Exercise 1: Basic JavaScript and TypeScript

Example Code may not work for all examples, just a reference point

JavaScript Basics

1. Create a function to add two numbers.
2. Write a function that returns the reverse of a string.
3. Write a function to filter out even numbers from an array.

TypeScript Basics

1. Convert the above JavaScript functions to TypeScript.
2. Define an interface for representing a user with `id`, `name`, and `email`.
3. Create a function that takes an array of users and returns the names of users.

Mock Payload:

```
[
    { "id": 1, "name": "John Doe", "email": "john@example.com" },
    { "id": 2, "name": "Jane Smith", "email": "jane@example.com" }
]
```

Example Code:

```
function add(a: number, b: number): number {
    return a + b;
}

function reverseString(str: string): string {
    return str.split('').reverse().join('');
}

interface User {
    id: number;
    name: string;
    email: string;
}

function getUserNames(users: User[]): string[] {
    return users.map(user => user.name);
}
```

Exercise 2: React Functional Components

Stateless Component

1. Create a simple stateless component that displays a welcome message.
2. Define the props interface for this component.

Stateful Component

1. Create a stateful component that has a button and a counter. The counter should increment when the button is clicked.

TypeScript with React

1. Add TypeScript interfaces to the above components.

Example Code:

```
import React, { useState } from 'react';

// Stateless Component
interface WelcomeProps {
    name: string;
}

const Welcome: React.FC<WelcomeProps> = ({ name }) => {
    return <h1>Welcome, {name}!</h1>;
};

// Stateful Component
const Counter: React.FC = () => {
    const [count, setCount] = useState<number>(0);
    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>Increment</button>
        </div>
    );
};

export { Welcome, Counter };
```

Exercise 3: Advanced React - Fetching Data

Mock API Call

1. Simulate making a GET request to fetch user data and display it in a list.

UseEffect for Side Effects

1. Use `useEffect` to fetch data when the component mounts.

Handling Async Data

1. Display a loading state while fetching data and handle potential errors.

Mock Payload:

```json
[
    { "id": 1, "name": "John Doe", "email": "john@example.com" },
    { "id": 2, "name": "Jane Smith", "email": "jane@example.com" }
]
```

Example Code:

```tsx
import React, { useState, useEffect } from 'react';

interface User {
    id: number;
    name: string;
    email: string;
}

const UserList: React.FC = () => {
    const [users, setUsers] = useState<User[]>([]);
    const [loading, setLoading] = useState<boolean>(true);
    const [error, setError] = useState<string | null>(null);

    useEffect(() => {
        async function fetchData() {
            try {
                const response = await fetch('/api/users'); // Mock endpoint
                const data: User[] = await response.json();
                setUsers(data);
            } catch (err) {
                setError('Failed to fetch users');
            } finally {
                setLoading(false);
            }
        }
        fetchData();
    }, []);

    if (loading) return <p>Loading...</p>;
    if (error) return <p>{error}</p>;

    return (
        <ul>
            {users.map(user => (
                <li key={user.id}>{user.name} - {user.email}</li>
            ))}
        </ul>
    );
};

export default UserList;
```

Exercise 4: Styling with Tailwind CSS

Setup Tailwind CSS

1. Install Tailwind CSS in your project.
2. Configure it correctly for use with Next.js.

Styling Components

1. Style the `UserList` and `Counter` components using Tailwind CSS.

Example Code:

```
// tailwind.config.js
module.exports = {
    purge: ['./pages/**/*.{js,ts,jsx,tsx}', './components/**/*.{js,ts,jsx,tsx}'],
    darkMode: false, // or 'media' or 'class'
    theme: {
        extend: {},
    },
    variants: {
        extend: {},
    },
    plugins: [],
};

// Example Usage in Component
import React from 'react';

const StyledButton: React.FC = () => {
    return (
        <button className="bg-blue-500 text-white font-bold py-2 px-4 rounded">
            Tailwind Button
        </button>
    );
};

export default StyledButton;
```

Exercise 5: Advanced TypeScript and React Patterns

Higher Order Components (HOC)

