

Mobile Architecture and Programming LAB (IT-765)

PRACTICAL FILE

**SUBMITTED TO-
Ms. Yashima Hooda
Assistant Professor, USICT**

**SUBMITTED BY-
Adeeb Ahmed
Enroll No.03216404523**



UNIVERSITY SCHOOL OF INFORMATION COMMUNICATION AND TECHNOLOGY

INDEX

S. No	Practical	Date	Signature
1	Design a mobile service architecture for a specific use case, considering scalability, security, and performance.		
2	Create a Login application and open a browser with any one search engine.		
3	Create an UI listing the engineering branches. If user selects a branch name, display the number of semesters and subjects in each semester.		
4	Create a RESTful API using Node.js to expose mobile services.		
5	Consume a RESTful API from a mobile app using JavaScript.		
6	Integrate a third-party web service into a simple mobile app.		
7	Develop a mobile app using cross-platform framework Flutter		
8	Use OWASP security testing tools to identify vulnerabilities in the mobile app.		

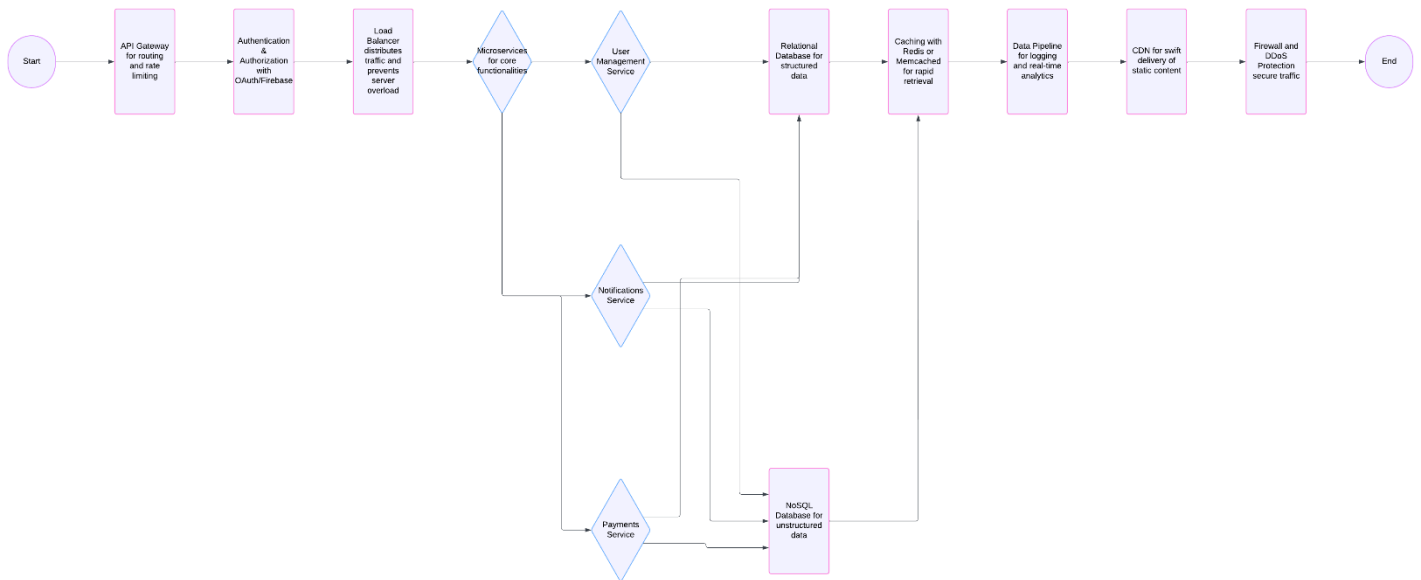
Practical-1

AIM- Design a mobile service architecture for a specific use case, considering scalability, security, and performance.

Use Case: Ride-Sharing Application (like Uber or Lyft)

The app must manage thousands of drivers and riders, secure sensitive data (e.g., payment information), and provide real-time updates (e.g., ride status and driver location). Each component of the architecture has a specific role in ensuring the app runs smoothly and securely, even under high demand.

Architecture Diagram



Detailed Breakdown of the Architecture

1. API Gateway:

- **Function:** Acts as the single entry point for client (mobile app) requests, ensuring that requests are routed to the correct backend service.
- **Scalability:** Provides load balancing and rate limiting to handle peak traffic periods (e.g., rush hours). This avoids overwhelming the backend and ensures the system can handle high traffic loads.
- **Security:** Protects backend services by filtering incoming requests, performing security checks (like IP whitelisting), and managing request authentication.
- **Use in Ride-Sharing:** Routes requests based on purpose, such as requesting a ride, updating profile information, or processing payments. It also enforces rate limits to prevent abuse, like spam ride requests.

2. Authentication & Authorization:

- **Function:** Verifies user identity through integrations with identity providers like OAuth or Firebase. Ensures that only registered users can access certain functionalities.
- **Security:** Ensures that only authenticated users can make requests, protecting the app from unauthorized access to sensitive user data, like payment information or ride history.
- **Use in Ride-Sharing:** Allows riders and drivers to securely log in and restricts access to sensitive endpoints (e.g., payment processing, personal information). Uses tokens or session management to maintain secure access during an active session.

3. Load Balancer:

- **Function:** Distributes incoming requests evenly across backend servers to prevent overload.
- **Scalability:** Balances traffic during peak times, allowing for dynamic scaling of backend services based on load. Ensures system uptime by redirecting traffic to healthy instances if one fails.
- **Performance:** Minimizes request processing times by distributing load efficiently.
- **Use in Ride-Sharing:** Handles high traffic efficiently during peak times (e.g., evening hours). Ensures that users experience minimal wait times and seamless performance by balancing requests among various services.

4. Microservices:

- **Function:** Decomposes the backend into independent, smaller services, each responsible for specific functionalities like User Management, Notifications, and Payments.
- **Scalability:** Each microservice can scale independently based on demand. For instance, the Payments service can scale differently from the Notifications service.
- **Performance:** Reduces bottlenecks by isolating functions and reducing dependencies between them.
- **Use in Ride-Sharing:**
 - **User Management Service:** Stores and manages user profiles, preferences, and ride history. Allows riders and drivers to view and update their information.
 - **Notifications Service:** Sends real-time push notifications for ride updates, promotions, and alerts (e.g., ride arrival or cancellation).
 - **Payments Service:** Manages fare calculations and processes transactions through integrations with payment gateways. This service ensures the security of financial data using encryption and compliance with payment security standards (e.g., PCI DSS).

5. Relational Database:

- **Function:** Manages structured data storage (e.g., SQL databases like PostgreSQL) with strong data integrity, ideal for structured data like transactions and user information.
- **Scalability:** Can handle read-replica scaling or sharding to accommodate high transaction volumes.
- **Security:** Offers fine-grained access control and data integrity features.
- **Use in Ride-Sharing:** Stores critical, structured data like user profiles, ride history, and transaction records, enabling detailed analytics and reporting.

