WEB X.0 IAT 2

Module 4

1) What is MongoDB?

MongoDB is an open-source document-oriented database that is designed to store a large scale of data and also allows you to work with that data very efficiently. It is categorized under the NoSQL (Not only SQL) database because the storage and retrieval of data in the MongoDB are not in the form of tables.

2) What is a collection and document in MongoDB? Give an example.

In MongoDB, a **collection** is a group of MongoDB documents. It is the equivalent of a table in a relational database.

Example: db.createCollection("users") creates a collection named "users".

A **document** is a set of key-value pairs in MongoDB. It is equivalent to a row in a relational database. Documents are JSON-like data structures that can contain various types of data, including strings, numbers, arrays, and even nested documents.

Example:

```
{
   "_id": ObjectId("61fd1fe61086e5000155b514"),
   "username": "john_doe",
   "full_name": "John Doe",
   "age": 30,
   "email": "john@example.com"
}
```

3) What are the features of MongoDB?

1. **Ad hoc queries:** MongoDB supports flexible ad hoc queries for data retrieval.
2. **Indexing:** Efficient indexing improves query performance in MongoDB.
3. **Replication:** MongoDB offers replication for high availability and fault tolerance.
4. **Sharding:** Horizontal scaling is possible through sharding in MongoDB.
5. **Data duplication:** Duplication of data across servers enhances fault tolerance.
6. **Load balancing:** MongoDB provides load balancing to evenly distribute client requests.
7. **Map-reduce and aggregation:** Supports map-reduce and aggregation operations.
8. **JavaScript usage:** MongoDB employs JavaScript for querying and manipulation.
9. **Scalability:** Designed for scalable handling of large datasets and high throughput.
10. **JSON storage:** MongoDB stores data in a flexible JSON-like format.
11. **Security:** Provides authentication, authorization, encryption, and auditing for data security.

4) List and Explain Data Types in MongoDB with example.

1. **Integer**: Stores numerical values, supporting both 32-bit and 64-bit integers.
   Example: {"Integer example": 62}
2. **Boolean**: Stores Boolean values (true or false).
   Example: {"Nationality Indian": true}
3. **Double**: Stores floating-point data.
   Example: {"double data type": 3.1415}
4. **String**: Stores text data.
   Example: {"string data type": "This is a sample message."}
5. **Arrays**: Stores arrays or lists of values under a single key.
   Example: {"Array Example": ["BCA", "BS", "MCA"]}
6. **Object**: Stores embedded documents.
   Example: {"Object data type": "This is Object", "Marks": {"English": 94, "Maths.": 96}}
7. **Null**: Represents a null value.
   Example: {"EmailID": null}
8. **Date**: Stores the current date and time in UNIX-time format.
   Example: {"Date": ISODate("2022-03-30T12:00:00Z")}
9. **Timestamp**: Stores a 64-bit value representing time.
   Example: {"Timestamp": Timestamp(1648656000, 1)}
10. **Object ID**: Stores the ID of the document.
    Example: {"_id": ObjectId("61fd1fe61086e5000155b514")}

5) Explain MongoDB CRUD Operations with example.

1. **Create Operations:** Add new documents to a collection.
   db.collection.insertOne() or db.collection.insertMany()

   Example: db.collection.insertOne({ "name": "John", "age": 30 })

   ```
   db.collection.insertMany([
     { "name": "John", "age": 30 },
     { "name": "Jane", "age": 25 }
   ])
   ```

2. **Read Operations:** Retrieve documents from a collection.
   db.collection.find()

   Example: db.collection.find({ "name": "John" })

3. **Update Operations:** Update/Replace documents in a collection.
   db.collection.updateOne() or db.collection.updateMany() or db.collection.replaceOne()

   Example: db.collection.updateOne({ "name": "John" }, { $set: { "age": 35 } })

   db.collection.updateMany({ "name": "John" }, { $set: { "age": 35 } })

   db.collection.replaceOne({ "name": "John" }, { "name": "Jack", "age": 40 })

4. **Delete Operations:** Delete documents from a collection.
   db.collection.deleteOne() or db.collection.deleteMany()

   Example: db.collection.deleteOne({ "name": "John" })
   db.collection.deleteMany({ "name": "John" })

6) Insert, delete, Update, find, replace, bulkwrite

1. **Insert**: Adds new documents to a collection.

   Example: **db.collection.insertOne({"name": "John", "age": 30})**

2. **Delete**: Removes documents from a collection based on specified criteria.

   Example: **db.collection.deleteOne({"name": "John"})**

3. **Update**: Modifies existing documents in a collection.

   Example: **db.collection.updateOne({"name": "John"}, {"$set": {"age": 35}})**

4. **Find**: Retrieves documents from a collection based on specified criteria.

   Example: **db.collection.find({"name": "John"})**

5. **Replace**: Replaces an existing document with a new one in a collection.

   Example: **db.collection.replaceOne({"name": "John"}, {"name": "Jane", "age": 35})**

6. **BulkWrite**: Performs multiple write operations in a single request for efficiency.

