JustScore Onboarding Documentation

Table of Contents

Overview

The JustScore onboarding flow is a multi-step process that guides users through the initial setup of their organization, action selection, team creation, and member management. It uses Next.js (App Router), React, and the Zustand state management library along with custom hooks for data management.

The onboarding is designed to be a progressive disclosure of information, with each step building on the previous one. The user can navigate back and forth between steps, and their progress is saved automatically through Zustand store and localStorage.

File Structure

Below is the structure of files related to the onboarding flow:

```
app/
├── dashboard/
│   ├── onboarding/
│   │   ├── components/
│   │   │   ├── ActionsSelector.tsx          # Action selection component
│   │   │   ├── MemberForm.tsx               # Member form component
│   │   │   ├── OnboardingHeader.tsx         # Deprecated header component
│   │   │   ├── PageNavigator.tsx            # Navigation between steps
│   │   │   ├── ProgressBar.tsx              # Progress indicator
│   │   │   ├── TeamForm.tsx                 # Team creation/edit form
│   │   │   └── TeamList.tsx                 # List of created teams
│   │   ├── function-actions/
│   │   │   └── page.tsx                     # Function Actions step
│   │   ├── global-actions/
│   │   │   └── page.tsx                     # Global Actions step
│   │   ├── intro/
│   │   │   └── page.tsx                     # Intro/landing page
│   │   ├── members/
│   │   │   └── page.tsx                     # Team Members step
│   │   ├── organisation/
│   │   │   └── page.tsx                     # Organisation info step
│   │   ├── summary/
│   │   │   └── page.tsx                     # Final review/summary step
│   │   ├── teams/
│   │   │   └── page.tsx                     # Teams creation step
│   │   ├── layout.tsx                       # Shared layout for all steps
│   │   ├── onboarding.css                   # Onboarding-specific styles
│   │   └── page.tsx                         # Redirect page
│   └── components/
│       └── configuration/
│           ├── ConfigurationOrganizationSummary.tsx  # Org summary component
│           ├── ConfigurationActionsSummary.tsx       # Actions summary component
│           └── ConfigurationTeamsSummary.tsx         # Teams summary component
hooks/
├── useActionsSelection.ts                  # Hook for actions selection
├── useMemberManagement.ts                  # Hook for member management
├── useOnboardingConfig.ts                  # Central config management hook
├── useOnboardingNavigation.ts              # Navigation management hook
├── useOnboardingValidation.ts              # Validation hook
├── useOrganisation.ts                      # Organisation management hook
├── useSetupProgress.ts                     # Progress tracking hook
├── useSetupStateSync.ts                    # State sync with localStorage
└── useTeamsManagement.ts                   # Teams management hook
lib/
└── utils/
    ├── onboarding-team.ts                  # Team-related utility functions
    └── validation.ts                          # Validation utility functions
store/
├── action-store.ts                            # Store for action-related data
├── config-store.ts                            # Main configuration store
├── config-types.ts                            # Types for config store
├── favorites-store.ts                         # Store for favorite actions
├── onboarding-store.ts                        # Onboarding state store
└── setup-store.ts                             # Setup progress store
```

Component and Hook Relationships

Core Components

1. `layout.tsx`
   - Provides the shared layout for all onboarding steps
   - Includes the ProgressBar component
   - Handles transitions between steps
2. `PageNavigator.tsx`
   - Manages navigation between steps
   - Shows the step title and description
   - Handles validation before navigation
   - Shows validation errors when navigation fails
   - Provides direct callback for validation failures with `onValidationAttempt`
3. `ProgressBar.tsx`
   - Shows visual progress through the onboarding steps
   - Calculates and displays completion percentage
4. `ActionsSelector.tsx`
   - Used in both Global Actions and Function Actions steps
   - Allows users to select actions from different categories
   - Handles validation for required action selections
5. `TeamForm.tsx` & `TeamList.tsx`
   - Used in the Teams step
   - Allows creation and management of teams
   - Shows list of created teams
   - Validates team data before creation/updating
6. `MemberForm.tsx`
   - Used in the Members step

- Handles adding, editing, and removing team members

Core Hooks

1. `useOnboardingConfig.ts`
   - Central hook for configuration management
   - Provides validation functions for each step
   - Offers navigation helpers and access to the config store
2. `useOnboardingValidation.ts`
   - Manages validation for each step
   - Provides error state management
   - Centralizes error message display logic
3. `useOnboardingNavigation.ts`
   - Handles navigation between steps
   - Validates before navigation
   - Manages error display
4. `useActionsSelection.ts`
   - Manages action selection across both action steps
   - Handles category filtering and action counting
   - Provides validation for action selection
5. `useMemberManagement.ts`
   - Manages member data in localStorage
   - Handles form state for adding and editing members
   - Provides validation for member data
6. `useTeamsManagement.ts`
   - Manages team creation and editing
   - Handles validation for team data
   - Synchronizes with config store and localStorage
7. `useOrganisation.ts`
   - Manages organization data in the config store
   - Provides validation for organization name
   - Handles form state for organization info