6. NoSQL Database:

- **Function:** Stores unstructured data, such as event logs, JSON documents, or location data, which are frequently accessed and updated.
- **Scalability:** NoSQL databases, like MongoDB or DynamoDB, can handle massive amounts of data with flexible schemas, ideal for storing data in near-real-time.
- **Performance:** Fast retrieval speeds and horizontal scaling improve response times.
- **Use in Ride-Sharing:** Stores unstructured data, such as live driver locations, trip logs, and analytics data, which require quick access and high availability.

7. Caching Layer (Redis/Memcached):

- **Function:** Temporarily stores frequently accessed data in-memory to improve retrieval times.
- **Performance:** Reduces database load by caching responses, resulting in faster data access for repeated requests.
- **Use in Ride-Sharing:** Caches data like frequently used routes, user preferences, and driver statuses, reducing the time required to retrieve commonly accessed data.

8. Data Pipeline:

- **Function:** Collects and processes logs and analytics in real-time for monitoring and alerting.
- **Scalability:** Enables real-time analysis and can integrate with monitoring tools to track system health and performance.
- **Performance:** Provides data insights that can identify and resolve issues proactively.
- **Use in Ride-Sharing:** Processes real-time data for system health and usage analytics, allowing operators to monitor driver behavior, app performance, and demand trends.

9. Content Delivery Network (CDN):

- **Function:** Caches static content (e.g., images, maps, and CSS files) globally for quick access.
- **Performance:** Reduces latency by delivering content from edge servers close to the user's location.
- **Use in Ride-Sharing:** Ensures fast loading times for static assets, such as maps, icons, and profile images, particularly for users in different regions.

10. Firewall and DDoS Protection:

- **Function:** Filters and protects the network against malicious traffic and attacks.
- **Security:** Defends against unauthorized access and distributed denial-of-service (DDoS) attacks, which could disrupt service.
- **Use in Ride-Sharing:** Provides security against unauthorized access to the application backend, helping to prevent data breaches and denial-of-service incidents.

Practical-2

AIM- Create a Login application and open a browser with any one search engine.

#CODE

```
import React, { useState } from 'react';
import { View, Text, TextInput, TouchableOpacity, StyleSheet, Alert } from 'react-native';
import * as WebBrowser from 'expo-web-browser';

export default function App() {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");

  const handleLogin = () => {
    if (email && password) {
      Alert.alert("Login Success", `Welcome ${email}!`);
    } else {
      Alert.alert("Error", "Please enter both email and password.");
    }
  };

  const openSearchEngine = async () => {
    const result = await WebBrowser.openBrowserAsync('https://www.google.com');
  };

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Login</Text>
      <TextInput
        style={styles.input}
        placeholder="Email"
        placeholderTextColor="#888"
        keyboardType="email-address"
        autoCapitalize="none"
      />
    </View>
  );
}
```

```

    value={email}
    onChangeText={setEmail}
  />
  <TextInput
    style={styles.input}
    placeholder="Password"
    placeholderTextColor="#888"
    secureTextEntry
    value={password}
    onChangeText={setPassword}
  />
  <TouchableOpacity style={styles.button} onPress={handleLogin}>
    <Text style={styles.buttonText}>Login</Text>
  </TouchableOpacity>
  <TouchableOpacity style={styles.link} onPress={openSearchEngine}>
    <Text style={styles.linkText}>Open Google Search</Text>
  </TouchableOpacity>
</View>
);
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#f8f9fa',
    justifyContent: 'center',
    padding: 20,
  },

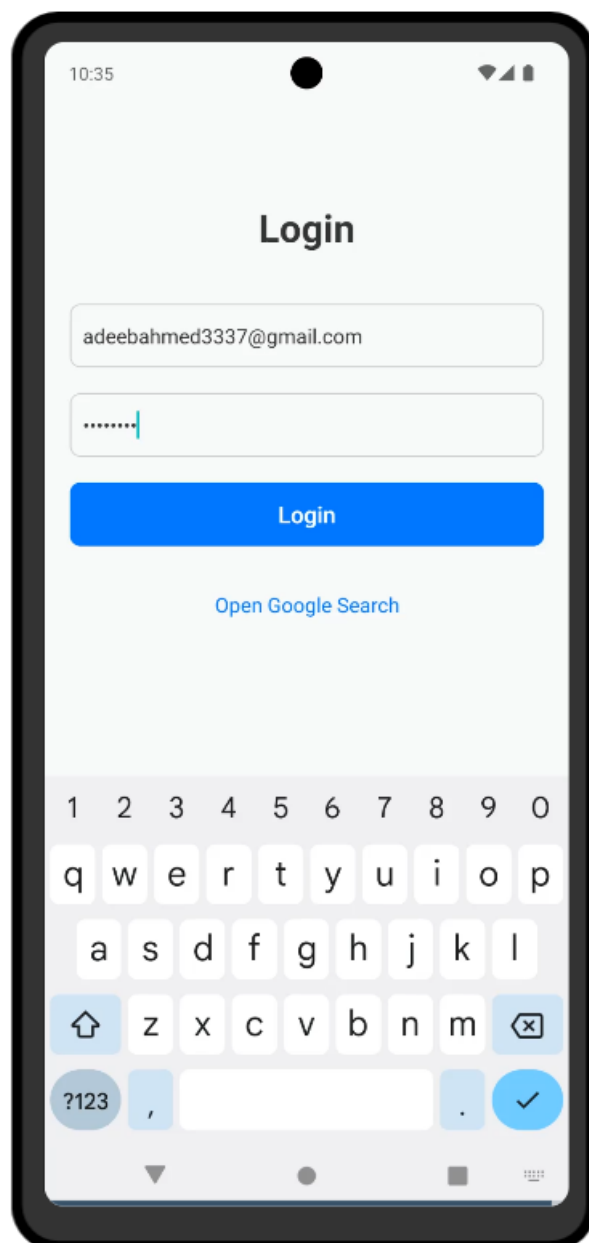
```



```
title: {
  fontSize: 30,
  fontWeight: 'bold',
  color: '#333',
  textAlign: 'center',
  marginBottom: 40,
},
input: {
  height: 50,
  borderColor: '#ccc',
  borderWidth: 1,
  borderRadius: 8,
  paddingHorizontal: 10,
  fontSize: 16,
  marginBottom: 20,
  color: '#333',
},
button: {
  backgroundColor: '#007bff',
  height: 50,
  justifyContent: 'center',
  alignItems: 'center',
  borderRadius: 8,
  marginBottom: 15,
},
buttonText: {
  color: 'fff',
  fontSize: 18,
  fontWeight: '600',
},
```

```
link: {  
  marginTop: 20,  
  alignItems: 'center',  
},  
linkText: {  
  color: '#007bff',  
  fontSize: 16,  
},  
});
```

#OUTPUT



Practical-3

AIM- Create an UI listing the engineering branches. If user selects a branch name, display the number of semesters and subjects in each semester

