Ανάπτυξη Κινητών Εφαρμογών

Εργαστήριο

Δημήτρης Ρίγγας

riggas@ionio.gr

Τμήμα Πληροφορικής

Ιόνιο Πανεπιστήμιο



Εργαστήριο 4 – React Native Development

Components, Λίστες & useEffect

Props, FlatList, Side Effects, Data Fetching



Στόχος:

- ✓ **Component-based Architecture** στο React Native
- ✓ **Props** και ροή δεδομένων (parent → child)
- ✓ Οργάνωση components σε **αρχεία & φακέλους**
- ✓ Λίστες: **ScrollView** vs **FlatList**
- ✓ **Safe Areas** (notch, status bar)
- ✓ **useEffect Hook** - Side Effects και Lifecycle
- ✓ **Data Fetching** με loading & error states
- ✓ **Derived state** - πότε **δεν** χρειαζόμαστε effect
- ✓ **Custom Hooks**

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 χρησιμοποιηθούν 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

Props - Παράμετροι Components

Τα **props** είναι τα "ορίσματα" ενός component:

- Ο **parent** τα ορίζει ως attributes στο JSX
- Το **child** τα λαμβάνει ως **ένα** object — συνήθως με **destructuring** (Lab 01)
- Μπορεί να είναι strings, numbers, objects, arrays, **functions** (callbacks)
- **Optional props**: ελέγχουμε αν υπάρχουν ή δίνουμε default τιμή

```
// components/UserCard.js – child
export default function UserCard({ name, email, city }) {
  return (
    <View style={styles.card}>
      <Text style={styles.name}>{name}</Text>
      <Text style={styles.email}>{email}</Text>
      {city ? <Text>📍 {city}</Text> : null}
    </View>
  );
}

// App.js – parent
<UserCard name="Μαρία" email="maria@ionio.gr" city="Κέρκυρα" />
<UserCard name="Νίκος" email="nikos@ionio.gr" />

// Default τιμή
function Badge({ label, color = '#3498db' }) { /* ... */ }
```

State vs Props - Βασικές Έννοιες

State	Props
Εσωτερική κατάσταση component	Παράμετροι από parent
Αλλάζει με `setState`	Read-only για το child
Διαχειρίζεται εσωτερικά	Περνάει από έξω
Αλλαγή → re-render	Αλλαγή (από parent) → re-render

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

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

> Το child αλλάζει δεδομένα του parent **μόνο** καλώντας μια function που του πέρασε ο parent ως prop (π.χ. `onPress`).

Οργάνωση σε αρχεία

```
usersapp/
├── App.js
├── components/
│   ├── UserCard.js
│   └── Clock.js
├── hooks/
│   └── useFetch.js
├── assets/
├── app.json
└── package.json
```

- **Ένα component ανά αρχείο**, όνομα αρχείου = όνομα component (PascalCase)
- Custom hooks σε `hooks/`, με όνομα που αρχίζει με `use`

```
// components/UserCard.js
export default function UserCard(props) { /* ... */ }

// App.js
import UserCard from './components/UserCard';
```

```
// Named exports (πολλά από ένα αρχείο)
// theme.js
export const colors = { primary: '#3498db' };
export const spacing = { md: 16 };

// οπουδήποτε
import { colors, spacing } from './theme';
```

Λίστες με ScrollView

Το `ScrollView` είναι ένα `View` που κάνει **scroll** όταν το περιεχόμενο δεν χωράει.

- Κάνει render **όλα** τα children αμέσως
- Ιδανικό για **λίγα** ή σταθερά στοιχεία: φόρμες, σελίδες ρυθμίσεων, οθόνη λεπτομερειών
- Χρειάζεται **`key`** σε κάθε στοιχείο του `map` (Lab 03)

```
const USERS = [
  { id: 1, name: 'Μαρία', email: 'maria@ionio.gr' },
  { id: 2, name: 'Νίκος', email: 'nikos@ionio.gr' },
  // ...
];

<ScrollView>
  {USERS.map(user => (
    <UserCard
      key={user.id}
      name={user.name}
      email={user.email}
    />
  ))}
</ScrollView>
```

ScrollView vs FlatList

ScrollView	FlatList
Κάνει render **όλα** τα children	Κάνει render μόνο ό,τι φαίνεται (+ buffer) — **virtualization**
Μνήμη ∝ πλήθος στοιχείων	Σταθερή μνήμη, ανεξάρτητα από το μέγεθος
Λίγα / σταθερά στοιχεία	Μεγάλες ή δυναμικές λίστες (API)
`map` + `key`	`data` + `renderItem` + `keyExtractor`
—	Έτοιμα: empty state, header/footer, separators, pull-to-refresh, infinite scroll

> ⚠️ **Μην** βάζετε `FlatList` μέσα σε `ScrollView` (ίδιου προσανατολισμού) — χάνεται η virtualization. Χρησιμοποιήστε `ListHeaderComponent` / `ListFooterComponent`.