   Example:
```
db.collection.bulkWrite([
  { insertOne: { "document": 1 } },
  { updateOne: { filter: { "name": "John" }, update: { $set: { "age": 40 } } } },
  { deleteOne: { "name": "Jane" } }
])
```

7) Write a short note on Mongoose schemas and Models.

Schema serves as a blueprint for defining the structure and behaviour of documents within a MongoDB collection. Specifies the fields, their data types, and validation rules.

Models apply the defined schema to each document in its corresponding collection. It compiles a version of the schema for document interactions and handles CRUD operations for documents in the collection.

Example:
```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

// Define Schema
const accountSchema = new Schema({
   acc_no: { type: Number, required: true, unique: true },
   accholder_name: { type: String, required: true, unique: true },
   created_on: { type: Date, required: true, unique: true }
});

// Define Model
const Account = mongoose.model('Account', accountSchema);

// Export Model
module.exports = Account;
```

8) How to set connection of NodeJS with MongoDB driver.

To connect a Node.js application to MongoDB, we have to use the Mongoose library.

Set the connection:

```
// Import the Mongoose library
const mongoose = require("mongoose");
// Connect to the MongoDB database server
mongoose.connect("mongodb://0.0.0.0:27017/mydb",  // MongoDB connection URL
{ useNewUrlParser:true, useUnifiedTopology:true})
// Promise-based connection handling
.then(()=>console.log("connection successful"))
.catch((err)=>console.log(err));
```

9) Demonstrate the usage of mongoose library in manipulating the database in MongoDB.

```
// Import mongoose library
const mongoose = require('mongoose');

// Connect to MongoDB database
mongoose.connect('mongodb:// 0.0.0.0:27017/mydb',
{ useNewUrlParser: true, useUnifiedTopology: true });

// Define schema for the collection
const Schema = mongoose.Schema;
const userSchema = new Schema({
  name: String,
  email: { type: String, required: true },
  age: { type: Number, default: 18 },
  createdAt: { type: Date, default: Date.now }
});

// Create model based on schema
const User = mongoose.model('User', userSchema);

// Create a new user document
const newUser = new User({
  name: 'John Doe',
  email: 'john@example.com',
  age: 30
});

// Save the document to the collection
newUser.save()
  .then((user) => {
    console.log('User saved successfully:', user);
  })
  .catch((error) => {
    console.error('Error saving user:', error);
  });
```

10) Write the queries to perform CRUD operations.

1. **Create Operations:** Add new documents to a collection.
   db.collection.insertOne() or db.collection.insertMany()
2. **Read Operations:** Retrieve documents from a collection.
   db.collection.find()
3. **Update Operations:** Update/Replace documents in a collection.
   db.collection.updateOne() or db.collection.updateMany() or db.collection.replaceOne()
4. **Delete Operations:** Delete documents from a collection.
   db.collection.deleteOne() or db.collection.deleteMany()

11) List and explain types of HTTP request and response codes supported by REST API.

Types of HTTP Request Methods:

1. **GET:** Retrieves data from the API without altering it.
2. **POST:** Submits new data to the API for processing or storage.
3. **PUT:** Updates existing data in the API with new information.
4. **PATCH:** Modifies a subset of existing data in the API.
5. **DELETE:** Removes specific data from the API.

Types of HTTP Response Status Codes:

1. **200 OK**: Successful request.
2. **201 Created**: Resource created successfully.
3. **204 No Content**: Successful request with no content returned.
4. **301 Moved Permanently**: Resource permanently moved.
5. **400 Bad Request**: Client error.
6. **401 Unauthorized**: Authentication required.
7. **403 Forbidden**: Permission denied.
8. **404 Not Found**: Resource not found.
9. **410 Gone**: Resource no longer available.
10. **429 Too Many Requests**: Rate limit exceeded.
11. **500 Internal Server Error**: Server error.
12. **503 Service Unavailable**: Server temporarily unable to handle the request.

12) Explain REST API features.

REST API features:

1. **Client-Server Architecture**: Requests from clients, responses from servers.
2. **Statelessness**: Each request carries all needed info; no server-side storage of state.
3. **Cacheability**: Responses can be cached by clients for faster subsequent requests.
4. **Layered System**: Modular backend systems that can be updated independently.
5. **Code-On-Demand** (Optional): Servers can send executable code to clients.
6. **Uniform Interface**: Standardized methods (GET, POST, etc.) and formats (JSON, XML).

13) Explain REST API design patterns.

REST API design patterns ensure consistency, scalability, and maintainability:

1. **Descriptive Endpoint Names**: Use intuitive names reflecting the accessed resource.
2. **GET vs. POST Operations**: Employ GET for data retrieval and POST for modification.
3. **Caching for Performance**: Implement mechanisms to store frequently accessed data locally.
4. **API Parameters for Flexibility**: Incorporate parameters for customizing requests.
5. **Leverage Existing Conventions**: Use established standards like HTTP methods.
6. **Comprehensive Documentation**: Document the API thoroughly using OpenAPI specs.
7. **Consistency with Style Guides**: Adhere to coding conventions for consistency across endpoints.

14) What is sharding in MongoDB?