#CODE

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

import { View, Text, TouchableOpacity, FlatList, StyleSheet } from 'react-native';

const App = () => {
  const [selectedBranch, setSelectedBranch] = useState(null);

  const branches = [
    { name: 'Computer Science', semesters: 8, subjectsPerSemester: 5 },
    { name: 'Mechanical Engineering', semesters: 8, subjectsPerSemester: 6 },
    { name: 'Electrical Engineering', semesters: 8, subjectsPerSemester: 5 },
    { name: 'Civil Engineering', semesters: 8, subjectsPerSemester: 6 },
  ];

  const handleBranchSelect = (branch) => {
    setSelectedBranch(branch);
  };

  const BranchList = () => (
    <FlatList
      data={branches}
      keyExtractor={(item) => item.name}
      renderItem={({ item }) => (
        <TouchableOpacity style={styles.branchItem} onPress={() => handleBranchSelect(item)}>
          <Text style={styles.branchText}>{item.name}</Text>
        </TouchableOpacity>
      )}
    />
  );
};
```

```

const BranchDetails = () => (
  <View style={styles.detailsContainer}>
    <Text style={styles.detailsTitle}>{selectedBranch.name}</Text>
    <Text style={styles.detailsText}>Total Semesters: {selectedBranch.semesters}</Text>
    {Array.from({ length: selectedBranch.semesters }).map((_, i) => (
      <Text key={i} style={styles.detailsText}>
        Semester {i + 1}: {selectedBranch.subjectsPerSemester} Subjects
      </Text>
    ))}
    <TouchableOpacity style={styles.backButton} onPress={() => setSelectedBranch(null)}>
      <Text style={styles.backButtonText}>Back</Text>
    </TouchableOpacity>
  </View>
);

return (
  <View style={styles.container}>
    <Text style={styles.title}>Engineering Branches</Text>
    {selectedBranch ? <BranchDetails /> : <BranchList />}
  </View>
);
};

```

```

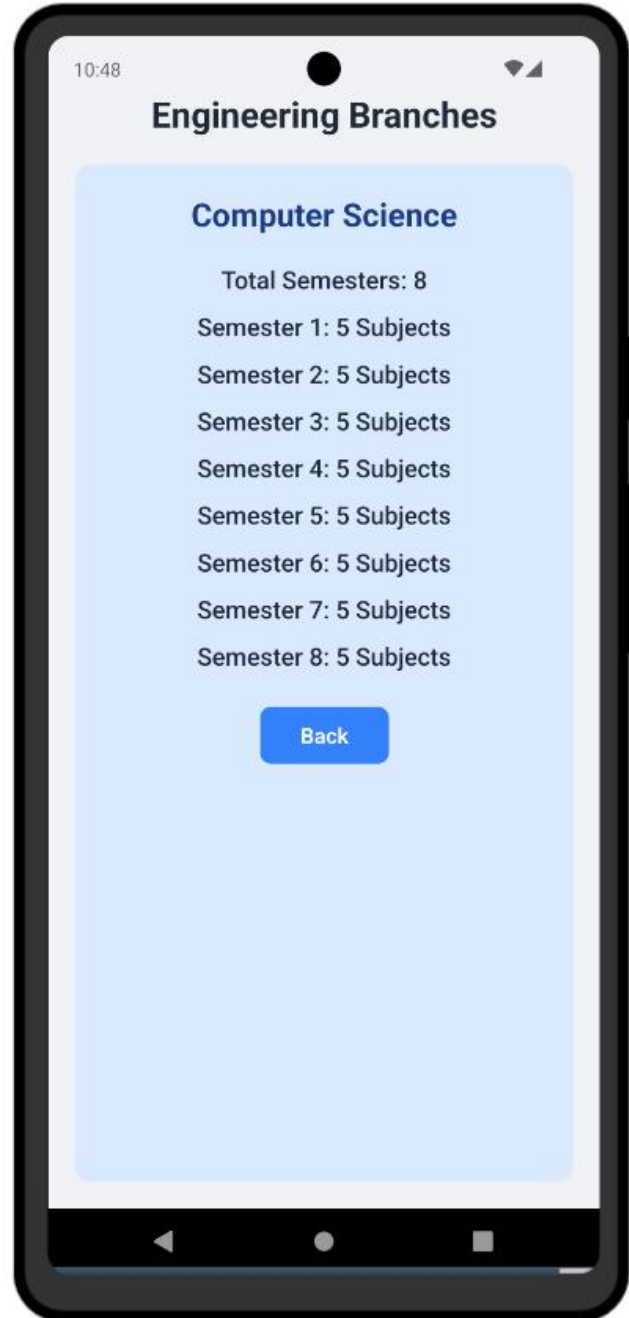
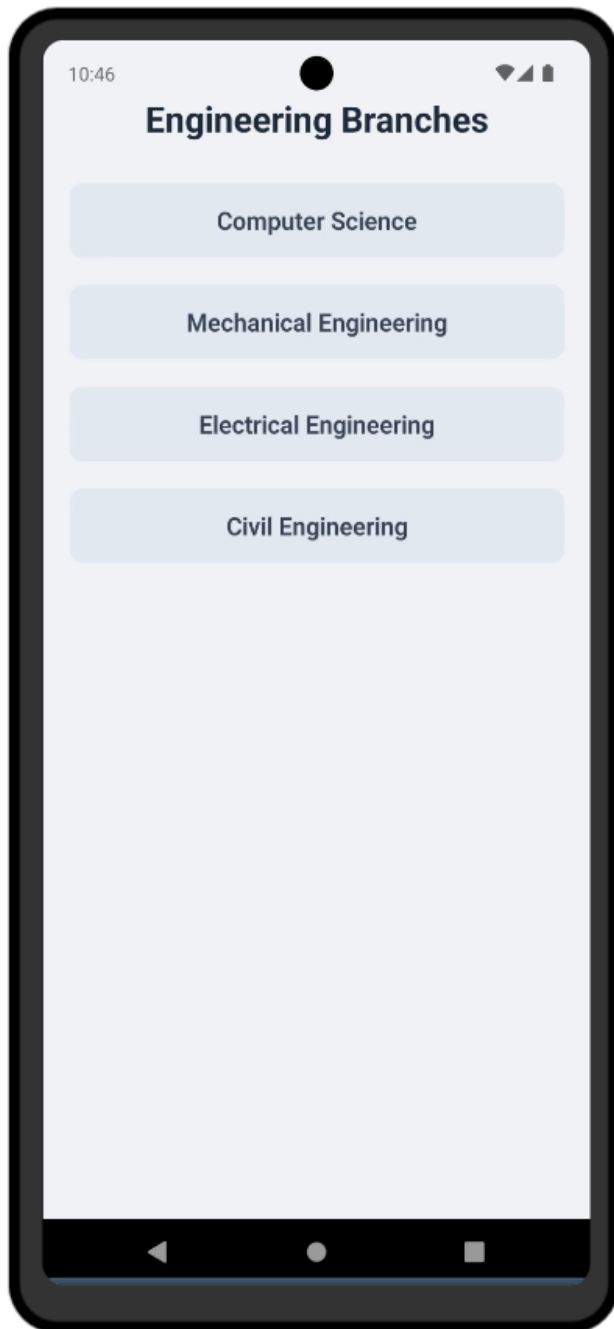
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#f1f3f6',
    padding: 20,
  },
  title: {
    fontSize: 26,
    fontWeight: '700',
    color: '#1f2937',
  },
});

```