Step-by-Step Validation and Data Flow

Updated Validation Mechanism

The onboarding flow implements a straightforward validation approach:

1. Each step has a `canContinue` condition that's evaluated when the user tries to proceed
2. If validation fails, error messages are shown directly in the UI
3. The `PageNavigator` component shows a top-level error alert
4. Individual form fields show specific validation errors
5. Errors only appear after a navigation attempt or form submission, not on initial load

Step 1: Organisation

Validation Requirements:

- Organization name must not be empty

Data Flow:

1. User enters organization name
2. `useOrganisation` hook updates the config store
3. When user clicks "Next", validation checks if the name is not empty
4. If validation fails, error messages are displayed
5. PageNavigator only enables navigation when validation passes

Step 2: Global Actions

Validation Requirements:

- At least 3 actions must be selected from each mandatory category

Data Flow:

1. `useActionsSelection` loads action categories
2. User selects actions from each category
3. Selections are stored in the config store
4. When user clicks "Next", validation checks for minimum selections per category
5. If validation fails, error messages are displayed

Step 3: Function Actions

Validation Requirements:

- At least 1 action must be selected from at least 1 function category

Data Flow:

1. `useActionsSelection` loads non-mandatory categories

2. User selects actions from relevant function categories
3. Selections are stored in the config store
4. When user clicks "Next", validation checks if at least one function has actions selected
5. If validation fails, error messages are displayed

Step 4: Members

Validation Requirements:

- At least 1 member must be added with a valid name and email

Data Flow:

1. `useMemberManagement` initializes with existing members from localStorage
2. User adds members with required fields
3. Member data is stored in localStorage
4. When user clicks "Next", validation checks if at least one valid member exists
5. If validation fails, error messages are displayed

Step 5: Teams

Validation Requirements:

- At least 1 team must be created with:
  - A name
  - At least 1 function
  - At least 1 member

Data Flow:

1. `useTeamsManagement` loads member data and function categories
2. User creates teams with required fields
3. Team data is stored in the config store and team member assignments in localStorage
4. When user clicks "Create Team" or "Next", validation checks if required fields are filled
5. If validation fails, error messages are displayed for specific fields
6. Team creation only proceeds when all validation passes

Step 6: Summary

Validation Requirements:

- All previous steps must be completed

Data Flow:

1. `useOnboardingConfig` checks the completion status of all steps
2. Summary components display the collected configuration
3. User confirms the setup
4. Configuration is submitted to the API
5. User is redirected to the dashboard

User Experience Flow

Step 1: Intro Page

- **UI Elements**: Overview of the onboarding process, step cards, start button
- **User Actions**: Click "Start Onboarding" to begin the process
- **Data Requirements**: None
- **Next Step**: Organization page

Step 2: Organisation Page

- **UI Elements**: Form with organization name input field
- **User Actions**: Enter organization name
- **Data Requirements**: Valid organization name
- **Validation**: Error messages only appear after attempting to navigate forward
- **Next Step**: Global Actions page

Step 3: Global Actions Page

- **UI Elements**: Category list, action selector, selection counters
- **User Actions**: Select actions from each mandatory category
- **Data Requirements**: Minimum number of actions per mandatory category
- **Validation**: Error messages shown when trying to proceed without required selections
- **Next Step**: Function Actions page

Step 4: Function Actions Page

- **UI Elements**: Category list, action selector, selection indicators
- **User Actions**: Select relevant function-specific actions
- **Data Requirements**: At least one function with selected actions
- **Validation**: Error messages shown when trying to proceed without required selections
- **Next Step**: Members page

Step 5: Members Page

- **UI Elements**: Member form, member list, edit/delete controls
- **User Actions**: Add team members with names, emails, and optional job titles
- **Data Requirements**: At least one member with valid name and email
- **Validation**: Validates form fields before adding members, and step completion before navigation
- **Next Step**: Teams page

Step 6: Teams Page

- **UI Elements**: Team form, team list, function and member selectors
- **User Actions**: Create teams and assign functions and members
- **Data Requirements**: At least one team with name, functions, and members
- **Validation**: Validates team data before creation and when navigating to next step
- **Next Step**: Summary page

Step 7: Summary Page

- **UI Elements**: Configuration summary cards, completion button
- **User Actions**: Review configuration and click "Complete Setup"
- **Data Requirements**: All previous steps completed
- **Next Step**: Dashboard

State Management

The onboarding flow uses three main approaches for state management:

1. **Zustand Store**
   - `config-store.ts`: Central store for configuration data
   - Handles organization info, action selections, and team data
   - Persisted with the Zustand persist middleware
2. **localStorage**
   - Used for data that doesn't fit the config model
   - Stores member data and team member assignments
   - Accessed through custom hooks
3. **React State**
   - Used for UI state like form fields, errors, and component state
   - Managed through custom hooks for each feature
   - Handles validation state and error display

Key State Objects

1. **Organisation**

   ```typescript
   {
     name: string;
   }
   ```

2. **Activities**

   ```typescript
   {
     selected: string[];
     selectedByCategory: Record<string, string[]>;
     favorites: Record<string, string[]>;
     hidden: Record<string, string[]>;
   }
   ```

3. **Teams**

   ```typescript
   Array<{
     id: string;
     name: string;
     functions: string[];
     categories: string[];
   }>
   ```

4. **Members** (in localStorage)

   ```typescript
   Array<{
     id: string;
     fullName: string;
     email: string;
     jobTitle: string;
   }>
   ```

5. **Team Members** (in localStorage)

```typescript
Array<{
  teamId: string;
  memberIds: string[];
}>
```

Navigation System

The onboarding uses Next.js App Router for navigation between steps. The flow is enhanced with:

1. **PageNavigator Component**
   - Provides consistent navigation UI
   - Handles validation before navigation
   - Shows step progress indicators
   - Displays validation errors directly in the UI
   - Supports validation callbacks for direct page communication
2. **useOnboardingNavigation Hook**
   - Manages programmatic navigation
   - Handles error display
3. **ProgressBar Component**
   - Shows visual progress through the flow
   - Calculates percentage based on current step
4. **Error Handling**
   - Errors are shown directly in the UI after validation attempts
   - Alert components show at the top of the page
   - Field-level errors highlight specific inputs
   - Errors are cleared when user makes corrections

Styling Guidelines

The onboarding uses Tailwind CSS v4 for styling with the following patterns:

1. **Card-based Layout**
   - Each step uses a Card component for content
   - Consistent padding and spacing
2. **Responsive Design**
   - Mobile-first approach

- Grid layouts adjust for different screen sizes

3. **Visual Hierarchy**
   - Clear typography with headings and descriptions
   - Important actions highlighted with appropriate colors
4. **Consistency**
   - Similar form controls across all steps
   - Consistent error display mechanisms
   - Unified button styles and interactions
5. **Accessibility**
   - Proper ARIA attributes for interactive elements
   - Error messages linked to form fields
   - Keyboard navigation support

Best Practices and Common Pitfalls

Best Practices

1. **Use Custom Hooks**: Extract complex logic into custom hooks for better separation of concerns.
2. **Simple, Direct Validation**: Keep validation logic simple and handle errors directly in components.
3. **Keep Components Focused**: Each component should have a single responsibility.
4. **Centralize Types**: Share type definitions across files to maintain consistency.
5. **Optimize Rendering**: Use `useMemo` and `useCallback` for expensive calculations and functions.
6. **Show Errors Only After User Interaction**: Don't display validation errors on initial load, only after the user attempts to proceed or submit a form.
7. **Use Consistent Error Patterns**: Display errors consistently across all steps for a uniform user experience.

Common Pitfalls

1. **Stale Data**: When removing items from UI, ensure they're also removed from persistent storage.
2. **Validation Inconsistency**: Make sure validation logic is consistent between the UI and hooks.
3. **Navigation Issues**: The back button should preserve form state, and Next.js router can cause issues with caching.
4. **Complex Event Systems**: Avoid overly complex event systems for validation - direct callbacks are often clearer.
5. **localStorage Limitations**: Be cautious of localStorage size limits and handle JSON parse errors.

Implementation Notes

When implementing changes to the onboarding flow, consider the following:

1. **Hook Dependencies**: Ensure all hook dependencies are correctly specified.
2. **State Synchronization**: Keep state synchronized between components and stores.
3. **Form Reset**: Always reset forms after successful submission.
4. **Step Completion**: Validation for step completion should be consistent.
5. **Error Messages**: Provide clear and descriptive error messages.
6. **Debug Carefully**: Use console logging strategically to debug validation issues.
7. **Test Edge Cases**: Test all validation scenarios, including edge cases.

Conclusion

The JustScore onboarding flow is a complex but well-structured system that guides users through the setup process. By maintaining a clear separation of concerns between UI components and business logic hooks, the codebase remains maintainable and extensible.

Each step builds on the previous ones, collecting the necessary data to create a complete configuration. The validation system ensures users provide all required information before proceeding, while the navigation system allows for easy movement between steps.

The updated validation approach focuses on simplicity and directness, showing errors only when needed and providing clear guidance to users. This enhances the user experience while maintaining code clarity.

By following the patterns and practices outlined in this documentation, developers can maintain and extend the onboarding flow with confidence.