Ανάπτυξη Κινητών Εφαρμογών

Εργαστήριο

Δημήτρης Ρίγγας

riggas@ionio.gr

Τμήμα Πληροφορικής

Ιόνιο Πανεπιστήμιο

Εργαστήριο 3 – React Native Development

State Management & Navigation

Hooks, Context API, Navigation

Στόχος:

- ✓ **Component-based Architecture** στο React Native
- ✓ **React Hooks** - Εισαγωγή και Φιλοσοφία
- ✓ **useState Hook** - Διαχείριση Local State
- ✓ **useEffect Hook** - Side Effects και Lifecycle
- ✓ **Prop Drilling Problem** και λύσεις
- ✓ **Context API** - Global State Management
- ✓ **useContext Hook** - Πρακτική Αξιοποίηση
- ✓ **useReducer Hook** - Complex State Logic
- ✓ **React Navigation** - Alternatives & Strategies
- ✓ **Best Practices** και Patterns

Component-based Architecture

Τι είναι Components;

Τα **Components** είναι τα δομικά στοιχεία (building blocks) κάθε React/React Native εφαρμογής. Λειτουργούν σαν ανεξάρτητες, επαναχρησιμοποιήσιμες μονάδες κώδικα που:

- **Encapsulation**: Περιέχουν τη δική τους λογική και UI
- **Reusability**: Μπορούν να χρησιοποιηθούν πολλές φορές
- **Composability**: Συνδυάζονται για να φτιάξουν πολύπλοκα UIs
- **Independence**: Λειτουργούν ανεξάρτητα το ένα από το άλλο

```
// Simple Component Example
const Welcome = ({ name }) => {
  return <Text>Γεια σου, {name}!</Text>;
};
```

Component Hierarchy

Δέντρο Components (Component Tree)

Τα components οργανώνονται σε **ιεραρχική δομή** (hierarchy):

```
App (Root)
├── Header
│   ├── Logo
│   └── Navigation
├── MainContent
│   ├── Sidebar
│   └── Content
│       ├── Article
│       └── Comments
└── Footer
```

Ροή Δεδομένων (Data Flow):

- Unidirectional (μονής κατεύθυνσης)
- Από parent → child μέσω **props**
- Immutable: Τα props δεν μπορούν να αλλάξουν από το child

State vs Props - Βασικές Έννοιες

State	Props
Εσωτερική κατάσταση component	Παράμετροι από parent
Mutable (μπορεί να αλλάξει)	Immutable (read-only)
Διαχειρίζεται εσωτερικά	Περνάει από έξω
Επηρεάζει rendering	Επηρεάζει rendering

```
// Props: Δέχεται από parent
const UserCard = ({ name, age }) => {
  // State: Εσωτερική κατάσταση
  const [isExpanded, setIsExpanded] = useState(false);

  return <View>...</View>;
};
```

React Hooks - Εισαγωγή: Τι είναι τα Hooks;

Τα **Hooks** είναι functions που μας επιτρέπουν να "hook into" React features από **functional components**:

Πριν τα Hooks (Class Components):

- State: `this.state`, `this.setState()`
- Lifecycle: `componentDidMount()`, `componentDidUpdate()`
- Πολύπλοκος κώδικας, δύσκολη επαναχρησιμοποίηση

Με Hooks (Functional Components):

- State: `useState()`
- Side Effects: `useEffect()`
- Πιο απλός, καθαρός κώδικας
- Καλύτερη επαναχρησιμοποίηση logic

React Hooks - Κανόνες Χρήσης

1. Καλέστε Hooks μόνο στο top level

2. Καλέστε Hooks μόνο από React Functions

- Functional Components ✅
- Custom Hooks ✅
  Functions that themselves use hooks and start with `use`
- Regular JavaScript functions ❌

```
// ✅ Σωστό
const MyComponent = () => {
  const [count, setCount] = useState(0);

  return <View>...</View>;
};

// ❌ Λάθος - μέσα σε if
const MyComponent = () => {
  if (condition) {
    const [count, setCount] = useState(0); // NO!
  }
};

// ❌ Λάθος - μέσα σε regular js function
function doSomething() {
  const [count, setCount] = useState(0);
}
```

useState Hook - Διαχείριση Local State

Το `useState` είναι το πιο βασικό Hook για state management.