```
    textAlign: 'center',
    marginVertical: 20,
  },
  branchItem: {
    backgroundColor: '#e3e8f1',
    borderRadius: 10,
    padding: 15,
    marginVertical: 10,
    alignItems: 'center',
    shadowColor: '#000',
    shadowOffset: { width: 0, height: 2 },
    shadowOpacity: 0.2,
    shadowRadius: 5,
  },
  branchText: {
    fontSize: 18,
    fontWeight: '600',
    color: '#374151',
  },
  detailsContainer: {
    flex: 1,
    backgroundColor: '#dbeafe',
    borderRadius: 12,
    padding: 20,
    alignItems: 'center',
    shadowColor: '#000',
    shadowOffset: { width: 0, height: 3 },
    shadowOpacity: 0.3,
    shadowRadius: 6,
  },
  detailsTitle: {
    fontSize: 24,
    fontWeight: '700',
    color: '#1e3a8a',
```

```
marginBottom: 15,  
  },  
  detailsText: {  
    fontSize: 18,  
    fontWeight: '500',  
    color: '#1f2937',  
    marginVertical: 5,  
  },  
  backButton: {  
    backgroundColor: '#3b82f6',  
    borderRadius: 8,  
    padding: 10,  
    marginTop: 20,  
    paddingHorizontal: 30,  
  },  
  backButtonText: {  
    color: 'fff',  
    fontSize: 16,  
    fontWeight: '600',  
  },  
});  
  
export default App;
```

#OUTPUT



Practical-4

AIM- Create a RESTful API using Node.js to expose mobile services.

#CODE

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const PORT = 3000;

app.use(bodyParser.json());

// In-memory data store for mobile devices
let mobileDevices = [
  { id: 1, brand: 'Apple', model: 'iPhone 13', price: 999 },
  { id: 2, brand: 'Samsung', model: 'Galaxy S21', price: 799 },
];

// Helper function to find a device by ID
const findDeviceById = (id) => mobileDevices.find(device => device.id === parseInt(id));

// GET: Fetch all devices
app.get('/api/devices', (req, res) => {
  res.status(200).json(mobileDevices);
});

// GET: Fetch a single device by ID
app.get('/api/devices/:id', (req, res) => {
  const device = findDeviceById(req.params.id);
  if (!device) return res.status(404).json({ message: 'Device not found' });
  res.status(200).json(device);
});
```


// POST: Add a new device

```
app.post('/api/devices', (req, res) => {  
  const { brand, model, price } = req.body;  
  const newDevice = {  
    id: mobileDevices.length + 1,  
    brand,  
    model,  
    price,  
  };  
  mobileDevices.push(newDevice);  
  res.status(201).json(newDevice);  
});
```

// PUT: Update an existing device by ID

```
app.put('/api/devices/:id', (req, res) => {  
  const device = findDeviceById(req.params.id);  
  if (!device) return res.status(404).json({ message: 'Device not found' });  
  
  const { brand, model, price } = req.body;  
  device.brand = brand || device.brand;  
  device.model = model || device.model;  
  device.price = price || device.price;  
  
  res.status(200).json(device);  
});
```

// DELETE: Remove a device by ID

```
app.delete('/api/devices/:id', (req, res) => {  
  const deviceIndex = mobileDevices.findIndex(device => device.id === parseInt(req.params.id));  
  if (deviceIndex === -1) return res.status(404).json({ message: 'Device not found' });
```

```
const deletedDevice = mobileDevices.splice(deviceIndex, 1);  
res.status(200).json({ message: 'Device deleted', deletedDevice });  
});
```

// Start the server

```
app.listen(PORT, () => {  
  console.log(`Server is running on http://localhost:${PORT}`);  
});
```

#OUTPUT

The screenshot shows a REST client interface. At the top, a GET request is defined for the URL `http://localhost:3003/api/devices`. Below the URL bar, tabs for Params, Authorization, Headers (8), Body, Scripts, Tests, and Settings are visible. The 'Body' tab is selected, showing a message: 'This request does not have a body'. Below this, radio buttons for content types (none, form-data, x-www-form-urlencoded, raw, binary, GraphQL) are shown, with 'none' selected. A 'Send' button is on the right. The response section at the bottom shows a status of '200 OK' with a response time of 54 ms and a size of 368 B. The response body is displayed in 'Pretty' JSON format, showing an array of two device objects. A 'Postbot' button is located at the bottom right of the response area.

```
1  [  
2    {  
3      "id": 1,  
4      "brand": "Apple",  
5      "model": "iPhone 13",  
6      "price": 999  
7    },  
8    {  
9      "id": 2,  
10     "brand": "Samsung",  
11     "model": "Galaxy S21",  
12     "price": 799  
13   }  
14 ]
```

Practical-5

AIM- Consume a RESTful API from a mobile app using JavaScript.

