If you'd like to gather all your form configurations in a **single folder**, while still maintaining a scalable and modular structure, you can organize the configurations within that folder and use a pattern to manage different categories effectively. This approach will keep everything in one place but ensure that it remains organized and easy to scale as the number of forms and categories increases.

Suggested Folder Structure

You can create a **`formConfigs/`** directory and group configurations by categories, each stored in separate files, but still within this single folder. You can then create an index file to easily import all the configurations from one place.

Here's how you can structure it:

```
src/
  ├── components/
  │   ├── form/
  │   │   ├── DynamicForm.tsx
  │   │   ├── FieldRenderer.tsx
  │   │   └── fields/
  │   └── ui/
  │       └── Button.tsx
  ├── hooks/
  │   └── useDynamicForm.ts
  ├── schemas/
  │   ├── candidateFormSchema.ts
  │   ├── jobSeekerFormSchema.ts
  │   └── validationSchemas.ts
  ├── store/
  │   └── useFormStore.ts
  ├── utils/
  │   └── formConfigFactory.ts
  ├── formConfigs/          // Single folder for all form configs
  │   ├── candidateFormConfig.ts
  │   ├── jobSeekerFormConfig.ts
  │   ├── companyFormConfig.ts  // Example of another form category
  │   └── index.ts            // Export all configurations from here
  └── pages/
      ├── CandidateFormPage.tsx
      └── JobSeekerFormPage.tsx
```

---

1. **Index File for Configuration Exports**

You can create an `index.ts` inside the `formConfigs/` folder to easily manage and access all your configurations in one place. This will allow you to import them dynamically into the form.

`formConfigs/index.ts`

```typescript
// src/formConfigs/index.ts

import { candidateFormConfig } from './candidateFormConfig';
import { jobSeekerFormConfig } from './jobSeekerFormConfig';
import { companyFormConfig } from './companyFormConfig';  // Another example

export {
  candidateFormConfig,
  jobSeekerFormConfig,
  companyFormConfig
};
```

Now, in your `formConfigFactory.ts`, you can just import everything from this index and dynamically access the necessary category configuration.

`formConfigFactory.ts`

```typescript
// src/utils/formConfigFactory.ts

import * as formConfigs from "../formConfigs";

export const formConfigFactory = (category: string) => {
  switch (category) {
    case "candidate":
      return formConfigs.candidateFormConfig;
    case "jobSeeker":
      return formConfigs.jobSeekerFormConfig;
    case "company":
      return formConfigs.companyFormConfig;
    // Add more categories as needed
    default:
      throw new Error(`Unknown form category: ${category}`);
  }
};
```

By gathering all the configurations in one folder, you can access any configuration dynamically and ensure that new categories can be added easily.

---

2. **Form Configurations in a Single Folder**

You can now keep all your form configurations in individual files within the `formConfigs/` folder. Here’s an example:

`candidateFormConfig.ts`

```typescript
```

```typescript
// src/formConfigs/candidateFormConfig.ts

export const candidateFormConfig = {
  category: "candidate",
  fields: [
    {
      type: "text",
      name: "fullName",
      label: "Full Name",
      placeholder: "Enter your full name",
    },
    {
      type: "email",
      name: "email",
      label: "Email Address",
      placeholder: "Enter your email",
    },
    {
      type: "text",
      name: "phone",
      label: "Phone Number",
      placeholder: "Enter your phone number",
    },
    {
      type: "select",
      name: "gender",
      label: "Gender",
      options: [
        { value: "male", label: "Male" },
        { value: "female", label: "Female" },
        { value: "other", label: "Other" },
      ],
    },
    {
      type: "checkbox",
      name: "experience",
      label: "Do you have work experience?",
    },
    {
      type: "date",
      name: "availableStartDate",
      label: "Available Start Date",
      placeholder: "Select a date",
    },
  ],
};
```

`jobSeekerFormConfig.ts`

```typescript
// src/formConfigs/jobSeekerFormConfig.ts
```

```
export const jobSeekerFormConfig = {
  category: "jobSeeker",
  fields: [
    {
      type: "text",
      name: "resumeLink",
      label: "Resume Link",
      placeholder: "Enter a link to your resume",
    },
    {
      type: "text",
      name: "portfolioLink",
      label: "Portfolio Link",
      placeholder: "Enter a link to your portfolio",
    },
    {
      type: "checkbox",
      name: "lookingForRemote",
      label: "Are you looking for remote opportunities?",
    },
    {
      type: "select",
      name: "preferredLocation",
      label: "Preferred Work Location",
      options: [
        { value: "onsite", label: "On-site" },
        { value: "remote", label: "Remote" },
      ],
    },
  ],
};
```

---

3. **Handling Forms with Zustand**

Zustand can still be used to manage the state of all forms in the app. Here’s how you can keep the global form state:

`useFormStore.ts`

```typescript
// src/store/useFormStore.ts

import create from "zustand";

interface FormState {
  formData: { [key: string]: any };
  setFormData: (category: string, fieldName: string, value: any) => void;
  resetFormData: () => void;
}

export const useFormStore = create<FormState>((set) => ({
  formData: {},
  setFormData: (category, fieldName, value) => set((state) => ({
    formData: { ...state.formData, [category]: { ...state.formData[category], [fieldName]: value } }
  })),
  resetFormData: () => set({ formData: {} })
}));
```

This allows you to manage form data per category in the `formData` object, so each form's data is stored independently based on its category.

---

4. **Dynamic Form Rendering**

When rendering a form, you can now fetch the correct configuration using the `formConfigFactory` and update the Zustand state for form data.

`DynamicForm.tsx`

```typescript
// src/components/form/DynamicForm.tsx

import React from "react";
import { useDynamicForm } from "../../hooks/useDynamicForm";
import { formConfigFactory } from "../../utils/formConfigFactory";
import { Button } from "../ui/Button";
import { useFormStore } from "../../store/useFormStore";
import { FieldRenderer } from "./FieldRenderer";

const DynamicForm = ({ category }) => {
  const config = formConfigFactory(category);
  const { control, handleSubmit, errors } = useDynamicForm(config.category);
  const { formData, setFormData } = useFormStore();

  const handleFieldChange = (name, value) => {
    setFormData(category, name, value); // Update the global store with category-based data
  };

  return (
    <form onSubmit={handleSubmit}>
      <h1>{config.category} Form</h1>
      {config.fields.map((fieldConfig) => (
        <div key={fieldConfig.name}>
          <label>{fieldConfig.label}</label>
          <FieldRenderer
            fieldConfig={fieldConfig}
            control={control}
            errors={errors}
            value={formData[category]?.[fieldConfig.name]}
            onChange={handleFieldChange}
          />
        </div>
      ))}
      <Button type="submit">Submit</Button>
    </form>
  );
};

export default DynamicForm;
```

Advantages of This Approach:

1. **Centralized Management**: All configurations are gathered in a single folder, making it easier to find and manage them.
2. **Scalability**: You can easily add new categories and forms by simply adding a new file inside the `formConfigs/` folder and updating the `formConfigFactory`.
3. **Separation of Concerns**: Each form configuration is modular and can be maintained independently of the others.
4. **Easy Dynamic Form Handling**: You can pass category-based configurations dynamically and handle form state in a centralized store (Zustand), reducing boilerplate and improving maintainability.

Conclusion

By organizing the form configurations into a single folder but using modular files for each category, you can maintain a clean and scalable structure. The centralization of the state and configuration makes it easy to manage different forms, add new ones, and ensure that the logic remains consistent across the app.

Let me know if you'd like further assistance or have more questions!