Master Composable Functions

Master Composable Functions

React, React Native, TypeScript & Zustand Complete Guide

Table of Contents

---

1. What Are Composable Functions? {#what-are-composable-functions}

Definition

Composable functions are small, pure functions that can be easily combined to create more complex functionality. They follow the principle: **"Do one thing well and be easily combinable."**

Core Principles

```
// ❌ Non-composable: Does too many things
function processUserDataAndSendEmail(userData: any) {
  const cleaned = cleanData(userData);
  const validated = validateData(cleaned);
  const saved = saveToDatabase(validated);
  sendWelcomeEmail(saved.email);
  return saved;
}

// ✅ Composable: Small, focused functions
const cleanData = (data: any) => ({ ...data, name:
data.name.trim() });
const validateData = (data: any) => data.email ? data :
null;
const saveToDatabase = (data: any) => ({ ...data, id:
Date.now() });
const sendWelcomeEmail = (email: string) =>
console.log(`Email sent to ${email}`);

// Compose them together
const processUser = (userData: any) => {
  const result = [cleanData, validateData, saveToDatabase]
    .reduce((acc, fn) => fn(acc), userData);

  if (result) sendWelcomeEmail(result.email);
  return result;
};
```

Function Composition vs Class Components

```
// ❌ Class Component Approach
class UserProfile extends Component {
  state = { user: null, loading: false };

  componentDidMount() {
    this.fetchUser();
```

```
  }

  fetchUser = async () => {
    this.setState({ loading: true });
    const user = await api.getUser(this.props.userId);
    this.setState({ user, loading: false });
  }

  render() {
    return <div>{this.state.loading ? 'Loading...' :
this.state.user?.name}</div>;
  }
}

// ✅ Composable Function Approach
const fetchUser = async (userId: string) => {
  const response = await api.getUser(userId);
  return response;
};

const useUserData = (userId: string) => {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    setLoading(true);
    fetchUser(userId)
      .then(setUser)
      .finally(() => setLoading(false));
  }, [userId]);

  return { user, loading };
};

const UserProfile = ({ userId }: { userId: string }) => {
  const { user, loading } = useUserData(userId);
```

```
  return <div>{loading ? 'Loading...' : user?.name}</div>;
};
```

---

2. TypeScript for Function Composition {#typescript-composition}

Syntax Pattern: Generic Function Types

```
// Basic composition types - these define the "shape" of
composable functions
type Fn<T, U> = (arg: T) => U;  // Function that takes T,
returns U
type Compose = <T, U, V>(f: Fn<U, V>, g: Fn<T, U>) =>
Fn<T, V>;

// Implementation follows the type signature
const compose: Compose = (f, g) => (x) => f(g(x));
```

Key Syntax Elements:

- `<T, U, V>` - Generic type parameters for flexibility
- `Fn<T, U>` - Reusable function type definition
- `(f, g) => (x) => f(g(x))` - Curried function returning another function

Syntax Pattern: Curry with Proper Typing

```
// This complex type ensures proper typing through the
curry chain
type Curry<T extends any[], R> = T extends [infer H,
```

```
  ...infer Rest]
    ? (arg: H) => Rest extends [] ? R : Curry<Rest, R>
    : R;
```

Syntax Breakdown:

- `T extends any[]` - T must be an array type (function parameters)
- `[infer H, ...infer Rest]` - Destructure first parameter (H) from rest
- `Rest extends []` - Check if remaining parameters are empty
- Recursive type definition creates the curry chain

---

3. Basic Function Composition Patterns {#basic-patterns}

Syntax Pattern: Result Type for Error Handling

```
// Union type pattern for composable error handling
type Result<T, E = Error> =
  | { success: true; data: T }
  | { success: false; error: E };

// Helper functions follow consistent patterns
const ok = <T>(data: T): Result<T> => ({ success: true,
data });
const err = <E>(error: E): Result<never, E> => ({ success:
false, error });
```

