tab’s content is usually another component (as listed below). Keeping the tab navigation logic (which tab is active, switching tabs) in its own component or within AuthEditor state makes it easier to add or remove tabs in the future.

- **UserDetailsTab**: This component contains the **basic profile information form** for the user. For example, fields like name, email, phone number, profile photo, etc. It might also display immutable fields like unique user ID or sign-up date. If the platform allows adding custom metadata to users, those could be shown here or in a separate “Metadata” tab. The form should be designed for **editability** (if admins can edit user info) with proper validation. If some fields are federated (from Google, etc.), the UI might display them as read-only with an info note (similar to Auth0’s approach where certain profile fields from identity providers are not editable by admins [5]). This component should handle its own state and validation, and communicate changes (e.g. via context or callbacks) back to AuthEditor for saving.

- **UserSecurityTab**: This tab covers security-related settings for the user. Key elements might include:

- **Authentication Methods**: If the user has multiple sign-in methods linked (e.g., OAuth, email/password), list them or allow linking new methods.
- **Password Reset**: A button to send a reset email or a form to manually set a password (depending on admin capabilities).
- **Two-Factor Authentication (2FA)**: Show whether the user has 2FA/MFA enabled, and possibly allow admin to disable 2FA or see 2FA devices. (Some systems let admins reset a user’s 2FA if the user is locked out.)
- **Active Sessions / Devices**: List the user’s active sessions or remembered devices with an option to revoke them. For instance, Clerk’s user profile UI shows active devices and allows signing them out [6], which is a good model for a security tab.

- This tab basically centralizes everything an admin might do to help secure or troubleshoot a user’s account. Designing it as a separate component means it can be developed and tested independently (e.g., one could mock a user with sessions and see if the list renders correctly).

- **UserRolesTab**: If role-based access control is supported, this tab would allow viewing and editing the roles/groups assigned to the user. It might list roles with checkboxes or tags for roles the user has, and a way to add or remove roles from the user. If roles have more detail (like role names and descriptions), it could show those as well. Changes here might affect permissions for the user, so the UI should confirm and possibly update in real-time. By isolating this in its own component, the

complexity of managing role assignments (including fetching available roles, etc.) is kept separate. This component would interface with the roles data (perhaps via context or a roles store) to get the list of roles. In future, if an **AuthRoles** section (roles management) exists, it would use the same data source.

- **UserActivityTab** (or **UserLogsTab**): This could show audit logs related to the user or recent authentication events (login timestamps, IP addresses, etc.), if the system surfaces those. It's not explicitly requested, but some advanced consoles have this (e.g., Firebase and Auth0 let you see last login time, etc., and Auth0 provides user log events). This tab would be mostly read-only, showing a table or list of recent actions by the user (login, logout, token refresh, etc.). It's optional, but planning for it ensures the architecture can handle read-only informational tabs too.

- **AuthProviders** components: These handle **login method configuration**. As the system needs to support Google OAuth currently and plan for others (magic links, email/password, SAML, etc.), a modular UI for providers is key. The **ProvidersList** component could be a page or panel that lists all available authentication providers (Google, Facebook, GitHub, Email/Password, Magic Link, SAML, etc.), indicating which are enabled. For each provider, an admin might toggle it on/off and open a detail editor to set it up. For example, clicking "Google" could open a **ProviderConfigEditor** that has fields for Google client ID/secret and redirect URLs. We separate these components so that adding a new provider is as simple as adding a new item to the list and a new config form if needed. This follows best practices seen in Firebase and Auth0, where enabling a login method is done in a dedicated screen: e.g., in Firebase you go to the "Sign-in Method" tab to enable Google or email login and input the config. Our ProvidersList serves a similar purpose. It's also similar to how Auth0 organizes connections/providers in their dashboard settings. By planning this as a distinct part of the UI, we ensure the main user management panel (AuthPanel) is not cluttered with configuration details.