FlatList - Βασική χρήση

Υποχρεωτικά props:

- `data`: το array
- `renderItem`: function που επιστρέφει το JSX για **ένα** στοιχείο — λαμβάνει `{ item, index }`

Σχεδόν πάντα:

- `keyExtractor`: μοναδικό **string** key ανά στοιχείο (όχι το index!)

Χρήσιμα:

- `ListEmptyComponent` - όταν `data` είναι κενό
- `ListHeaderComponent` / `ListFooterComponent`
- `ItemSeparatorComponent`
- `refreshing` + `onRefresh` - pull-to-refresh
- `onEndReached` - infinite scroll / pagination

```jsx
import { FlatList, Text } from 'react-native';

<FlatList
  data={users}
  keyExtractor={item => item.id.toString()}
  renderItem={({ item }) => (
    <UserCard
      name={item.name}
      email={item.email}
      city={item.address.city}
    />
  )}
  ListEmptyComponent={
    <Text>Δεν βρέθηκαν χρήστες</Text>
  }
  refreshing={refreshing}
  onRefresh={refresh}
/>
```

Safe Areas

Στα σύγχρονα κινητά, μέρος της οθόνης καλύπτεται από **notch / Dynamic Island**, **status bar**, **home indicator**, στρογγυλεμένες γωνίες.

- Βιβλιοθήκη: `react-native-safe-area-context`
  `npx expo install react-native-safe-area-context`
- `SafeAreaProvider` μία φορά, στη ρίζα
- `SafeAreaView` ή `useSafeAreaInsets()` όπου χρειάζεται
- Το `SafeAreaView` του `react-native` είναι **deprecated** (και μόνο iOS)
- Με React Navigation (Lab 05) το **header** χειρίζεται το πάνω μέρος

```
import {
  SafeAreaProvider,
  SafeAreaView,
} from 'react-native-safe-area-context';

export default function App() {
  return (
    <SafeAreaProvider>
      <SafeAreaView style={{ flex: 1 }}>
        <Text style={styles.title}>Χρήστες</Text>
        {/* ... */}
      </SafeAreaView>
    </SafeAreaProvider>
  );
}

// Εναλλακτικά: χειροκίνητο padding
const insets = useSafeAreaInsets();
<View style={{ paddingTop: insets.top }} />
```

Υπενθύμιση: Κανόνες των Hooks

1. **Καλέστε Hooks μόνο στο top level**
   - Όχι μέσα σε `if`, loops ή nested functions
2. **Καλέστε Hooks μόνο από React Functions**
   - Functional Components ✅
   - Custom Hooks ✅
     Functions που χρησιμοποιούν hooks και αρχίζουν με `use`
   - Regular JavaScript functions ❌

Έτσι η React αντιστοιχίζει σωστά κάθε `useState` / `useEffect` στην ίδια "θέση" σε κάθε render.

```jsx
// ✅ Σωστό
const MyComponent = () => {
  const [count, setCount] = useState(0);
  useEffect(() => { /* ... */ }, []);
  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);
}
```

useEffect Hook - Side Effects και Lifecycle

Τα components πρέπει να είναι **pure**: ίδια props/state → ίδιο JSX.

Ό,τι "βγαίνει έξω" από το render είναι **side effect** και πάει σε `useEffect`:

- Data fetching
- Subscriptions (sensors, WebSockets)
- Timers
- Logging / analytics

Το effect τρέχει **μετά** το render (η οθόνη έχει ήδη ζωγραφιστεί).

```
import { useEffect, useState } 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:

Array	Πότε τρέχει
(κανένα)	Μετά από **κάθε** render
`[]`	**Μία** φορά (mount)
`[a, b]`	Mount + όταν αλλάξει `a` ή `b`

> Βάζετε στο array **κάθε** τιμή από props/state που χρησιμοποιεί 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

Χωρίς cleanup: timers & subscriptions συνεχίζουν να τρέχουν → memory leaks, διπλά events.

```
const Clock = () => {
  const [now, setNow] = useState(new Date());

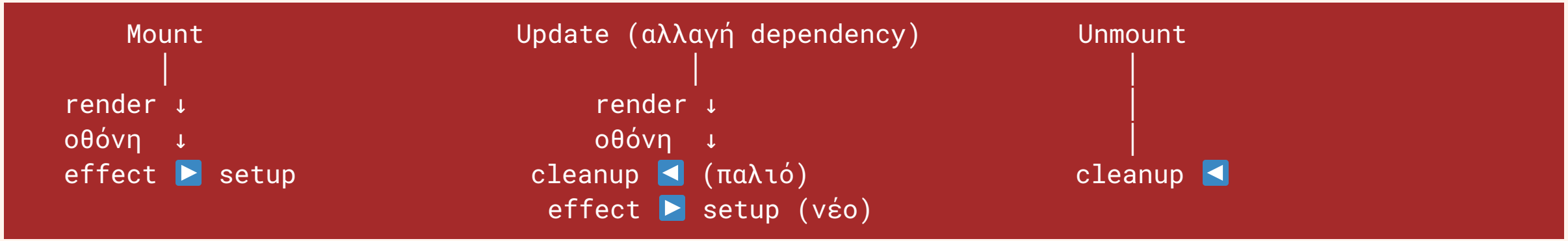
  useEffect(() => {
    // Setup: Δημιουργία interval
    const id = setInterval(() => {
      setNow(new Date());
    }, 1000);