Key Syntax Concepts:

- Union types `|` for either/or scenarios
- Discriminated unions with `success` property
- Generic defaults `E = Error`
- `Result<never, E>` - never type for error case

Syntax Pattern: Function Factories

```
// Higher-order function pattern - function that returns a
function
const createValidator = <T>(predicate: (value: T) =>
boolean, errorMsg: string) =>
  (value: T): Result<T> =>
    predicate(value) ? ok(value) : err(new
Error(errorMsg));
```

Syntax Elements:

- `<T>` - Generic for input type
- `predicate: (value: T) => boolean` - Function parameter
- Returns `(value: T): Result<T>` - Another function
- Ternary operator for conditional logic

---

4. React Composable Functions {#react-composable}

Syntax Pattern: Higher-Order Components (HOCs)

```
// HOC pattern - component that wraps another component
const withLoading = <P extends object>(
  Component: React.ComponentType<P>
```

```
) => (props: P & { loading?: boolean }) => {
  const { loading, ...restProps } = props;

  if (loading) return <div>Loading...</div>;
  return <Component {...(restProps as P)} />;
};
```

Syntax Breakdown:

- `<P extends object>` - Generic constrained to objects
- `React.ComponentType<P>` - React component type
- `P & { loading?: boolean }` - Intersection type (P plus loading)
- `{ loading, ...restProps }` - Destructuring with rest operator
- `{...(restProps as P)}` - Spread with type assertion

Syntax Pattern: Render Function Composition

```
// Function composition for render logic
type RenderFunction<T> = (props: T) => React.ReactNode;

const composeRenders = <T>(...renders: RenderFunction<T>
[]): RenderFunction<T> =>
  (props: T) => renders.map((render, index) => (
    <React.Fragment key={index}>{render(props)}
</React.Fragment>
  ));
```

Key Patterns:

- `(...renders: RenderFunction<T>[])` - Rest parameters with array type
- `React.ReactNode` - Type for any renderable content
- `.map()` with JSX elements

- `React.Fragment` for multiple elements without wrapper

---

5. React Native Composable Functions {#react-native-composable}

Syntax Pattern: Platform-Specific Composition

```
const createPlatformComposer = <T>(
  iosValue: T,
  androidValue: T,
  webValue?: T
): T => {
  if (Platform.OS === 'ios') return iosValue;
  if (Platform.OS === 'android') return androidValue;
  if (Platform.OS === 'web' && webValue) return webValue;
  return androidValue; // fallback
};
```

Syntax Elements:

- Optional parameters with `?`
- Platform.OS comparison
- Fallback logic with default return
- Generic `<T>` ensures all values same type

Syntax Pattern: Animation Composition

```
// Function composition for animations
const createAnimation = (value: Animated.Value, toValue:
number, duration = 300) =>
```

```
  Animated.timing(value, { toValue, duration,
useNativeDriver: true });

const createSequence = (...animations:
Animated.CompositeAnimation[]) =>
  Animated.sequence(animations);
```

Key Concepts:

- Default parameters `duration = 300`
- Rest parameters for variable arguments
- Object shorthand `{ toValue, duration }`
- Function returns another function call

---

6. Zustand Composable Functions {#zustand-composable}

Syntax Pattern: Store Slice Composition

```
// Interface pattern for store slices
interface UserSlice {
  user: User | null;
  setUser: (user: User | null) => void;
  clearUser: () => void;
}

// Slice creator pattern
const createUserSlice = (set: any): UserSlice => ({
  user: null,
  setUser: (user) => set({ user }),
  clearUser: () => set({ user: null }),
});
```

Syntax Patterns:

- Interface for type safety
- Function that takes `set` and returns slice object
- Arrow functions for concise methods
- Implicit returns with object shorthand