Το useState είναι ένα React Hook που επιτρέπει σε ένα function component να θυμάται δεδομένα μεταξύ renderings.
Χωρίς αυτό, κάθε φορά που το component θα "ξαναζωγραφιζόταν", όλες οι τιμές θα χάνονταν.

Μέρος	Ρόλος
`count`	Η **μεταβλητή κατάστασης** (state variable). Κρατά την τρέχουσα τιμή.
`setCount`	Η **συνάρτηση ενημέρωσης** (setter). Αλλάζει την τιμή του state.
`useState(0)`	Η αρχικοποίηση της τιμής (`0` στην αρχή).

Κάθε φορά που καλείτε το setCount, η React:

- Αλλάζει την αποθηκευμένη τιμή του count.
- Ξανακάνει render το component.
- Εμφανίζει τη νέα τιμή στην οθόνη.

Το state δεν είναι απλό variable — είναι δεμένο με τον κύκλο ζωής του component.

```jsx
import { useState } from 'react';

const Counter = () => {
  // Δήλωση state variable
  const [count, setCount] = useState(0);
  //     ↑state  ↑setter    ↑initial value

  const increment = () => {
    setCount(count + 1); // Ενημέρωση state
  };

  return (
    <View>
      <Text>Count: {count}</Text>
      <Button title="+" onPress={increment} />
    </View>
  );
};
```

useState - Πολλαπλές State Μεταβλητές

Μπορούμε να έχουμε πολλές state μεταβλητές στο ίδιο component:

```
const UserProfile = () => {
  const [name, setName] = useState('');
  const [age, setAge] = useState(0);
  const [email, setEmail] = useState('');
  const [isActive, setIsActive] = useState(false);

  return (
    <View>
      <TextInput
        value={name}
        onChangeText={setName}
        placeholder="Όνομα"
      />
      <TextInput
        value={email}
        onChangeText={setEmail}
        placeholder="Email"
      />
      <Switch value={isActive} onValueChange={setIsActive} />
    </View>
  );
};
```

useState - State ως Object

Για σύνθετα δεδομένα, χρησιμοποιούμε objects.

Όταν χρησιμοποιούμε useState με ένα object το React κρατάει ολόκληρο το αντικείμενο user σαν μία “μονάδα” state.

Όταν καλούμε setUser() με νέο αντικείμενο, η React **αντικαθιστά ολόκληρο** το προηγούμενο state με αυτό που δίνουμε.

Αν προσπαθήσουμε να ενημερώσουμε μόνο ένα πεδίο χωρίς το spread:
`setUser({ name: newName }); //` ❌
τότε το νέο state θα είναι πχ:
`{ name: 'Maria' }`
και όλα τα άλλα πεδία (email, age, preferences) θα χαθούν, επειδή δεν μεταφέρθηκαν στο νέο object.

Η σωστή λύση: Spread operator (`...`).

Αυτό λέγεται immutable update — δηλαδή, δημιουργούμε ένα νέο object αντί να αλλάζουμε το παλιό.

⚠️ Η React αναγνωρίζει ότι το state άλλαξε μόνο αν πάρει ένα νέο αντικείμενο. Αν αλλάξουμε απευθείας την ιδιότητα (user.name = 'Maria') χωρίς να καλέσουμε setUser, το React δεν θα κάνει re-render, γιατί δεν “βλέπει” τη μεταβολή.

```
const UserProfile = () => {
  const [user, setUser] = useState({
    name: '',
    email: '',
    age: 0,
    preferences: {
      theme: 'light',
      notifications: true
    }
  });

  // !!! Πρέπει να κρατάμε τα υπόλοιπα properties
  const updateName = (newName) => {
    setUser({
      ...user,            // Spread existing properties
      name: newName       // Override specific property
    });
  };

  return <View>...</View>;
};
```

useState - Functional Updates

Όταν το νέο state εξαρτάται από το προηγούμενο, χρησιμοποιούμε **functional update**.