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

  return <Text>{now.toLocaleTimeString('el-GR')}</Text>;
};

// Mount / unmount με conditional rendering
{showClock && <Clock />}
```

Ο κύκλος ζωής με useEffect



Class components (παλιά)	Function components + hooks
`componentDidMount`	`useEffect(() => {...}, [])`
`componentDidUpdate`	`useEffect(() => {...}, [deps])`
`componentWillUnmount`	`return () => {...}` (cleanup)

> Στο development, το **React Strict Mode** μπορεί να εκτελέσει setup → cleanup → setup αμέσως, για να εντοπίσει effects χωρίς σωστό cleanup.

useEffect - Παράδειγμα: Data Fetching

```
const API_URL = 'https://jsonplaceholder.typicode.com/users';

export default function App() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  const loadUsers = async () => {
    try {
      setError(null);
      const response = await fetch(API_URL);
      if (!response.ok) {
        throw new Error(`HTTP ${response.status}`);
      }
      setUsers(await response.json());
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => {
    loadUsers();
  }, []);
```

```
  if (loading) {
    return <ActivityIndicator size="large" />;
  }
  if (error) {
    return (
      <View>
        <Text>Σφάλμα: {error}</Text>
        <Button title="Δοκιμή ξανά" onPress={retry} />
      </View>
    );
  }
  return <FlatList data={users} /* ... */ />;
}
```

Προσοχή:

- `useEffect(async () => ...)` ✗ — το effect πρέπει να επιστρέφει τίποτα ή cleanup, όχι Promise
- Το `fetch` **δεν** κάνει reject σε 404/500 → ελέγχουμε `response.ok`
- **Τρεις** καταστάσεις UI: loading, error, data (+ empty)

Derived State: Δεν χρειάζεστε πάντα effect

Αν μια τιμή **υπολογίζεται** από υπάρχοντα props/state, υπολογίστε τη **κατά το render**.

```
// ❌ Περιττό state + effect
const [filtered, setFiltered] = useState([]);

useEffect(() => {
  setFiltered(
    users.filter(u => u.name.includes(query))
  );
}, [users, query]);
// → 2 renders, κίνδυνος ασυνέπειας
```

```
// ✅ Derived state
const [query, setQuery] = useState('');

const q = query.trim().toLowerCase();
const filteredUsers = users.filter(u =>
  u.name.toLowerCase().includes(q)
);

<TextInput value={query} onChangeText={setQuery} />
<FlatList data={filteredUsers} /* ... */ />
```

Κανόνας: `useEffect` για συγχρονισμό με **εξωτερικά** συστήματα (network, timers, sensors) — όχι για μετασχηματισμό δεδομένων.

Custom Hooks - Reusability

```
// hooks/useFetch.js
export default function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    let cancelled = false;
    const load = async () => {
      try {
        setLoading(true);
        setError(null);
        const response = await fetch(url);
        if (!response.ok) throw new Error(`HTTP ${response.status}`);
        const json = await response.json();
        if (!cancelled) setData(json);
      } catch (err) {
        if (!cancelled) setError(err.message);
      } finally {
        if (!cancelled) setLoading(false);
      }
    };
    load();
    return () => { cancelled = true; };
  }, [url]);

  return { data, loading, error };
}
```

```
// Usage
const UsersScreen = () => {
  const { data, loading, error } = useFetch(
    'https://jsonplaceholder.typicode.com/users'
  );

  if (loading) return <ActivityIndicator />;
  if (error) return <Text>Error: {error}</Text>;
  return <FlatList data={data} /* ... */ />;
};
```

- Custom hook = function που αρχίζει με **use** και καλεί άλλα hooks
- Μοιράζεται **λογική**, όχι state: κάθε component που το καλεί έχει το **δικό του** state
- Το `cancelled` αγνοεί απαντήσεις από "παλιά" requests (αν αλλάξει το `url` ή γίνει unmount)

Debugging Tips

Συμβουλές για Debugging

1. Logs

- Το `console.log` εμφανίζεται στο **terminal** του `npx expo start` (και στο browser console για web)

2. React Native DevTools

- Πατήστε `j` στο terminal του Expo → Console, Sources (breakpoints), **Components** (props & state), Profiler

3. Dev menu

- `m` στο terminal ή **shake** της συσκευής → Reload, Toggle Element Inspector, Performance monitor

4. Log σε κάθε αλλαγή τιμής

```
useEffect(() => {
  console.log('[users]', users.length);
}, [users]);
```

Και τώρα.. *hands on*!

Συνεχίζουμε στο:

https://github.com/riggas-ionio/mobile-apps/tree/class-2026/lab04



Παραπομπές & Πηγές

- Official Documentation
  - React - Passing Props to a Component
  - React - Synchronizing with Effects
  - React - You Might Not Need an Effect
  - React Native - FlatList
  - Expo - Safe areas
  - Expo - Debugging tools