#CODE

```
import React, { useState, useEffect } from 'react';
import { View, Text, FlatList, StyleSheet, ActivityIndicator } from 'react-native';

const App = () => {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);

  // Function to fetch data from API
  const fetchData = async () => {
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/users');
      const result = await response.json();
      setData(result);
      setLoading(false);
    } catch (error) {
      console.error(error);
    }
  };

  // Fetch data when component mounts
  useEffect(() => {
    fetchData();
  }, []);

  // Loading indicator
  if (loading) {
    return (
      <View style={styles.loader}>
        <ActivityIndicator size="large" color="#0000ff" />
      </View>
    );
  }
}
```

```

        </View>

    );
}

// Render the data
return (
    <View style={styles.container}>
        <Text style={styles.title}>Using REST API By Adeeb Ahmed</Text>
        <FlatList
            data={data}
            keyExtractor={({item}) => item.id.toString()}
            renderItem={({ item }) => (
                <View style={styles.item}>
                    <Text style={styles.name}>{item.name}</Text>
                    <Text style={styles.email}>{item.email}</Text>
                </View>
            )}
        />
    </View>
);
};

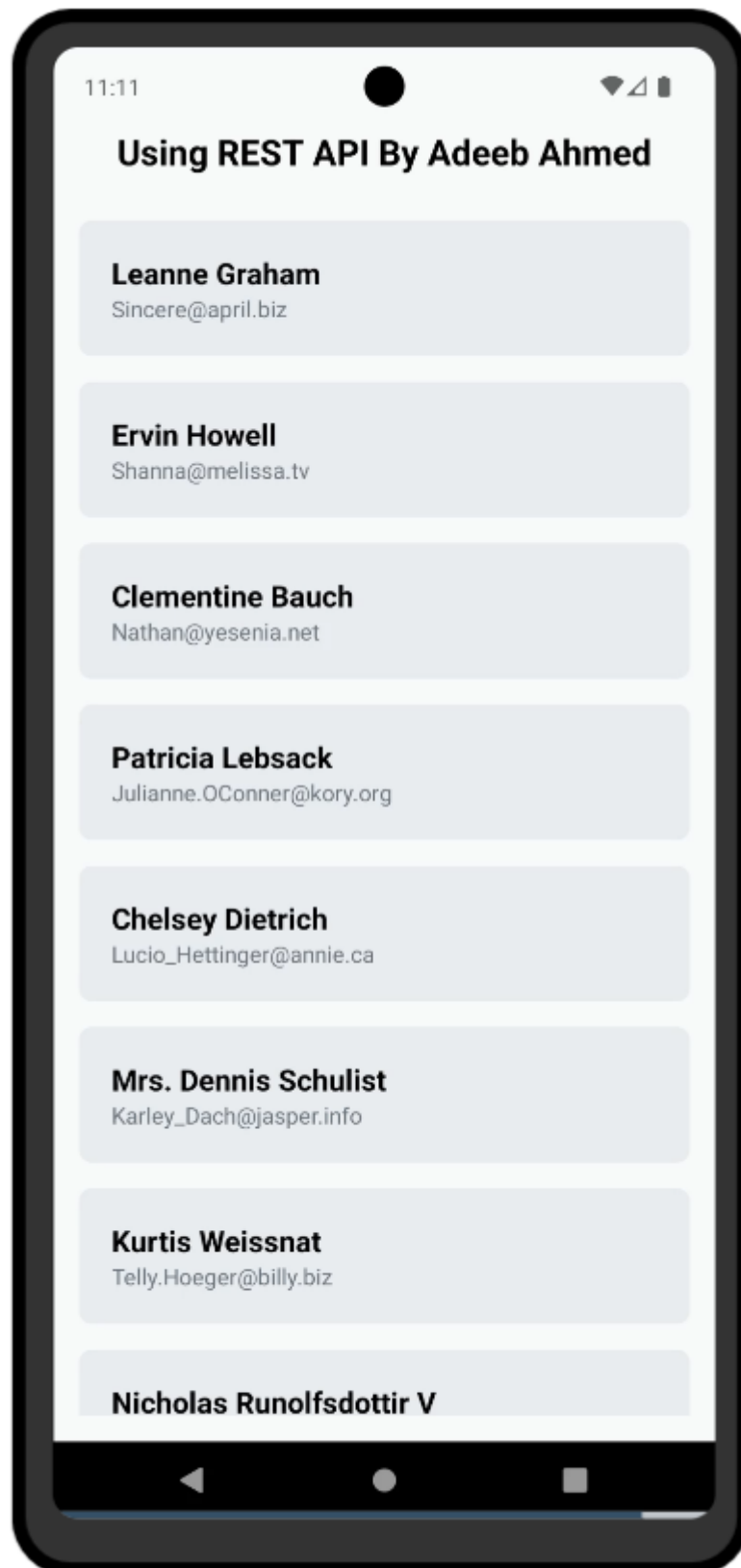
const styles = StyleSheet.create({
    container: {
        flex: 1,
        padding: 16,
        paddingTop: 50,
        backgroundColor: '#f8f9fa',
    },

```

```
title: {
  fontSize: 22,
  fontWeight: 'bold',
  marginBottom: 20,
  textAlign: 'center',
},
loader: {
  flex: 1,
  justifyContent: 'center',
  alignItems: 'center',
},
item: {
  padding: 20,
  marginVertical: 8,
  backgroundColor: '#e9ecef',
  borderRadius: 8,
},
name: {
  fontSize: 18,
  fontWeight: 'bold',
},
email: {
  fontSize: 14,
  color: '#6c757d',
},
});
```

```
export default App;
```

#OUTPUT



Practical-6

AIM- Integrate a third-party web service into a simple mobile app.

#CODE

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

import { View, Text, TextInput, Button, StyleSheet, ActivityIndicator } from 'react-native';

const App = () => {
  const [city, setCity] = useState("");
  const [weather, setWeather] = useState(null);
  const [loading, setLoading] = useState(false);

  const API_KEY = '9r64298f0abb56390u06ufjknk34950';

  const fetchWeather = async () => {
    if (!city.trim()) {
      alert("Please enter a city name.");
      return;
    }

    setLoading(true);
    try {
      const response = await
fetch('https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${API_KEY}&units=metric`');
      const data = await response.json();
      if (data.cod === 200) {
        setWeather({
          temp: data.main.temp,
          description: data.weather[0].description,
          humidity: data.main.humidity,
          windSpeed: data.wind.speed,
        });
      } else {
        alert("City not found!");
      }
    } catch (error) {
      console.error(error);
      alert("Error fetching weather data.");
    }
    setLoading(false);
  };

  return (
    <View style={styles.container}>
      <Text style={styles.title}>Weather App</Text>
      <TextInput
        style={styles.input}
        placeholder="Enter city name"
        value={city}
        onChangeText={setCity}
      />
      <Button title="Get Weather" onPress={fetchWeather} />
    </View>
  );
};
```

```

{loading ? (
  <ActivityIndicator size="large" color="#0000ff" style={styles.loader} />
) : (
  weather && (
    <View style={styles.weatherContainer}>
      <Text style={styles.weatherText}>Temperature: {weather.temp}°C</Text>
      <Text style={styles.weatherText}>Description: {weather.description}</Text>
      <Text style={styles.weatherText}>Humidity: {weather.humidity}%</Text>
      <Text style={styles.weatherText}>Wind Speed: {weather.windSpeed} m/s</Text>
    </View>
  )
)}
</View>
);
};

```

```

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
    justifyContent: 'center',
    backgroundColor: '#f0f8ff',
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    textAlign: 'center',
    marginBottom: 20,
  },
  input: {
    height: 50,
    borderColor: 'ccc',
    borderWidth: 1,
    padding: 10,
    marginBottom: 10,
    borderRadius: 5,
  },
  loader: {
    marginTop: 20,
  },
  weatherContainer: {
    marginTop: 20,
    padding: 15,
    backgroundColor: '#e0f7fa',
    borderRadius: 8,
  },
  weatherText: {
    fontSize: 18,
    marginBottom: 5,
    textAlign: 'center',
  },
});