Το πρόβλημα των async updates είναι ότι η React δεν ενημερώνει το state αμέσως.
Αντίθετα, προγραμματίζει την ενημέρωση για το επόμενο render.
Αυτό σημαίνει ότι αν κληθεί πολλές φορές `setCount(count + 1)` στη σειρά όλες αυτές οι κλήσεις βλέπουν το ίδιο "παλιό" count.

Η λύση είναι η χρήση συνάρτησης μέσα στο setCount. Η παράμετρος prev να έχει την πιο πρόσφατη τιμή του count. Το prev δεν είναι κάποια "μαγική" λέξη· μπορεί να χρησιμοποιηθεί οποιοδήποτε όνομα παραμέτρου το οποίο απλώς αντιπροσωπεύει την τιμή *currentStateValue*.

```
const Counter = () => {
  const [count, setCount] = useState(0);

  // ❌ Πρόβλημα με async updates
  const incrementThreeTimes = () => {
    setCount(count + 1);
    setCount(count + 1);
    setCount(count + 1);
    // Result: count + 1 (not + 3!)
  };

  // ✅ Σωστό με functional update
  const incrementThreeTimesSafe = () => {
    setCount(prev => prev + 1);
    setCount(prev => prev + 1);
    setCount(prev => prev + 1);
    // Result: count + 3 ✓
  };
};
```

useEffect Hook - Side Effects και Lifecycle

Το `useEffect` χειρίζεται **side effects** όπως:

- Data fetching
- Subscriptions
- Timers
- DOM manipulations
- Logging

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

const DataFetcher = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    // Effect code - τρέχει μετά το render
    fetch('https://api.example.com/data')
      .then(res => res.json())
      .then(setData);
  }, []); // Dependency array

  return <View>...</View>;
};
```

useEffect - Dependency Array: Πότε τρέχει το Effect;

Το **dependency array** ελέγχει πότε εκτελείται το effect:

```
// 1. Κάθε render (NO dependency array)
useEffect(() => {
  console.log('Runs on every render');
});

// 2. Μόνο μια φορά (Empty array)
useEffect(() => {
  console.log('Runs once on mount');
}, []);

// 3. Όταν αλλάζουν dependencies
useEffect(() => {
  console.log('Runs when count changes');
}, [count]);

// 4. Multiple dependencies
useEffect(() => {
  console.log('Runs when count OR name changes');
}, [count, name]);
```

useEffect - Cleanup Function: Καθαρισμός Resources

Το effect μπορεί να επιστρέψει **cleanup function**.

Cleanup τρέχει:

- Πριν το επόμενο effect run
- Όταν το component unmounts

```
const Timer = () => {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    // Setup: Δημιουργία interval
    const interval = setInterval(() => {
      setSeconds(s => s + 1);
    }, 1000);

    // Cleanup: Καθαρισμός όταν το component unmounts
    return () => {
      clearInterval(interval);
      console.log('Cleanup: interval cleared');
    };
  }, []);

  return <Text>{seconds}s</Text>;
};
```

useEffect - Παράδειγμα: Data Fetching

```
const UserList = () => {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    let cancelled = false; // Για cleanup

    const fetchUsers = async () => {
      try {
        const response = await fetch('https://api.example.com/users');
        const data = await response.json();

        if (!cancelled) {
          setUsers(data);
          setLoading(false);
        }
      } catch (err) {
        if (!cancelled) {
          setError(err.message);
          setLoading(false);
        }
      }
    };

    fetchUsers();

    return () => { cancelled = true; }; // Cleanup
  }, []);