Syntax Pattern: Generic Slice Composer

```
type SliceCreator<T> = (set: any, get: any) => T;

const composeSlices = <T extends Record<string, any>>(
  ...sliceCreators: SliceCreator<any>[]
) => (set: any, get: any): T =>
  sliceCreators.reduce(
    (acc, createSlice) => ({ ...acc, ...createSlice(set,
get) }),
    {} as T
  );
```

Advanced Syntax:

- `Record<string, any>` - Object with string keys
- Type constraint `extends Record<string, any>`
- `.reduce()` with spread operator for object merging
- Type assertion `{} as T`

---

7. Advanced Composition Techniques {#advanced-techniques}

Async Action Composition Pattern

```
const createAsyncActions = <T, U>(
  name: string,
  asyncFn: (params: T) => Promise<U>
) => (set: any, get: any) => ({
  [`${name}Loading`]: false,
  [`${name}Error`]: null,
  [`${name}Data`]: null,

  [name]: async (params: T) => {
    set({ [`${name}Loading`]: true, [`${name}Error`]: null
});
    try {
      const data = await asyncFn(params);
      set({ [`${name}Data`]: data, [`${name}Loading`]:
false });
      return data;
    } catch (error) {
      set({ [`${name}Error`]: error.message,
[`${name}Loading`]: false });
      throw error;
    }
  },
});
```

Advanced Syntax Elements:

- Computed property names `[name]` and `[\ ${name}Loading ]`
- Template literals for dynamic keys
- Async/await pattern
- Try/catch with state updates
- Promise return types

Computed Values with Object.defineProperty

```
const createComputedSlice = <T extends Record<string,
any>>(
  computedValues: Record<string, (state: T) => any>
) => (set: any, get: any) => {
  const computed = {} as Record<string, any>;

  Object.keys(computedValues).forEach(key => {
    Object.defineProperty(computed, key, {
      get: () => computedValues[key](get()),
      enumerable: true,
    });
  });

  return computed;
};
```

Key Patterns:

- `Object.defineProperty` for getter properties
- `enumerable: true` for iteration
- Function as getter value
- Dynamic property creation

---

8. Real-World Composition Examples {#real-world-examples}

Complete Todo App Pattern

```
// Combine multiple composition patterns
type TodoStore = TodoSlice & ReturnType<typeof
createTodoComputedSlice>;
```

```
const useTodoStore = create<TodoStore>()(
  devtools(
    (set, get) => ({
      ...createTodoSlice(set),
      ...createTodoComputedSlice(set, get),
    })
  )
);
```

Syntax Concepts:

- `ReturnType<typeof function>` - Infer return type
- Multiple spread operators for object composition
- Middleware wrapping pattern
- Type intersection with `&`

---

9. Best Practices & Patterns {#best-practices}

Key Syntax Patterns to Master:

1. **Generic Constraints:**

```
<T extends SomeType>  // Limit generic to specific types
```

2. **Function Composition:**

```
const composed = (...fns) => (input) =>
fns.reduce((acc, fn) => fn(acc), input);
```

3. **Conditional Types:**

```
type Result<T> = T extends string ? string[] : T[];
```

4. **Utility Types:**

```
Partial<T>    // All properties optional
Pick<T, K>    // Select specific properties
Omit<T, K>    // Exclude specific properties
```

5. **Union and Intersection:**

```
type Union = A | B;         // Either A or B
type Intersection = A & B; // Both A and B
```

Composition Principles:

1. **Single Responsibility:** Each function does one thing well
2. **Pure Functions:** No side effects, predictable outputs
3. **Type Safety:** Use TypeScript generics and constraints
4. **Composability:** Functions can be easily combined
5. **Immutability:** Don't mutate inputs, return new values

Common Patterns:

- **Pipeline:** `pipe(fn1, fn2, fn3)(input)`
- **Factory:** `createFunction(config) => actualFunction`
- **HOC:** `withFeature(Component) => EnhancedComponent`
- **Slice:** `createSlice(set, get) => sliceObject`
- **Middleware:** `middleware(config) => enhancedConfig`

This syntax-focused approach helps you understand the building blocks to create any composable function pattern you need!