- **AuthSettings**: A component for miscellaneous auth settings that affect the whole project/tenant. This could be a separate page or a tab within an "Authentication" section of the console. Possible settings include:

- **Allow User Signups** (on/off toggle).
- **Require Email Verification** for new users.
- **Password Policy** (min length, complexity requirements).
- **Multi-Factor Authentication Policy** – e.g., whether 2FA is optional, required for certain actions, etc.
- **Magic Link Expiration** settings, etc.

- **Custom SMTP or SMS settings** if the platform allows (Supabase, for example, has custom SMTP config in its Auth settings [7]). This component collects all such settings in one form or divided into subsections. It promotes **developer ergonomics** by centralizing configuration, much like Supabase's Auth settings pages or Auth0's settings. Having a dedicated settings UI means future features (like captcha toggling, IP allowlists, etc.) can be added here without impacting other components.

- **AuthRoles** components (for RBAC): If we implement a full roles management interface, the **RolesPanel** would list all roles in the system (with perhaps a brief info like description or number of users per role) and an "Add Role" button. Selecting a role could open a **RoleEditor**, which allows editing the role name, description, and managing its permissions. Permissions could be application-

specific, but at minimum it might show which users have this role (and allow adding/removing users to the role, if not done solely from the user's side). This resembles how Auth0 or AWS Cognito might allow role management separate from user management. By keeping roles in a separate set of components, we acknowledge that roles are a first-class entity. We ensure **separation of concerns**: user management vs role management, though they intersect, each has its own UI. This also sets us up to later introduce things like **permission management** or grouping roles (if needed) without entangling that logic with the user list UI.

- **AuthModals**: Various modal dialogs to handle user actions:

- **AddUserModal**: A form in a modal for adding a new user. This might let an admin input an email (and possibly name or initial password) to invite a user. It could support both creating a password-based user or sending an invitation/magic link. Having it as a modal (versus a standalone page) keeps the admin in context of the user list. The modal on submit will call the appropriate backend API and then refresh the list.
- **EditUserModal**: (If the design chooses to edit certain info in a modal rather than inline in the editor.) This could be used for quick edits like changing a user's email or name if not done in the main editor. However, since we have a detailed editor pane, this might be unnecessary except for perhaps editing metadata or custom claims in a separate dialog.
- **ConfirmDeleteModal**: A confirmation dialog that appears when an admin attempts to delete a user or remove them from an organization. This modal should be generic (e.g., you pass in the name of the entity to delete and it displays a warning and requires confirmation).
- **ResetPasswordModal / SendMagicLinkModal**: Instead of navigating away, an admin might trigger a "reset password" or "send magic link" action which pops up a modal to confirm sending an email to the user. This provides feedback that something happened.

- **RoleAssignmentModal**: If assigning roles via a modal (for example, clicking "Add Role" for a user opens a modal with a multi-select of roles). Using modals for such actions follows UI best practices to avoid leaving the current context and to ensure confirmation for destructive actions. We isolate them in the **AuthModals** folder so they can be shared between components (e.g., the AddUserModal might be triggered from AuthPanelToolbar as well as from an Org detail screen, if one exists).

- **Common Components/Utilities**: Although not listed explicitly above, we should note that any smaller component or utility used in many places should be factored out. For instance, if multiple forms need an email input with the same validation, a common `EmailInput.tsx` could reside in a shared folder (or use a library component). Similarly, if there's a user avatar component or a badge for provider icon (Google, GitHub logos next to user), those can be shared. This prevents duplication and ensures consistency in the UI.

Each of these components is designed to have a single responsibility and to be as decoupled as possible from others, communicating via props, events, or a shared state/store. This modular breakdown makes the system easier to maintain and extend. For example, if we later add **Organization management UI**, we could introduce `OrgPanel` and `OrgEditor` components in a similar fashion, without altering the existing AuthPanel/AuthEditor much. (In fact, Auth0 and similar platforms have separate organization management screens [2], which is a pattern we can emulate when the time comes.)