  if (loading) return <Text>Φόρτωση...</Text>;
  if (error) return <Text>Σφάλμα: {error}</Text>;
  return <FlatList data={users} ... />;
};
```

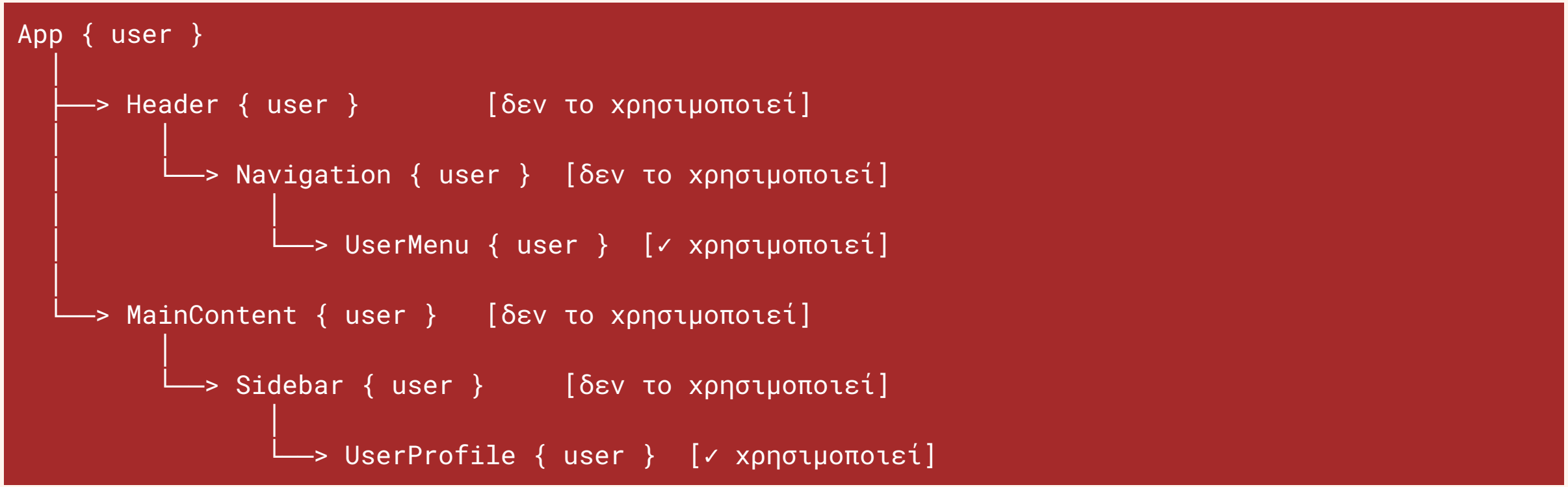
Το Πρόβλημα του Prop Drilling

Prop Drilling: Μεταφορά props μέσα από πολλά επίπεδα components

```
// ❌ Πρόβλημα: Τα MiddleComponent & IntermediateComponent
//    δεν χρειάζονται το user, απλά το περνάνε παρακάτω

const App = () => {
  const [user, setUser] = useState({ name: 'John', role: 'admin' });
  return <MiddleComponent user={user} />;
};

const MiddleComponent = ({ user }) => {
  return <IntermediateComponent user={user} />;
};

const IntermediateComponent = ({ user }) => {
  return <DeepComponent user={user} />;
};

const DeepComponent = ({ user }) => {
  return <Text>{user.name}</Text>; // Τελικά χρησιμοποιείται εδώ
};
```

Προβλήματα: Δύσκολη συντήρηση, refactoring, testing

Το Πρόβλημα του Prop Drilling: Οπτικοποίηση του Προβλήματος



Τα `user` props περνάνε από **6 components** αλλά χρησιμοποιούνται μόνο σε **2**!

Context API - Λύση στο Prop Drilling: Global State χωρίς Props Chains

Το **Context API** επιτρέπει sharing data σε όλο το component tree χωρίς props:

```
import { createContext } from 'react';

// 1. Δημιουργία Context
const UserContext = createContext(null);

// 2. Provider - Παρέχει τα data
const App = () => {
  const [user, setUser] = useState({ name: 'John', role: 'admin' });

  return (
    <UserContext.Provider value={{ user, setUser }}>
      <MiddleComponent />
    </UserContext.Provider>
  );
};