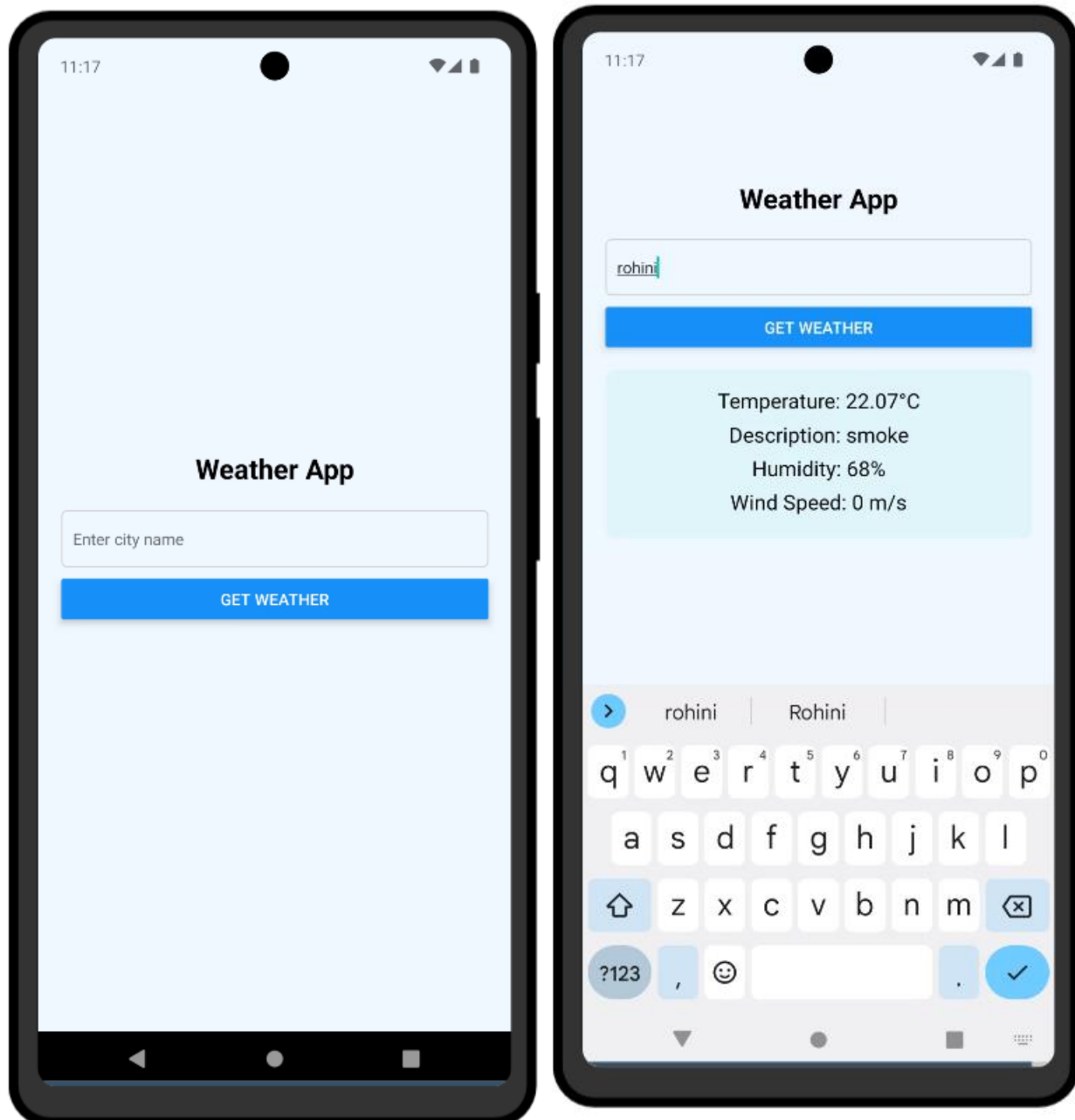
```

```

export default App;

```


#OUTPUT



Practical-7

AIM- Develop a mobile app using cross-platform framework Flutter.

#CODE

```
import 'package:flutter/material.dart';
import 'dart:convert';
import 'package:http/http.dart' as http;

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Joke App',
      theme: ThemeData(primarySwatch: Colors.blue),
      home: JokeScreen(),
    );
  }
}

class JokeScreen extends StatefulWidget {
  @override
  _JokeScreenState createState() => _JokeScreenState();
}

class _JokeScreenState extends State<JokeScreen> {
  String joke = "Press the button to get a random joke!";

  Future<void> fetchJoke() async {
    final response = await http.get(Uri.parse("https://v2.jokeapi.dev/joke/Any?type=single"));
    if (response.statusCode == 200) {
      final data = json.decode(response.body);
      setState(() {
        joke = data['joke'];
      });
    } else {
      setState(() {
        joke = "Failed to load joke.";
      });
    }
  }

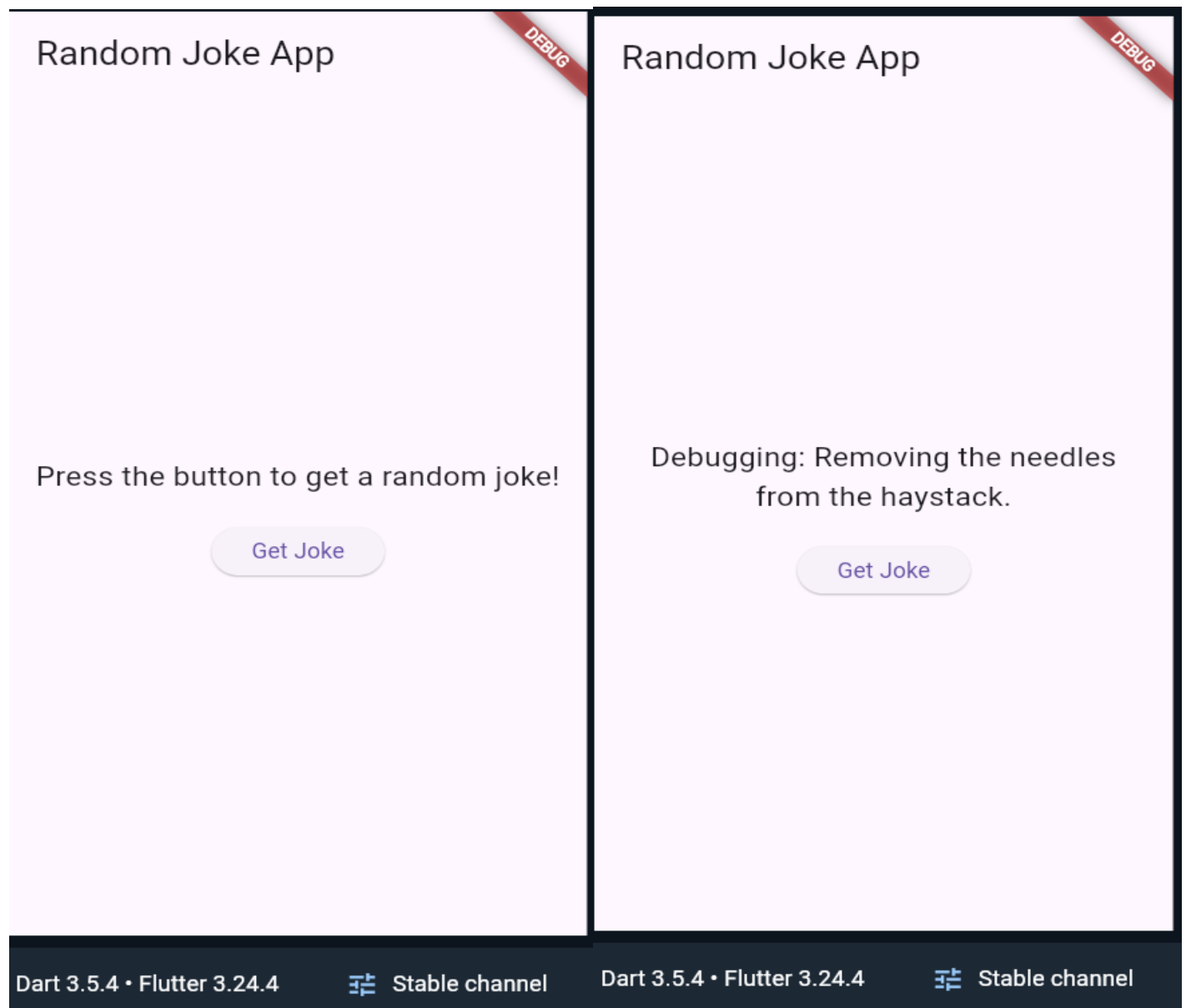
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Random Joke App")),
      body: Center(
        child: Padding(
          padding: const EdgeInsets.all(16.0),
          child: Column(
```

