Onboarding Guide

Fokuslah Developer Onboarding Guide

Welcome to the Fokuslah team! This guide will help you get up and running with our codebase, development practices, and team workflows.

🎯 Project Overview

Fokuslah is a Duolingo-style math learning application designed for teenagers. Our goal is to make math practice engaging and rewarding through gamification elements like XP, levels, and streaks.

Key Features

- Interactive math lessons with multiple choice and input questions
- Progressive leveling system (RPG-style)
- Daily streak tracking
- XP-based progression
- Mobile-first responsive design

🏗️ Tech Stack

Frontend

- **Next.js 15** with App Router
- **React 19** with Server Components
- **TypeScript** for type safety
- **Tailwind CSS 4** for styling
- **React Query** for data fetching and caching

Backend

- **Hono** for API routes (RPC-style)

- **Prisma ORM** with PostgreSQL
- **NextAuth.js** for authentication

UI Components

- **Radix UI** for accessible primitives
- **Lucide React** for icons
- **Framer Motion** for animations

Development Tools

- **ESLint** and **Prettier** for code quality
- **Jest** and **React Testing Library** for testing
- **GitHub Actions** for CI/CD

📁 Project Structure

```
fokuslah/
├── app/                # Next.js App Router pages
├── components/         # Shared UI components
│   ├── common/         # Reusable components
│   ├── layout/         # Layout components
│   └── ui/             # Primitive UI components
├── constants/          # Application constants
├── docs/               # Documentation
├── features/           # Feature modules
│   ├── [feature]/      # Individual feature
│   │   ├── api/        # Frontend API hooks
│   │   ├── components/  # Feature-specific components
│   │   ├── server/     # Backend API routes
│   │   ├── types/      # Feature-specific types
│   │   └── index.ts    # Feature exports
├── lib/                # Core library functions
├── prisma/             # Database schema and migrations
├── providers/          # React context providers
├── public/            # Static assets
├── types/             # Shared TypeScript types
├── utils/             # Utility functions
└── __tests__/          # Test files
    ├── unit/          # Unit tests
    └── integration/     # Integration tests
```

🚀 Getting Started

Prerequisites

1. **Node.js** (v18 or higher)
2. **npm** or **yarn** package manager
3. **PostgreSQL** database
4. **Git** for version control

Installation

```
# 1. Clone the repository
git clone [repository-url]
cd fokuslah

# 2. Install dependencies
npm install

# 3. Set up environment variables
cp .env.example .env.local
# Update .env.local with your configuration

# 4. Set up the database
npx prisma generate
npx prisma migrate dev

# 5. Seed the database (optional)
```

```
npx prisma db seed

# 6. Start the development server
npm run dev
```

Environment Variables

Create a `.env.local` file with these variables:

```
# Database
DATABASE_URL="postgresql://user:password@localhost:5432/fokuslah"
DIRECT_URL="postgresql://user:password@localhost:5432/fokuslah"

# NextAuth
NEXTAUTH_URL="http://localhost:3000"
NEXTAUTH_SECRET="your-secret-key"

# Application
NEXT_PUBLIC_APP_URL="http://localhost:3000"
```

🧪 Development Workflow

Daily Workflow

1. **Start the development server**: `npm run dev`
2. **Run tests**: `npm run test`
3. **Check code quality**: `npm run lint`
4. **Format code**: `npm run format`

Creating a New Feature

1. **Create a feature branch**:

   ```
   git checkout -b feature/your-feature-name
   ```

2. **Create the feature directory structure**:

```
features/your-feature/
├── api/
├── server/
├── components/
├── types/
└── index.ts
```

3. **Follow the feature structure pattern** (see example below)
4. **Write tests** in `__tests__/` directory
5. **Commit with conventional commits**:

```
git commit -m "feat: add user profile page"
```

6. **Push and create a pull request**

Feature Structure Example

Let's say you're creating a "leaderboard" feature:

```
features/leaderboard/
├── api/
│   ├── use-get-leaderboard.ts    # Query hook for fetching leaderboard
│   └── index.ts                  # API exports
├── server/
│   ├── route.ts                  # Hono API routes
│   └── index.ts                  # Server exports
├── components/
│   ├── leaderboard-card.tsx      # Feature-specific component
│   └── index.ts                  # Component exports
├── types/
│   ├── leaderboard.ts            # Type definitions
│   └── index.ts                  # Type exports
└── index.ts                      # Feature exports
```

📊 Code Organization Principles

Feature-Based Architecture

We organize code by feature rather than by technology. This makes it easier to:

- Find related code
- Work on features independently
- Scale the team
- Reduce merge conflicts

Component Organization

- `/components/common`: Reusable components used across multiple features
- `/components/layout`: Layout and structural components
- `/components/ui`: Primitive UI components (buttons, cards, etc.)
- `/features/[feature]/components`: Feature-specific components

Type Management

- `/types`: Shared types used across multiple features
- `/features/[feature]/types`: Feature-specific types
- Always export types from index files for easy importing