React, React Native, TypeScript & Zustand Complete Guide

Table of Contents

---

1. What Are Composable Functions? {#what-are-composable-functions}

Definition

Composable functions are small, pure functions that can be easily combined to create more complex functionality. They follow the principle: **"Do one thing well and be easily combinable."**

Core Principles

```
// ❌ Non-composable: Does too many things
function processUserDataAndSendEmail(userData: any) {
  const cleaned = cleanData(userData);
```

```
  const validated = validateData(cleaned);
  const saved = saveToDatabase(validated);
  sendWelcomeEmail(saved.email);
  return saved;
}

// ✅ Composable: Small, focused functions
const cleanData = (data: any) => ({ ...data, name:
data.name.trim() });
const validateData = (data: any) => data.email ? data :
null;
const saveToDatabase = (data: any) => ({ ...data, id:
Date.now() });
const sendWelcomeEmail = (email: string) =>
console.log(`Email sent to ${email}`);

// Compose them together
const processUser = (userData: any) => {
  const result = [cleanData, validateData, saveToDatabase]
    .reduce((acc, fn) => fn(acc), userData);

  if (result) sendWelcomeEmail(result.email);
  return result;
};
```

Function Composition vs Class Components

```
// ❌ Class Component Approach
class UserProfile extends Component {
  state = { user: null, loading: false };

  componentDidMount() {
    this.fetchUser();
  }

  fetchUser = async () => {
```

```
    this.setState({ loading: true });
    const user = await api.getUser(this.props.userId);
    this.setState({ user, loading: false });
  }

  render() {
    return <div>{this.state.loading ? 'Loading...' :
this.state.user?.name}</div>;
  }
}

// ✅ Composable Function Approach
const fetchUser = async (userId: string) => {
  const response = await api.getUser(userId);
  return response;
};

const useUserData = (userId: string) => {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    setLoading(true);
    fetchUser(userId)
      .then(setUser)
      .finally(() => setLoading(false));
  }, [userId]);

  return { user, loading };
};

const UserProfile = ({ userId }: { userId: string }) => {
  const { user, loading } = useUserData(userId);
  return <div>{loading ? 'Loading...' : user?.name}</div>;
};
```

2. TypeScript for Function Composition {#typescript-composition}

Syntax Pattern: Generic Function Types

```
// Basic composition types - these define the "shape" of
composable functions
type Fn<T, U> = (arg: T) => U;  // Function that takes T,
returns U
type Compose = <T, U, V>(f: Fn<U, V>, g: Fn<T, U>) =>
Fn<T, V>;

// Implementation follows the type signature
const compose: Compose = (f, g) => (x) => f(g(x));
```

Key Syntax Elements:

- `<T, U, V>` - Generic type parameters for flexibility
- `Fn<T, U>` - Reusable function type definition
- `(f, g) => (x) => f(g(x))` - Curried function returning another function

Syntax Pattern: Curry with Proper Typing

```
// This complex type ensures proper typing through the
curry chain
type Curry<T extends any[], R> = T extends [infer H,
...infer Rest]
  ? (arg: H) => Rest extends [] ? R : Curry<Rest, R>
  : R;
```

Syntax Breakdown:

- `T extends any[]` - T must be an array type (function parameters)
- `[infer H, ...infer Rest]` - Destructure first parameter (H) from rest
- `Rest extends []` - Check if remaining parameters are empty
- Recursive type definition creates the curry chain

---

3. Basic Function Composition Patterns {#basic-patterns}

Syntax Pattern: Result Type for Error Handling

```
// Union type pattern for composable error handling
type Result<T, E = Error> =
  | { success: true; data: T }
  | { success: false; error: E };

// Helper functions follow consistent patterns
const ok = <T>(data: T): Result<T> => ({ success: true,
data });
const err = <E>(error: E): Result<never, E> => ({ success:
false, error });
```

Key Syntax Concepts:

- Union types `|` for either/or scenarios
- Discriminated unions with `success` property
- Generic defaults `E = Error`
- `Result<never, E>` - never type for error case

Syntax Pattern: Function Factories

```
// Higher-order function pattern - function that returns a
function
const createValidator = <T>(predicate: (value: T) =>
boolean, errorMsg: string) =>
  (value: T): Result<T> =>
    predicate(value) ? ok(value) : err(new
Error(errorMsg));
```

Syntax Elements:

- `<T>` - Generic for input type
- `predicate: (value: T) => boolean` - Function parameter
- Returns `(value: T): Result<T>` - Another function
- Ternary operator for conditional logic

---

4. React Composable Functions {#react-composable}

Syntax Pattern: Higher-Order Components (HOCs)

```
// HOC pattern - component that wraps another component
const withLoading = <P extends object>(
  Component: React.ComponentType<P>
) => (props: P & { loading?: boolean }) => {
  const { loading, ...restProps } = props;

  if (loading) return <div>Loading...</div>;
  return <Component {...(restProps as P)} />;
};
```

Syntax Breakdown:

- `<P extends object>` - Generic constrained to objects
- `React.ComponentType<P>` - React component type
- `P & { loading?: boolean }` - Intersection type (P plus loading)
- `{ loading, ...restProps }` - Destructuring with rest operator
- `{...(restProps as P)}` - Spread with type assertion

Syntax Pattern: Render Function Composition

```
// Function composition for render logic
type RenderFunction<T> = (props: T) => React.ReactNode;

const composeRenders = <T>(...renders: RenderFunction<T>
[]): RenderFunction<T> =>
  (props: T) => renders.map((render, index) => (
    <React.Fragment key={index}>{render(props)}
</React.Fragment>
  ));
```

Key Patterns:

- `(...renders: RenderFunction<T>[])` - Rest parameters with array type
- `React.ReactNode` - Type for any renderable content
- `.map()` with JSX elements
- `React.Fragment` for multiple elements without wrapper

---

5. React Native Composable Functions {#react-native-composable}

Syntax Pattern: Platform-Specific Composition

```
const createPlatformComposer = <T>(
  iosValue: T,
  androidValue: T,
  webValue?: T
): T => {
  if (Platform.OS === 'ios') return iosValue;
  if (Platform.OS === 'android') return androidValue;
  if (Platform.OS === 'web' && webValue) return webValue;
  return androidValue; // fallback
};
```

Syntax Elements:

- Optional parameters with `?`
- Platform.OS comparison
- Fallback logic with default return
- Generic `<T>` ensures all values same type

Syntax Pattern: Animation Composition

```
// Function composition for animations
const createAnimation = (value: Animated.Value, toValue:
number, duration = 300) =>
  Animated.timing(value, { toValue, duration,
useNativeDriver: true });

const createSequence = (...animations:
Animated.CompositeAnimation[]) =>
  Animated.sequence(animations);
```

Key Concepts:

- Default parameters `duration = 300`
- Rest parameters for variable arguments
- Object shorthand `{ toValue, duration }`
- Function returns another function call

---

6. Zustand Composable Functions {#zustand-composable}

Syntax Pattern: Store Slice Composition

```
// Interface pattern for store slices
interface UserSlice {
  user: User | null;
  setUser: (user: User | null) => void;
  clearUser: () => void;
}

// Slice creator pattern
const createUserSlice = (set: any): UserSlice => ({
  user: null,
  setUser: (user) => set({ user }),
  clearUser: () => set({ user: null }),
});
```

Syntax Patterns:

- Interface for type safety
- Function that takes `set` and returns slice object
- Arrow functions for concise methods
- Implicit returns with object shorthand

Syntax Pattern: Generic Slice Composer

```
type SliceCreator<T> = (set: any, get: any) => T;

const composeSlices = <T extends Record<string, any>>(
  ...sliceCreators: SliceCreator<any>[]
) => (set: any, get: any): T =>
  sliceCreators.reduce(
    (acc, createSlice) => ({ ...acc, ...createSlice(set,
get) }),
    {} as T
  );
```