```

mainAxisAlignment: MainAxisAlignment.center,
children: [
  Text(
    joke,
    style: TextStyle(fontSize: 18),
    textAlign: TextAlign.center,
  ),
  SizedBox(height: 20),
  ElevatedButton(
    onPressed: fetchJoke,
    child: Text("Get Joke"),
  ),
],
),
),
),
);
}
}

```

#OUTPUT



Practical-8

AIM- Use OWASP security testing tools to identify vulnerabilities in the mobile app.

1. OWASP Mobile Security Testing Guide (MSTG)

- The **OWASP Mobile Security Testing Guide (MSTG)** provides a comprehensive framework for testing the security of mobile apps. The guide is divided into several areas like security testing, coding, and security requirements.
- Key security areas to focus on:
 - **Code Security:** Ensuring that the app does not expose sensitive information, APIs, or cryptographic keys.
 - **Data Storage:** Ensuring that sensitive data is encrypted properly both in transit and at rest.
 - **Network Communication:** Ensuring that data exchanged between the mobile app and servers is encrypted (e.g., using HTTPS).

2. OWASP ZAP (Zed Attack Proxy)

- **OWASP ZAP** is a popular penetration testing tool for identifying security vulnerabilities in web and mobile applications. ZAP can be used to test the backend APIs used by your mobile app, including issues such as:
 - **Broken Authentication:** Testing for flaws in authentication mechanisms.
 - **Sensitive Data Exposure:** Ensuring that sensitive information such as passwords, tokens, and personal data are not exposed through insecure APIs.

3. OWASP Dependency-Check

- This tool scans your app's dependencies for known vulnerabilities in third-party libraries or frameworks. It can be integrated into your mobile app development process to ensure that the app does not include insecure dependencies.

4. Mobile App Penetration Testing (PenTest)

- Penetration testing on the mobile app focuses on finding vulnerabilities in the app's code and behavior by simulating real-world attacks. Some common tests include:
 - **Reverse Engineering:** Using tools like JADX or IDA Pro to decompile the APK and look for sensitive information or weak encryption keys.
 - **Dynamic Analysis:** Testing the app while it is running on a mobile device using tools like Frida or Burp Suite to analyze network requests and interactions.

5. OWASP Security Knowledge Framework (SKF)

- The **OWASP SKF** provides security best practices and secure coding techniques. It can be useful when you want to ensure that the code you write for your mobile app adheres to security standards and practices. It includes:
 - Secure handling of user input.
 - Prevention of common vulnerabilities like SQL Injection, Cross-Site Scripting (XSS), and others.

6. Mobile Security Tools

Some other tools you can use include:

- **Burp Suite**: Ideal for intercepting and modifying HTTP/HTTPS traffic to test for vulnerabilities in APIs or data communication.
- **Frida**: A dynamic instrumentation toolkit for performing penetration testing, especially useful for testing apps running on Android or iOS.
- **Drozer**: A comprehensive security testing framework for Android that helps test apps for common vulnerabilities such as insecure data storage or improper use of permissions.

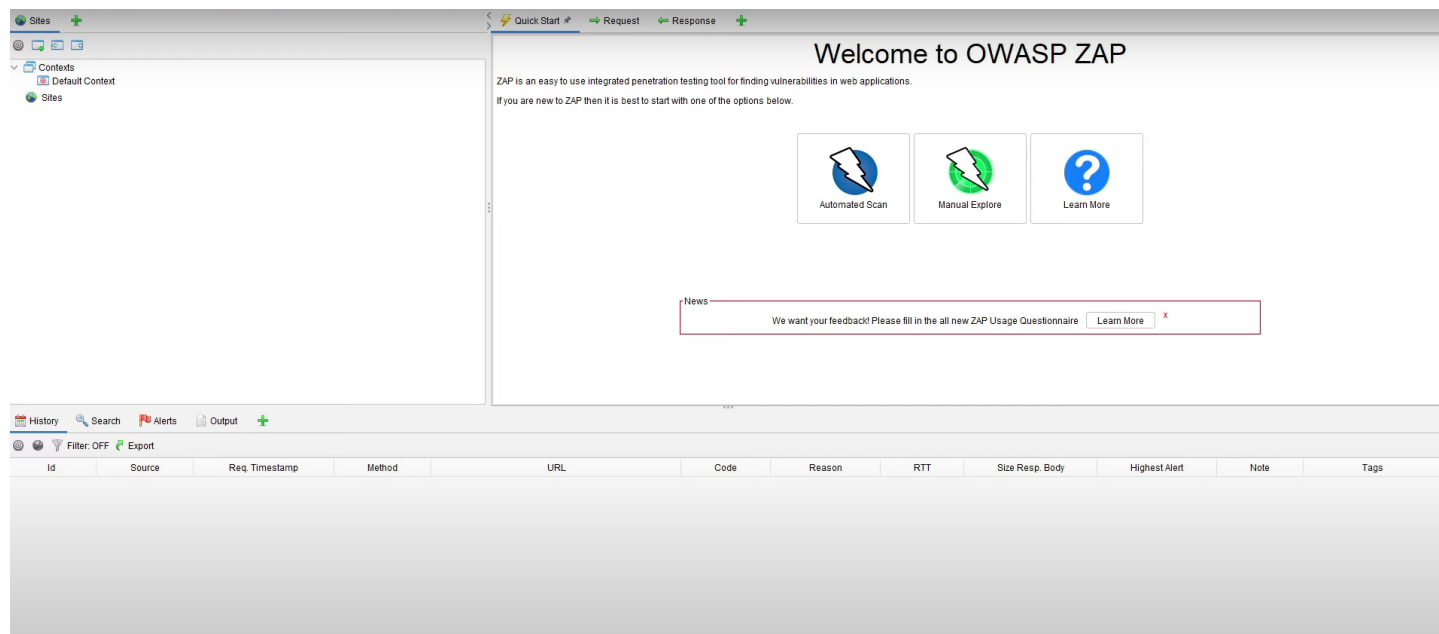
Testing Process