1. Create an HOC to add a title to any component.

Context API

1. Use React's Context API to manage and provide user authentication state throughout the app.

Example Code:

```
import React, { createContext, useContext, useState } from 'react';

// HOC Example
const withTitle = (Component: React.ComponentType<any>, title: string) => (props: any)
=> (
    <>
        <h1>{title}</h1>
        <Component {...props} />
    </>
);

// Usage:
const HelloWithTitle = withTitle(Hello, 'Hello title');

// Context API Example
interface AuthContextType {
    user: string | null;
    setUser: (user: string | null) => void;
}

const AuthContext = createContext<AuthContextType | undefined>(undefined);

const AuthProvider: React.FC = ({ children }) => {
    const [user, setUser] = useState<string | null>(null);
    const value = { user, setUser };
    return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>;
};

const useAuth = () => {
    const context = useContext(AuthContext);
    if (context === undefined) {
        throw new Error('useAuth must be used within an AuthProvider');
    }
    return context;
};

export { AuthProvider, useAuth };
```

Exercise 6: Performance Optimization

Memoization

1. Use `React.memo` to optimize a list rendering by preventing unnecessary re-renders.

UseMemo and UseCallback

1. Use `useMemo` to memoize expensive computations.
2. Use `useCallback` to memoize callback functions.

Example Code:

```tsx
import React, { useState, useMemo, useCallback, memo } from 'react';

// Expensive computation example
const computeExpensiveValue = (num: number) => {
    console.log('Computing...');
    return num * 2;
};

interface ListProps {
    items: string[];
}

const List: React.FC<ListProps> = memo(({ items }) => {
    console.log('List re-rendered');
    return (
        <ul>
            {items.map((item, index) => (
                <li key={index}>{item}</li>
            ))}
        </ul>
    );
});

const PerformanceComponent: React.FC = () => {
    const [num, setNum] = useState<number>(0);
    const [inputValue, setInputValue] = useState<string>('');
    const [items, setItems] = useState<string[]>([]);

    const expensiveValue = useMemo(() => computeExpensiveValue(num), [num]);

    const addItem = useCallback(() => {
        setItems([...items, inputValue]);
    }, [items, inputValue]);

    return (
        <div>
            <h1>Expensive Value: {expensiveValue}</h1>
            <input
                type="text"
                value={inputValue}
                onChange={(e) => setInputValue(e.target.value)}
            />
            <button onClick={addItem}>Add Item</button>
            <List items={items} />
        </div>
    );
};

export default PerformanceComponent;
```

Exercise 7: Error Boundary

Create an Error Boundary

1. Implement an error boundary to catch and display errors from the child components.

Example Code:

```
import React, { Component, ErrorInfo } from 'react';

interface ErrorBoundaryState {
    hasError: boolean;
}

class ErrorBoundary extends Component<{}, ErrorBoundaryState> {
    constructor(props: {}) {
        super(props);
        this.state = { hasError: false };
    }

    static getDerivedStateFromError(_: Error) {
        return { hasError: true };
    }

    componentDidCatch(error: Error, errorInfo: ErrorInfo) {
        console.error('Error caught by ErrorBoundary:', error, errorInfo);
    }

    render() {
        if (this.state.hasError) {
            return <h1>Something went wrong.</h1>;
        }
        return this.props.children;
    }
}

export default ErrorBoundary;
```

Usage:

```
// Usage
import React from 'react';
import ErrorBoundary from './ErrorBoundary';
import SomeComponent from './SomeComponent';

const App: React.FC = () => (
    <ErrorBoundary>
        <SomeComponent />
    </ErrorBoundary>
);
```

```
export default App;
```