// 3. Τα intermediate components ΔΕΝ χρειάζονται props!
const MiddleComponent = () => <IntermediateComponent />;
const IntermediateComponent = () => <DeepComponent />;
```

useContext Hook: Κατανάλωση Context Data

Το `useContext` Hook καταναλώνει Context data, οπότε δεν χρειάζονται **καθόλου props** στα ενδιάμεσα components:

```
import { useContext } from 'react';

const DeepComponent = () => {
  // Απευθείας πρόσβαση στο user!
  const { user, setUser } = useContext(UserContext);

  const updateName = (newName) => {
    setUser(prev => ({ ...prev, name: newName }));
  };

  return (
    <View>
      <Text>Όνομα: {user.name}</Text>
      <Text>Ρόλος: {user.role}</Text>
      <Button
        title="Αλλαγή Ονόματος"
        onPress={() => updateName('Maria')}
      />
    </View>
  );
};
```

Context API - Παράδειγμα με Provider & Χρήση του Custom Hook

```jsx
// UserContext.js
import { createContext, useState, useContext } from 'react';

const UserContext = createContext();

// Custom Provider component
export const UserProvider = ({ children }) => {
  const [user, setUser] = useState(null);
  const [isAuthenticated, setIsAuthenticated] = useState(false);

  const login = (userData) => {
    setUser(userData);
    setIsAuthenticated(true);
  };

  const logout = () => {
    setUser(null);
    setIsAuthenticated(false);
  };

  return (
    <UserContext.Provider value={{
      user,
      isAuthenticated,
      login,
      logout
    }}>
      {children}
    </UserContext.Provider>
  );
};

// Custom Hook για εύκολη χρήση
export const useUser = () => {
  const context = useContext(UserContext);
  if (!context) {
    throw new Error('useUser must be used within UserProvider');
  }
  return context;
};
```

```jsx
// App.js
const App = () => (
  <UserProvider>
    <Navigation />
  </UserProvider>
);

// LoginScreen.js
const LoginScreen = () => {
  const { login } = useUser();

  const handleLogin = () => {
    login({ id: 1, name: 'John', email: 'john@example.com' });
  };

  return <Button title="Login" onPress={handleLogin} />;
};

// ProfileScreen.js
const ProfileScreen = () => {
  const { user, logout, isAuthenticated } = useUser();

  if (!isAuthenticated) return <Text>Please login</Text>;

  return (
    <View>
      <Text>Όνομα: {user.name}</Text>
      <Text>Email: {user.email}</Text>
      <Button title="Logout" onPress={logout} />
    </View>
  );
};
```

Context API - Καλές Πρακτικές

1. **Χωρίστε τα Contexts ανά domain**

2. **Memoize το value για performance**

Το useMemo είναι ένα React Hook που επιτρέπει την απομνημόνευση (memoize) μιας τιμής, ώστε να μην ξαναϋπολογίζεται σε κάθε render αν δεν έχουν αλλάξει οι εξαρτήσεις της.

Χωρίς αυτή σε κάθε render η React θα δημιουργεί ένα νέο object, οπότε το Context θα θεωρεί ότι η τιμή άλλαξε → και θα ξανακάνει re-render όλα τα components που το χρησιμοποιούν.

```
// ✅ Ξεχωριστά contexts
<UserProvider>
  <ThemeProvider>
    <LanguageProvider>
      <App />
    </LanguageProvider>
  </ThemeProvider>
</UserProvider>

// ❌ Ένα μεγάλο context για όλα
<AppProvider value={{ user, theme, language, cart, ... }}>
```

```
const value = useMemo(() => ({
  user, login, logout
}), [user]); // Αποφεύγει unnecessary re-renders
```

Context API - Βελτιστοποίηση με useMemo για αποφυγή περιττών re-renders

```jsx
// UserContext.js
import { createContext, useState, useContext, useMemo } from 'react';

const UserContext = createContext();

export const UserProvider = ({ children }) => {
  const [user, setUser] = useState(null);
  const [isAuthenticated, setIsAuthenticated] = useState(false);

  const login = (userData) => {
    setUser(userData);
    setIsAuthenticated(true);
  };

  const logout = () => {
    setUser(null);
    setIsAuthenticated(false);
  };