1. **Static Analysis**: Review the app's source code and assets for sensitive information or insecure coding practices. Tools like **Checkmarx** or **Fortify** can help automate this process.
2. **Dynamic Analysis**: Run the app in an environment (e.g., emulator or real device) and use tools like **Burp Suite** or **OWASP ZAP** to analyze the traffic between the app and backend servers.
3. **API Testing**: Test all API endpoints that the app uses to ensure they are secure, especially for authorization and access control issues.
4. **Mobile Device Testing**: Use tools like **Frida** or **Drozer** to inspect how the app interacts with the mobile operating system, especially focusing on data storage and inter-process communication.

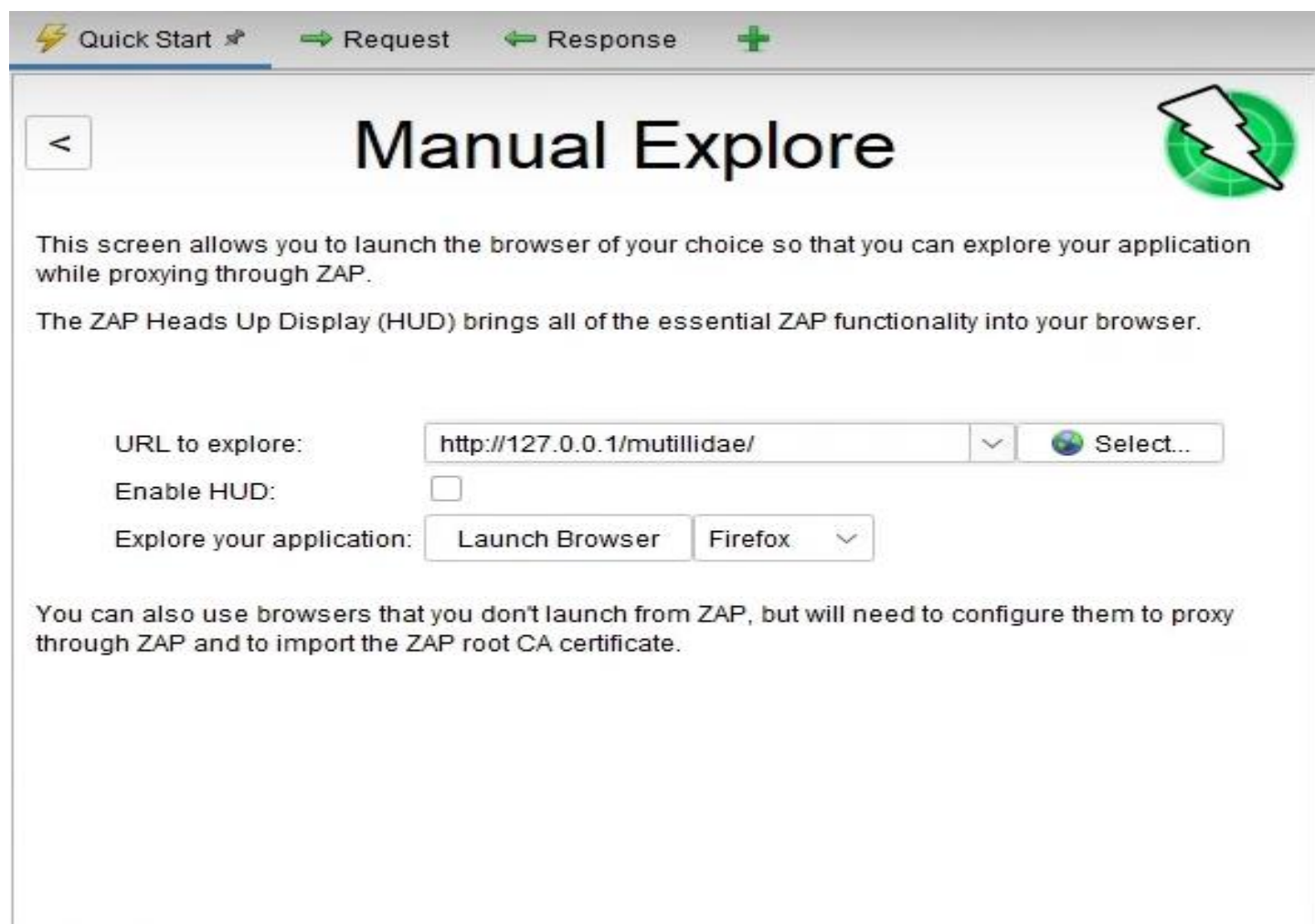
Common Vulnerabilities to Look For:

- **Insecure Data Storage**: Check if sensitive data like passwords or personal information is stored in plaintext.
- **Weak Authentication and Authorization**: Ensure proper session management and that users can only access resources they are authorized to.
- **Cryptographic Issues**: Verify that cryptography is used correctly, including for data in transit and at rest.
- **Reverse Engineering**: Check if the app can be easily reverse-engineered to expose sensitive information or logic flaws.

#OUTPUT



The screenshot shows the OWASP ZAP Welcome screen. The top navigation bar includes 'Quick Start', 'Request', and 'Response'. The main content area has a heading 'Welcome to OWASP ZAP' and a subtext 'ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. If you are new to ZAP then it is best to start with one of the options below.' There are three buttons: 'Automated Scan' (lightning bolt icon), 'Manual Explore' (lightning bolt icon), and 'Learn More' (question mark icon). A 'News' banner at the bottom says 'We want your feedback! Please fill in the all new ZAP Usage Questionnaire' with a 'Learn More' link.



The screenshot shows the 'Manual Explore' screen in OWASP ZAP. The top navigation bar includes 'Quick Start', 'Request', and 'Response'. The main content area has a heading 'Manual Explore' and a subtext 'This screen allows you to launch the browser of your choice so that you can explore your application while proxying through ZAP. The ZAP Heads Up Display (HUD) brings all of the essential ZAP functionality into your browser.' There are three input fields: 'URL to explore:' with a text box containing 'http://127.0.0.1/mutillidae/' and a 'Select...' button; 'Enable HUD:' with a checkbox; and 'Explore your application:' with a 'Launch Browser' button and a 'Firefox' dropdown menu. A paragraph at the bottom says 'You can also use browsers that you don't launch from ZAP, but will need to configure them to proxy through ZAP and to import the ZAP root CA certificate.'



The screenshot shows the 'History' table in OWASP ZAP. The table has columns: Id, Source, Req. Timestamp, Method, URL, Code, Reason, RTT, Size Resp. B..., Highest Alert, Note, and Tags. There are two rows of data.

Id	Source	Req. Timestamp	Method	URL	Code	Reason	RTT	Size Resp. B...	Highest Alert	Note	Tags
84	Proxy	30/07/2021, 19:02:...	GET	http://127.0.0.1/mutillidae/index.php?pa...	200	OK	25...	56,816 bytes	Medium		Form, Passw..
83	Proxy	30/07/2021, 19:02:...	GET	http://127.0.0.1/mutillidae/	200	OK	11...	53,280 bytes	Medium		Form, Hidde...