🛠️ Development Practices

Naming Conventions

- **Files**: kebab-case (`user-profile.ts`)
- **Components**: PascalCase (`UserProfileCard`)
- **Variables/Functions**: camelCase (`getUserProfile`)
- **Constants**: UPPER_SNAKE_CASE (`MAX_RETRY_ATTEMPTS`)
- **React Hooks**: `use` prefix (`useGetUserProfile`)

TypeScript Guidelines

- Always define interfaces for component props
- Use strict typing for API responses
- Avoid `any` type unless absolutely necessary
- Export types from feature `index.ts` files

Component Development

```
// Good example
interface UserProfileCardProps {
  user: User;
  isLoading?: boolean;
  onEdit?: () => void;
}

export const UserProfileCard = ({
  user,
  isLoading = false,
  onEdit
}: UserProfileCardProps) => {
  // Component implementation
};
```

API Development

```
// Frontend hook (features/profile/api/use-get-profile.ts)
export const useGetProfile = () => {
  return useQuery<UserProfile>({
    queryKey: QUERY_KEYS.PROFILE,
    queryFn: async () => {
      const response = await client.api.profile.$get();

      if (!response.ok) {
        const errorMessage = await handleApiResponseError(response);
        throw new Error(errorMessage);
      }

      const { data } = await response.json();
      return data;
    }
  });
};
```

🧪 Testing

Unit Tests

Place unit tests in `__tests__/unit/`:

```
__tests__/unit/
├── utils/
│   ├── format.test.ts
│   └── error.test.ts
└── components/
    └── user-profile-card.test.tsx
```

Integration Tests

Place integration tests in `__tests__/integration/`:

```
__tests__/integration/
├── api/
│   └── profile-api.test.ts
└── features/
    └── profile-flow.test.ts
```

Running Tests

```
# Run all tests
npm run test

# Run tests in watch mode
npm run test:watch

# Run specific test file
npm run test:unit utils/format.test.ts
```

🎨 UI/UX Guidelines

Responsive Design

- Mobile-first approach
- Test on common screen sizes (390px, 768px, 1024px, 1280px)
- Use Tailwind's responsive prefixes (`sm:`, `md:`, `lg:`)

Accessibility

- Use semantic HTML
- Provide proper alt text for images
- Ensure sufficient color contrast
- Use ARIA attributes when needed

Performance

- Optimize images and assets
- Use code splitting for large components
- Implement proper loading states
- Minimize bundle size

🔄 Git Workflow

Branch Strategy

- **Main branch:** `main` (production-ready code)
- **Feature branches:** `feature/feature-name`
- **Bug fix branches:** `fix/bug-description`
- **Hotfix branches:** `hotfix/urgent-fix`

Commit Guidelines

Use conventional commits:

- `feat:` New feature
- `fix:` Bug fix
- `docs:` Documentation changes
- `style:` Code style changes
- `refactor:` Code refactoring
- `test:` Adding or updating tests
- `chore:` Maintenance tasks

Example:

```
git commit -m "feat: add user profile level display"
git commit -m "fix: resolve streak calculation bug"
```

Pull Request Process

1. Ensure all tests pass
2. Update documentation if needed
3. Assign reviewers from the team
4. Address feedback promptly
5. Squash and merge after approval

🚨 Error Handling

Client-Side Errors

```
import { handleApiResponseError } from "@/utils/error";

try {
  const response = await apiCall();
  if (!response.ok) {
    throw new Error(await handleApiResponseError(response));
  }
} catch (error) {
  // Handle error gracefully
  toast.error(handleApiError(error));
}
```

Server-Side Errors

```
import { HTTPException } from "hono/http-exception";

// In API routes
if (!userId) {
  throw new HTTPException(403, { message: "Unauthenticated" });
}
```

📚 Learning Resources

Documentation

- Next.js Documentation
- React Documentation
- TypeScript Handbook
- Prisma Documentation
- Hono Documentation

Codebase Specific

- Review existing features to understand patterns
- Check the `docs/` directory for additional guides
- Ask questions in our development channel

Getting Help

- **Team Lead**: [Name/Contact]
- **Slack Channel**: #fokuslah-dev
- **Code Reviews**: All PRs require review
- **Pair Programming**: Available on request

✅ First Week Checklist

Day 1

- [ ] Set up development environment
- [ ] Get access to repositories and tools
- [ ] Review this onboarding guide
- [ ] Run the application locally
- [ ] Meet with team lead for introduction

Day 2-3

- [ ] Review existing codebase structure
- [ ] Understand the main features
- [ ] Run existing tests
- [ ] Make a small change and submit PR

Day 4-5

- [ ] Work on a small feature with guidance
- [ ] Participate in code review

- [ ] Ask questions about unclear areas
- [ ] Document any missing information

🎉 Welcome to the Team!

We're excited to have you on board. Don't hesitate to ask questions, suggest improvements, or share your ideas. Our team values collaboration, learning, and building great software together.

Happy coding! 🚀