Exercise 8: Custom Hooks

Create Custom Hooks

1. Create a custom hook to manage form inputs.
2. Create a custom hook for fetching data.

Example Code:

```
import { useState, useEffect } from 'react';

// useForm Hook
export const useForm = <T extends Record<string, any>>(initialValues: T) => {
    const [values, setValues] = useState<T>(initialValues);

    const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
        setValues({
            ...values,
            [e.target.name]: e.target.value,
        });
    };

    return { values, handleChange };
};

// useFetch Hook
export const useFetch = <T extends any>(url: string) => {
    const [data, setData] = useState<T | null>(null);
    const [loading, setLoading] = useState<boolean>(true);
    const [error, setError] = useState<string | null>(null);

    useEffect(() => {
        const fetchData = async () => {
            try {
                const response = await fetch(url);
                const data: T = await response.json();
                setData(data);
            } catch (err) {
                setError('Failed to fetch data');
            } finally {
                setLoading(false);
            }
        };

        fetchData();
    }, [url]);
    return { data, loading, error };
};
```

Exercise 9: Redux for State Management

Install Redux and Redux Toolkit

1. Set up a Redux store using Redux Toolkit.
2. Create reducers and actions for managing user authentication state.

Example Code:

```
import { configureStore, createSlice, PayloadAction } from '@reduxjs/toolkit';

// Auth Slice
interface AuthState {
    user: string | null;
}

const initialState: AuthState = {
    user: null,
};

const authSlice = createSlice({
    name: 'auth',
    initialState,
    reducers: {
        login(state, action: PayloadAction<string>) {
            state.user = action.payload;
        },
        logout(state) {
            state.user = null;
        },
    },
});

export const { login, logout } = authSlice.actions;
export const authReducer = authSlice.reducer;

// Store
const store = configureStore({
    reducer: {
        auth: authReducer,
    },
});

export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;
export default store;
```

Usage in a component:

```
// Usage
import React from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { RootState, login, logout } from './store';

const AuthComponent: React.FC = () => {
    const dispatch = useDispatch();
    const user = useSelector((state: RootState) => state.auth.user);

    const handleLogin = () => {
        dispatch(login('User123'));
    };

    const handleLogout = () => {
        dispatch(logout());
    };

    return (
        <div>
            {user ? (
                <>
                    <p>Logged in as {user}</p>
                    <button onClick={handleLogout}>Logout</button>
                </>
            ) : (
                <button onClick={handleLogin}>Login</button>
            )}
        </div>
    );
};

export default AuthComponent;
```

Exercise 10: Integrate with an External API

Fetch Data from an External API

1. Use an external API service (e.g., OpenWeatherMap) to fetch and display data.

Example Code:

```
import React, { useState, useEffect } from 'react';

interface WeatherData {
    main: {
        temp: number;
    };
    weather: {
        description: string;
    }[];
}
```

```
const WeatherComponent: React.FC = () => {
    const [weatherData, setWeatherData] = useState<WeatherData | null>(null);
    const [loading, setLoading] = useState<boolean>(true);
    const [error, setError] = useState<string | null>(null);

    useEffect(() => {
        const fetchWeather = async () => {
            try {
                const response = await fetch(
                    'https://api.openweathermap.org/data/2.5/weather?
q=London&appid=YOUR_API_KEY'
                );
                const data: WeatherData = await response.json();
                setWeatherData(data);
            } catch (err) {
                setError('Failed to fetch weather data');
            } finally {
                setLoading(false);
            }
        };

        fetchWeather();
    }, []);

    if (loading) return <p>Loading...</p>;
    if (error) return <p>{error}</p>;
    if (!weatherData) return null;

    return (
        <div>
            <h1>Weather in London</h1>
            <p>Temperature: {weatherData?.main.temp}°C</p>
            <p>Description: {weatherData?.weather[0].description}</p>
        </div>
    );
};

export default WeatherComponent;
```

Exercise 11: Tests with Jest and React Testing Library

Write Unit Tests

1. Write unit tests for functional components and hooks.
2. Write tests for async data fetching.

Example Code:

```
// Example test for Component
import React from 'react';
import { render, screen } from '@testing-library/react';
import '@testing-library/jest-dom/extend-expect';
import UserComponent from './UserComponent'; // Assume this component exists

test('renders the user component with correct name', () => {
    render(<UserComponent name="John Doe" />);
    const nameElement = screen.getByText(/john doe/i);
    expect(nameElement).toBeInTheDocument();
});
```