Scalability & Future-Proofing

Designing with future features in mind ensures the architecture remains robust as the product grows. Some scalability considerations and how this architecture addresses them:

- **New Auth Methods and Providers:** The system already supports Google OAuth; planning for email/password, magic links, and others means our UI should be adaptable. By having the **AuthProviders** section, we can easily add new provider entries (e.g., "GitHub OAuth" or "Twitter OAuth") or new methods. If a new auth method doesn't require OAuth (like magic link or SAML SSO), it can still be represented in the Providers list or Settings with minimal changes. This modular approach is more scalable than hard-coding for one provider. It mirrors how Firebase and Auth0 allow enabling multiple identity providers via their console UIs (each provider is a separate card/row to configure). Our UI would allow adding providers without restructuring the whole auth module.

- **User Base Scaling:** As the number of users grows, the UI must handle large datasets efficiently. The architecture supports this by:

- Using a paginated or virtualized **AuthList** to only render a subset of users at any time.
- Providing search and filters in the **AuthPanelToolbar** to quickly narrow down results.

- Potentially offering export options or bulk actions (these could be added as toolbar buttons or in context menus of the panel). Additionally, separating the list and detail view means we only render the detail for one user at a time, keeping rendering work under control. If needed, we can further optimize by memoizing list items or using windowing libraries for lists.

- **Feature Toggles & Conditional UI:** Not every tenant might use all features (e.g., some might not enable 2FA or multi-tenancy). The component architecture allows optional rendering of parts of the UI. For example, the **UserRolesTab** might be shown only if RBAC is enabled for the project. Because it's a separate component, we can conditionally include it in the tab list. Similarly, the **Organization Switcher** in the toolbar is only shown if multi-tenant support is active for the account. This way, the UI is dynamic but the code for each piece remains isolated. This follows practices seen in enterprise-grade consoles where features appear or hide based on settings or user permissions.

- **Multi-Tenancy (Organization Support):** The design anticipates multi-organization support. At the UI level, this might mean:

- A dropdown (org switcher) in a global nav or in the AuthPanelToolbar to choose the active organization context (as noted earlier).
- Alternatively, a separate Organizations management page (with its own panel/editor) to create or configure organizations and their members. If needed, we could add an `OrgPanel` listing organizations and an `OrgEditor` for org details (name, billing info, members list, etc.). Because our architecture is feature-folder based, adding a new folder like `OrgPanel/OrgEditor` is straightforward and keeps the code organized.
- Within user management, the list would filter by the current org, and the **AddUserModal** would likely include an option to assign the new user to an organization (or if we're in a specific org context, it does it implicitly).

- Clerk.dev's solution allows users to switch orgs easily in the UI [4], and Auth0 provides an org context for B2B apps [2]. We take inspiration from those by ensuring our UI can handle that extra layer of grouping.

- **Scalability:** Multi-tenant support also means potentially more data (users segmented by org). Our design keeps the data scope controlled by context — e.g., we load only users for one org at a time — which is efficient and maintains clarity in the UI (the admin knows which set of users they are viewing).

- **Extending RBAC:** If we introduce fine-grained permissions or more complex role hierarchies later, the presence of a Roles management component means we have a place to extend that functionality. The UI could grow to include permission sets in the RoleEditor, or a **PermissionsTab** if needed. The key is that by establishing these sections now, we make future enhancements less disruptive.

- **Integration with Other Console Areas:** The file structure under `/Auth` means all auth-related UI lives together. If the console has a main navigation (e.g., "Database", "Storage", "Auth", "Logs"), this Auth section is self-contained. This modularity allows independent development and also the possibility to reuse parts in other contexts (for example, maybe showing a mini user list in a dashboard view, etc., by reusing AuthPanel).

In summary, the architecture is **scalable** both in terms of data (supporting many users and orgs) and in terms of product features (easily accommodating new auth methods, settings, and management screens).

