QN 1

a.

```
const db = firebase.firestore();

db.collection('employees')
  .where('salary', '>', 1200000)
  .get()
  .then((querySnapshot) => {
    querySnapshot.forEach((doc) => {
      console.log(`${doc.id} => ${doc.data()}`);
    });
  })
  .catch((error) => {
    console.log('Error getting documents: ', error);
  });
```

b.

In Flutter, a BuildContext is a handle to the location of a widget in the widget tree. It is used to access information about the widget's position and properties within the widget hierarchy. Every widget has its own BuildContext that allows it to interact with other widgets and access the widget tree. Below is the code for it

```
import 'package:flutter/material.dart';


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


class TodoApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: TodoHomePage(),
    );
  }
}

class TodoHomePage extends StatefulWidget {
  @override
  _TodoHomePageState createState() => _TodoHomePageState();
}

class _TodoHomePageState extends State<TodoHomePage> {
  List<String> todos = [];
  TextEditingController todoController = TextEditingController();

  void addTodo() {
    setState(() {
      todos.add(todoController.text);
      todoController.clear();
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Simple To-Do List'),
      ),
      body: Column(
        children: [
          Padding(
            padding: const EdgeInsets.all(8.0),
            child: TextField(
              controller: todoController,
              decoration: InputDecoration(labelText: 'Enter a task'),
            ),
          ),
          ElevatedButton(
            onPressed: addTodo,
            child: Text('Add Task'),
          ),
          Expanded(
            child: ListView.builder(
              itemCount: todos.length,
              itemBuilder: (context, index) {
                return ListTile(
                  title: Text(todos[index]),
                );
              },
            ),
          ),
        ],
      ),
    );
  }
}
```

c.

In Node.js, event-driven programming refers to a programming paradigm in which the flow of the program is determined by events such as user actions, sensor outputs, or messages from other programs. In Node.js, the EventEmitter module is commonly used for managing events. It allows functions (listeners) to be attached to events, and these listeners are invoked when the corresponding event is emitted.

For example:

```
const EventEmitter = require('events');
const eventEmitter = new EventEmitter();
const handleEvent = () => {
  console.log('Event occurred!');
};
eventEmitter.on('myEvent', handleEvent);
eventEmitter.emit('myEvent');
```

d.

```
const express = require('express');
const app = express();
const port = 3000;

app.use(express.static('public'));
const employees = [
  { id: 'Emp01', name: 'John Doe', salary: 1500000 },
  { id: 'Emp02', name: 'Jane Doe', salary: 1100000 },
];
app.get('/employees', (req, res) => {
  res.json(employees);
});

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

The HTML

```
const db = firebase.firestore();


function updateEmployee(docId, updatedData) {
  db.collection('employees').doc(docId).update(updatedData)
    .then(() => {
      console.log('Document successfully updated!');
    })
    .catch((error) => {
      console.error('Error updating document: ', error);
    });
}
updateEmployee('Emp01', {
  name: 'Jane Doe',
  phone: '987654321',
  email: 'jane@example.com',
  dob: '1992-02-02',
  jobDesk: 'Manager',
  salary: 1600000,
});
```

QN 2

a.

```
const db = firebase.firestore();


function addEmployee(employee) {
  db.collection('employees').add(employee)
    .then((docRef) => {
      console.log('Document written with ID: ', docRef.id);
    })
    .catch((error) => {
      console.error('Error adding document: ', error);
    });
}
addEmployee({
  name: 'John Doe',
  phone: '123456789',
  email: 'john@example.com',
  dob: '1990-01-01',
  jobDesk: 'Software Engineer',
  salary: 1500000,
});
```

b.

```
const db = firebase.firestore();


function updateEmployee(docId, updatedData) {
  db.collection('employees').doc(docId).update(updatedData)
    .then(() => {
      console.log('Document successfully updated!');
    })
    .catch((error) => {
      console.error('Error updating document: ', error);
```

```
   });
}
updateEmployee('Emp01', {
  name: 'Jane Doe',
  phone: '987654321',
  email: 'jane@example.com',
  dob: '1992-02-02',
  jobDesk: 'Manager',
  salary: 1600000,
});
```

c.

1. Install Express

```
npm init -y
npm install express
```

2. Create app.js:

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

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

3. Run the application:

```
node app.js
```

d.

Flutter button UI

```
import 'package:flutter/material.dart';
```

```
void main() {
 runApp(ButtonApp());
}

class ButtonApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   home: Scaffold(
    appBar: AppBar(title: Text('Button Example')),
    body: Center(
     child: ElevatedButton(
      onPressed: () {},
      child: Text('Click Me!'),
     ),
    ),
   ),
  );
 }
}
```

QN.4

i. Abstraction: Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object. In Jumia's shopping system, the concept of "Payment" can be abstracted. The customer doesn't need to know how the payment is processed, only that they have to input payment details like credit card info.

Example:

```
class Payment {
  processPayment(amount) {
    console.log(`Processing payment of ${amount}...`);
  }
}

const customerPayment = new Payment();
customerPayment.processPayment(1500);
```

ii. Objects: An object is an instance of a class in Object-Oriented Programming (OOP) that holds data and methods to manipulate that data. In Jumia, an example could be an "Order" object which represents the details of a customer's order.