Advanced Syntax:

- `Record<string, any>` - Object with string keys
- Type constraint `extends Record<string, any>`
- `.reduce()` with spread operator for object merging
- Type assertion `{} as T`

---

7. Advanced Composition Techniques {#advanced-techniques}

Async Action Composition Pattern

```
const createAsyncActions = <T, U>(
  name: string,
  asyncFn: (params: T) => Promise<U>
) => (set: any, get: any) => ({
  [`${name}Loading`]: false,
  [`${name}Error`]: null,
  [`${name}Data`]: null,
```

```
  [name]: async (params: T) => {
    set({ [`${name}Loading`]: true, [`${name}Error`]: null
});
    try {
      const data = await asyncFn(params);
      set({ [`${name}Data`]: data, [`${name}Loading`]:
false });
      return data;
    } catch (error) {
      set({ [`${name}Error`]: error.message,
[`${name}Loading`]: false });
      throw error;
    }
  },
});
```

Advanced Syntax Elements:

- Computed property names `[name]` and `[\ ${name}Loading ]`
- Template literals for dynamic keys
- Async/await pattern
- Try/catch with state updates
- Promise return types

Computed Values with Object.defineProperty

```
const createComputedSlice = <T extends Record<string,
any>>(
  computedValues: Record<string, (state: T) => any>
) => (set: any, get: any) => {
  const computed = {} as Record<string, any>;

  Object.keys(computedValues).forEach(key => {
```

```
    Object.defineProperty(computed, key, {
      get: () => computedValues[key](get()),
      enumerable: true,
    });
  });

  return computed;
};
```

Key Patterns:

- `Object.defineProperty` for getter properties
- `enumerable: true` for iteration
- Function as getter value
- Dynamic property creation

---

8. Real-World Composition Examples {#real-world-examples}

Complete Todo App Pattern

```
// Combine multiple composition patterns
type TodoStore = TodoSlice & ReturnType<typeof
createTodoComputedSlice>;

const useTodoStore = create<TodoStore>()(
  devtools(
    (set, get) => ({
      ...createTodoSlice(set),
      ...createTodoComputedSlice(set, get),
    })
```

```
  )
);
```

Syntax Concepts:

- `ReturnType<typeof function>` - Infer return type
- Multiple spread operators for object composition
- Middleware wrapping pattern
- Type intersection with `&`

---

9. Best Practices & Patterns {#best-practices}

Key Syntax Patterns to Master:

1. **Generic Constraints:**

```
<T extends SomeType>  // Limit generic to specific types
```

2. **Function Composition:**

```
const composed = (...fns) => (input) => fns.reduce((acc, fn) => fn(acc), input);
```

3. **Conditional Types:**

```
type Result<T> = T extends string ? string[] : T[];
```

4. **Utility Types:**

```
Partial<T>    // All properties optional
Pick<T, K>    // Select specific properties
Omit<T, K>    // Exclude specific properties
```

5. **Union and Intersection:**

```
type Union = A | B;         // Either A or B
type Intersection = A & B; // Both A and B
```

Composition Principles:

1. **Single Responsibility:** Each function does one thing well
2. **Pure Functions:** No side effects, predictable outputs
3. **Type Safety:** Use TypeScript generics and constraints
4. **Composability:** Functions can be easily combined
5. **Immutability:** Don't mutate inputs, return new values

Common Patterns:

- **Pipeline:** `pipe(fn1, fn2, fn3)(input)`
- **Factory:** `createFunction(config) => actualFunction`
- **HOC:** `withFeature(Component) => EnhancedComponent`
- **Slice:** `createSlice(set, get) => sliceObject`
- **Middleware:** `middleware(config) => enhancedConfig`

This syntax-focused approach helps you understand the building blocks to create any composable function pattern you need!