Accessibility Considerations

Accessibility is a must for a production-grade system. We should ensure all UI components are built to be **ARIA-compliant, keyboard navigable, and screen-reader friendly**: - Use semantic HTML elements where appropriate. For example, the user list could be a semantic table or list (`<ul>` / `<li>` or `<table>`) with proper headings if tabular. This helps screen readers announce content properly. - The **AuthPanelToolbar** controls (search input, buttons, dropdowns) should have proper `<label>` tags or aria-labels. The add-user button should be reachable via keyboard (e.g., using Tab key) and have an `aria-label="Invite new user"` or similar. - The tabs in AuthEditor should be implemented as an accessible tab list (using `<div role="tablist">` and `<button role="tab">` or appropriate ARIA attributes if using custom elements). This allows users to navigate through tabs via keyboard (often with arrow keys) and screen readers to identify them as tabs. - Color contrast needs to be sufficient for text on any backgrounds (following WCAG AA or AAA standards). For instance, status indicators (enabled/disabled user) should not rely on color alone; include an icon or text label for "Disabled" so that color-blind users or screen reader users get the information (e.g., an icon with `aria-label="Disabled user"`). - **Focus management:** When modals open, focus should be trapped within the modal and return to the triggering element after closing (this can be handled by using a common modal component or library that ensures this). Also, when switching between the list and detail via keyboard, we should manage focus (e.g., if a user uses arrow keys to navigate the list and presses Enter to open details, focus could move to the first field in the detail panel). - **Form elements** in UserDetailsTab and others should have visible labels or at least aria-labels. Any important image (like perhaps user avatar) should have an `alt` text (like the user's name or "User avatar"). - We should also consider **internationalization** (i18n) as part of accessibility/global accessibility – all text strings should be externalized for easy translation, and the layout should accommodate different

lengths of text. - For components like the Organization Switcher or Role assignments, ensure that all interactive controls are reachable without a mouse. E.g., a dropdown list of orgs should be operable via keyboard (arrow down to open, etc.). - **Responsive design**: While a separate mobile app is planned, the web console should ideally be responsive enough to at least function on smaller screens or zoom levels (this also ties into accessibility for low-vision users who zoom in). Using flexible layouts or CSS grid can ensure the two-panel layout can collapse into one column on very narrow screens, for example.

By baking in these accessibility practices, we follow the lead of top tech companies (Google, Apple, Microsoft all emphasize accessibility in their products). This not only widens the potential user base but often leads to cleaner, more semantic code which is easier to maintain.

Developer Ergonomics & Reusability

A well-structured architecture improves developer experience and long-term maintainability:

- **Modularity and Separation of Concerns:** Each folder and component has a focused purpose, so developers know *where* to add or modify features. For instance, a developer working on the new magic link feature will likely touch **AuthSettings** (to add a toggle or config for magic links) and maybe **AuthProviders** (to show Magic Link as an auth method), without needing to tangle with the user list or editor code. This reduces risk of regressions. Similarly, if a bug is found in user detail editing, one can go straight to `AuthEditor/UserDetailsTab.tsx` rather than wading through a monolithic file.
- **Consistency with Existing Patterns:** Since the structure aligns with how the Arcstrum console already organizes code (e.g., DatabaseEditor, DatabasePanel), developers familiar with the project can easily pick up the auth module. Consistent naming (AuthPanel, AuthEditor, etc.) means there's no guesswork in finding files. This consistency is crucial for team development and on-boarding new engineers.
- **Reusable Components:** We aim to make components as reusable as possible. For example, the form inputs in the AddUserModal could reuse the same components as in UserDetailsTab (to avoid duplication). If the console has a design system or component library (buttons, inputs, switches, etc.), all these auth components would use those, ensuring a unified look and feel. This not only makes the UI cohesive but also means less custom styling to maintain.
- **Cross-Platform Sharing:** The UI needs to be reused on web and mobile. To facilitate this, we can separate pure layout/markup from business logic:
- For instance, data-fetching hooks (for loading users or roles) can reside outside the UI components (maybe in a `/src/hooks/authHooks.ts` or context providers). The components then simply consume data via props or context. This way, a mobile implementation (say React Native) could reuse the *logic* (perhaps via a shared service or even by calling the same APIs) but render using native components.
- If using a framework that supports cross-platform components (like React with React Native Web or Flutter with common widget logic), we could even share some component code. Even if not directly, having a clear separation makes rewriting in another platform easier because you can follow the same structure. The "console's pattern" can be applied in mobile by creating analogous components (maybe a mobile AuthPanel that uses a mobile-friendly list, etc.).
- We ensure no platform-specific assumptions are baked into the components. For example, avoid direct DOM manipulations that wouldn't work in React Native; instead use abstracted libraries or pass in platform-specific handlers. On web we might use an HTML table for user list; on mobile, a