```
class Order {
  constructor(orderId, customerName, items) {
    this.orderId = orderId;
    this.customerName = customerName;
    this.items = items;
  }

  displayOrderDetails() {
    console.log(`Order ${this.orderId} by ${this.customerName}:`);
    this.items.forEach(item => console.log(`${item.name} - ${item.price}`));
  }
}

const order = new Order('JMA001', 'John Doe', [
  {name: 'Phone', price: 300},
  {name: 'Shoes', price: 50}
]);
order.displayOrderDetails();
```

b)

- **Firebase** is a Backend-as-a-Service (BaaS) platform that provides real-time databases, authentication, and other services for mobile and web applications. It uses a NoSQL database and allows for real-time data synchronization.
- **MongoDB** is a NoSQL database where you can store documents in BSON (Binary JSON) format. It's a general-purpose database often used in backends that handle large datasets.

Example:

- **Firebase** is ideal for real-time applications like a Jumia shopping cart system, where changes in stock and prices are reflected in real-time.
- **MongoDB** is better suited for large datasets where you might have complex queries on the product catalog.

Firebase Example:

```
firebase.firestore().collection('products').get().then((snapshot) => {
  snapshot.forEach(doc => {
    console.log(doc.data());
  });
});
```

MongoDB Example:

```
const { MongoClient } = require('mongodb');
const uri = "your_mongodb_uri";
const client = new MongoClient(uri);

async function getProducts() {
  await client.connect();
  const collection = client.db("shopDB").collection("products");
  const products = await collection.find().toArray();
  console.log(products);
}
```

c)

Flutter widgets are the building blocks of a Flutter app's user interface. Widgets can either be **stateless** or **stateful**.

- **Stateless Widgets** do not change their state after they are built, meaning they remain the same throughout the app's life cycle.
- **Stateful Widgets** can change their internal state and re-render based on user interactions or other factors.

Example of a Stateful Widget:

```
class MyCounter extends StatefulWidget {
  @override
  _MyCounterState createState() => _MyCounterState();
}


class _MyCounterState extends State<MyCounter> {
  int _counter = 0;


  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }


  @override
  Widget build(BuildContext context) {
    return Column(
      children: <Widget>[
        Text('Counter: $_counter'),
        ElevatedButton(onPressed: _incrementCounter, child: Text('Increment')),
      ],
    );
  }
}
```

d)

NPM (Node Package Manager) is the default package manager for Node.js. It is used to install, manage, and share JavaScript libraries or modules.

Implementing Async in Node.js: Node.js supports asynchronous programming via callbacks, promises, and async/await. Here's an example using async/await:

```
const fetchData = async () => {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error fetching data:', error);
  }
};

fetchData();
```

QN5

a.

i. Inheritance:

Inheritance is a principle where a class inherits properties and methods from another class. In Jumia's system, a class Product could be inherited by a more specific class like Electronics or Clothing.

Example:

```
class Product {
  constructor(name, price) {
    this.name = name;
    this.price = price;
  }
}

class Electronics extends Product {
  constructor(name, price, brand) {
    super(name, price);
    this.brand = brand;
  }
}

const phone = new Electronics('Smartphone', 500, 'Samsung');
console.log(phone);
```

ii. Instance:

An instance is a specific realization of any object created from a class. For example, in Jumia, a specific customer order or product is an instance of a broader Order or Product class.

Example:

```
const order1 = new Order('JMA001', 'John Doe', [{name: 'Phone', price: 500}]);
const order2 = new Order('JMA002', 'Jane Doe', [{name: 'Laptop', price: 1000}]);
```

Here, order1 and order2 are instances of the Order class.

b)

Firebase automatically generates unique document IDs when using add(). However, you can also use push() in the Firebase Realtime Database to generate unique keys.

Example

```
const uniqueKey = firebase.firestore().collection('orders').doc().id;

firebase.firestore().collection('orders').doc(uniqueKey).set({

  customerName: 'John Doe',

  total: 2000

});
```

c)

State management in Flutter refers to how you manage data that can change over time (such as app state, UI state, etc.) and keep your UI updated with that data. In an app like SafeBoda, state management would handle:

- Ride request status (e.g., requested, accepted, completed).
- Real-time updates about driver location, user's current location, etc.

Flutter provides different approaches for state management:

- **setState()**: Used for simple state changes.
- **Provider**: Used for more complex applications where you need to share data across widgets.
- **Bloc**: Used for reactive programming.

For example, if you want to show real-time updates of the ride request status in SafeBoda, you would use Provider to update the UI when the state of the ride changes.

d)

Modules in Node.js are libraries or functionalities packaged together that you can import and use in your application. Common Node.js modules include:

- **HTTP**: For handling HTTP requests and responses.
- **fs**: For interacting with the file system.
- **path**: For working with file and directory paths.

Example:

```
const http = require('http');

const fs = require('fs');

const server = http.createServer((req, res) => {

  fs.readFile('index.html', (err, data) => {

    res.writeHead(200, { 'Content-Type': 'text/html' });
  res.write(data);

  res.end();

 });

});


server.listen(3000);
```

Frontend vs. Backend Development:

- **Frontend Development** deals with the visual elements of an application that users interact with. It involves technologies like HTML, CSS, JavaScript, and frameworks like React, Angular, or Vue.js.
- **Backend Development** is concerned with server-side logic, databases, and APIs that power the frontend. It involves languages like Node.js, Python, Java, and databases like MongoDB or Firebase.

4o