  // ✅ Memoize το value ώστε να μη δημιουργείται νέο object σε κάθε render
  const value = useMemo(() => ({
    user,
    isAuthenticated,
    login,
    logout
  }), [user, isAuthenticated]);

  return (
    <UserContext.Provider value={value}>
      {children}
    </UserContext.Provider>
  );
};

export const useUser = () => {
  const context = useContext(UserContext);
  if (!context) {
    throw new Error('useUser must be used within UserProvider');
  }
  return context;
};
```

```jsx
// App.js
const App = () => (
  <UserProvider>
    <Navigation />
  </UserProvider>
);

// LoginScreen.js
const LoginScreen = () => {
  const { login } = useUser();

  const handleLogin = () => {
    login({ id: 1, name: 'Maria', email: 'maria@university.gr' });
  };

  return <Button title="Login" onPress={handleLogin} />;
};

// ProfileScreen.js
const ProfileScreen = () => {
  const { user, logout, isAuthenticated } = useUser();

  if (!isAuthenticated) return <Text>Παρακαλώ συνδεθείτε</Text>;

  return (
    <View>
      <Text>Όνομα: {user.name}</Text>
      <Text>Email: {user.email}</Text>
      <Button title="Logout" onPress={logout} />
    </View>
  );
};
```

useReducer Hook - Complex State Logic

Το `useReducer` είναι ιδανικό για **πολύπλοκη state logic**:

```
import { useReducer } from 'react';

// Reducer function
const cartReducer = (state, action) => {
  switch (action.type) {
    case 'ADD_ITEM':
      return {
        ...state,
        items: [...state.items, action.payload]
      };
    case 'REMOVE_ITEM':
      return {
        ...state,
        items: state.items.filter(item => item.id !== action.payload)
      };
    case 'UPDATE_QUANTITY':
      return {
        ...state,
        items: state.items.map(item =>
          item.id === action.payload.id
            ? { ...item, quantity: action.payload.quantity }
            : item
        )
      };
    default:
      return state;
  }
};
```

useReducer Hook - Χρήση

```
const ShoppingCart = () => {
  const initialState = {
    items: [],
    total: 0
  };

  const [state, dispatch] = useReducer(cartReducer, initialState);

  const addItem = (product) => {
    dispatch({
      type: 'ADD_ITEM',
      payload: { ...product, quantity: 1 }
    });
  };

  const removeItem = (id) => {
    dispatch({
      type: 'REMOVE_ITEM',
      payload: id
    });
  };

  return (
    <View>
      {state.items.map(item => (
        <View key={item.id}>
          <Text>{item.name} x {item.quantity}</Text>
          <Button title="Remove" onPress={() => removeItem(item.id)} />
        </View>
      ))}
    </View>
  );
};
```

useState vs useReducer

Πότε χρησιμοποιούμε ποιο;

useState	useReducer
Simple state (primitives)	Complex state (objects, arrays)
Λίγες state updates	Πολλές state updates
Independent state changes	Related state transitions
Μικρά components	Μεγάλα components

useState Example:

```
const [count, setCount] = useState(0);
const [isOpen, setIsOpen] = useState(false);
```

useReducer Example:

```
const [state, dispatch] = useReducer(formReducer, {
  username: '', password: '', email: '', errors: {}
});
```

Context + useReducer - Συνδυασμός για Global State

```
// store/CartContext.js
const CartContext = createContext();

export const CartProvider = ({ children }) => {
  const [state, dispatch] = useReducer(cartReducer, initialState);

  const addItem = (product) => {
    dispatch({ type: 'ADD_ITEM', payload: product });
  };

  const removeItem = (id) => {
    dispatch({ type: 'REMOVE_ITEM', payload: id });
  };

  const value = {
    items: state.items,
    total: state.total,
    addItem,
    removeItem
  };

  return (
    <CartContext.Provider value={value}>
      {children}
    </CartContext.Provider>
  );
};

export const useCart = () => useContext(CartContext);
```

Παραπομπές & Πηγές

- Official Documentation
  - React Documentation
  - React Native Documentation
  - JavaScript Info