FlatList – but both could be fed by the same data source. Designing the interface of these components with that in mind (e.g., separating style from data) improves reuse.

- **Scalability of Development:** As the project grows, multiple developers or teams might work on different auth features concurrently. With this structure, one team can safely work on Role management (in AuthRoles folder) while another works on UI for new providers, without stepping on each other's toes. Merge conflicts are minimized since files are granular. It also encourages writing smaller, testable units (you can write unit tests for, say, the UserListItem or the OrgSwitcher easily in isolation).
- **Documentation and Discoverability:** Each folder or component can contain a README or comments at the top explaining its purpose. Since the question is about design and not actual code, we mention this as a practice: e.g., a brief README in `/Auth` can outline how data flows, what each sub-component does, and how to add a new one. This helps future contributors. In code, prop types or TypeScript interfaces for each component serve as documentation for how to use them.
- **Use of Best Practices & References:** We took inspiration from established platforms. For example, Clerk's approach of offering both ready UI components and an admin dashboard [8] shows the importance of having a robust internal UI for user management, not just end-user auth components. Auth0's separation of concerns (users vs roles vs orgs) [2] influenced our folder breakdown. By referencing these, we ensure our design isn't reinventing the wheel but standing on shoulders of well-tested solutions. This gives developers confidence that the patterns chosen will work at scale because they mirror what successful platforms do.
- **Performance and Optimization:** Developers will find it easier to optimize this app because of the clear component boundaries. For example, if rendering the user list becomes slow, one can optimize the `AuthList` or `UserListItem` (using `React.memo` or similar) without touching anything in the editor or settings. Likewise, if a certain tab (say UserActivityTab) makes heavy API calls, we could implement lazy loading for that tab (load data only when the tab is active), which is straightforward when each tab is a separate component.
- **Testing:** This architecture also improves testability. Each piece can be unit-tested with mock data (e.g., feed a fake user list to AuthList and verify it renders correctly, test that AuthEditorTabs switches content, etc.). End-to-end tests can focus on integration between these components. Having smaller components usually means fewer complex interactions in each, making both manual and automated testing more manageable.

In conclusion, this frontend component architecture provides a **structured, modular, and scalable** foundation for Arcstrum's authentication console. It balances an **intuitive UI** (inspired by best-in-class developer platforms) with clean separation of concerns under the hood. The folder layout and component breakdown will make it easy for indie developers to feel at home (thanks to a clear and friendly interface), while also offering the power and extensibility needed for teams managing complex auth configurations. By planning ahead for OAuth providers, magic links, 2FA, RBAC, and multi-tenant support, we ensure the **authentication system's UI can grow** alongside future requirements without needing a costly refactor. Each new feature can be slotted into this structure logically. The result is a frontend architecture that is maintainable for developers, welcoming for users, and robust enough for production.

---

[1] [7] User Management | Supabase Docs
https://supabase.com/docs/guides/auth/managing-user-data

[2] Manage Users
https://auth0.com/docs/manage-users

3 5 Manage Users Using the Dashboard
https://auth0.com/docs/manage-users/user-accounts/manage-users-using-the-dashboard

4 Navigating Multi-Tenancy and User Experience with Clerk: A Deep Dive into RapidStream’s Approach | by Maurice Renck | konzentrik | Medium
https://medium.com/konzentrik/navigating-multi-tenancy-and-user-experience-with-clerk-a-deep-dive-into-rapidstreams-approach-7e624bfbf224

6 8 Clerk | Authentication and User Management
https://clerk.com/