Sharding in MongoDB is the process of distributing data across multiple machines to improve scalability and performance. It allows MongoDB to handle large datasets and high write loads by horizontally partitioning data and distributing it across multiple servers called shards. Each shard contains a subset of the data, and MongoDB automatically balances the data distribution across shards.

15) How to connect MongoDB from NodeJS. Perform CRUD operations to manipulate MongoDB documents.

Set the connection:

```
// Import the Mongoose library
const mongoose = require("mongoose");
// Connect to the MongoDB database server
mongoose.connect("mongodb://0.0.0.0:27017/mydb",  // MongoDB connection URL
{ useNewUrlParser:true, useUnifiedTopology:true})
// Promise-based connection handling
.then(()=>console.log("connection successful"))
.catch((err)=>console.log(err));
```

CRUD operations:

```
// Insert a document
const db = client.db('mydb');
const collection = db.collection('mycollection');
collection.insertOne({ name: 'John', age: 30 })
  .then(result => console.log('Document inserted:', result.insertedId))
  .catch(err => console.error('Error inserting document:', err));

// Find documents
collection.find({ name: 'John' }).toArray()
  .then(docs => console.log('Found documents:', docs))
  .catch(err => console.error('Error finding documents:', err))
```

```
// Update a document
collection.updateOne({ name: 'John' }, { $set: { age: 35 } })
  .then(result => console.log('Document updated:', result))
  .catch(err => console.error('Error updating document:', err));

// Delete a document
collection.deleteOne({ name: 'John' })
  .then(result => console.log('Document deleted:', result))
  .catch(err => console.error('Error deleting document:', err));
```

16) What are the REST API rules? (Same as Q. 12)

REST API rules:

7. **Client-Server Architecture**: Requests from clients, responses from servers.
8. **Statelessness**: Each request carries all needed info; no server-side storage of state.
9. **Cacheability**: Responses can be cached by clients for faster subsequent requests.
10. **Layered System**: Modular backend systems that can be updated independently.
11. **Code-On-Demand** (Optional): Servers can send executable code to clients.
12. **Uniform Interface**: Standardized methods (GET, POST, etc.) and formats (JSON, XML).

Module 5

1) How to create a simple application in flask? Explain all steps with an example.

1. Flask Environment Setup:

Step 1: Install virtualenv for development environment: **pip install virtualenv**
Step 2: Create new virtual environment: **mkdir newproj => cd newproj => virtualenv venv**
Step 3: Activate the virtual environment: **venv\scripts\activate**
Step 4: Install Flask: **pip install Flask**

2. First Web Application:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def home():
   return "Hello";
if __name__ =="__main__":
   app.run(debug = True)
```

Run the command "**python app.py**" to run the application.

Output: Hello

2) Enlist the features of Flask.

Features of Flask:

1. Lightweight Framework
2. Development Server and debugger
3. Unit Testing
4. Restful Request Handling
5. URL Routing
6. Unicode Support
7. Secure cookies

3) What is WSGI and Jinja2?

WSGI stands for Web Server Gateway Interface. It is a standard interface between web servers and web applications or frameworks in Python. WSGI enables communication between the web server and Python web applications, allowing for interoperability between different servers and frameworks.

Jinja2 is a templating engine for Python. It provides a way to generate dynamic HTML content by combining templates with data sources. Jinja2 allows developers to create reusable templates and fill them with dynamic content, making it easier to generate web pages dynamically in Python web applications.

4) Explain the use of app.route() and app.run() in flask application.

In Flask, **app.route()** and **app.run()** are important functions used for defining URL routes and running the Flask application, respectively.

1. **app.route(rule, options)**: **app.route()** is a decorator that binds a URL route to a specific function in your Flask application.

   It takes two parameters:

   - **rule:** Specifies the URL path triggering the associated function.
   - **options:** Additional settings like allowed HTTP methods.

2. **app.run(host, port, debug, options)**: **app.run()** is used to run the Flask development server to serve the Flask application.

   It takes several parameters, but the most commonly used ones are:

   - **host:** Specifies the hostname or IP address to listen on. Default is 127.0.0.1 (localhost).
   - **port:** Sets the port number to listen on. Default is 5000.
   - **debug:** When True, it enables a debug mode for additional debug info and auto-reloads server on code changes.

5) Explain app routing in python flask with a proper example.

App routing in Python Flask refers to the process of mapping URLs to specific functions in the Flask application that handle the logic for those URLs. This allows developers to create dynamic web applications where different URLs trigger different functionalities.

Example:

```
from flask import Flask
app = Flask(__name__)
@app.route('/home'          // maps the /home URL route to the home() function
def home():                 //defines the logic executed when the /home route is accessed
   return "Hello, welcome to our website!"
if __name__ == "__main__":      //ensures that script runs the Flask application directly
   app.run(debug=True)          // starts the Flask application
```

6) Why flask is called a microframework?

Flask is considered a “micro” framework because it is lightweight and only provides essential components for web development such as request handling, routing, sessions, etc. There are no specific libraries or tools required by Flask and it doesn’t contain components like form validation, database integration, and upload handling that other frameworks may provide; instead, Flask allows you to choose which libraries to use instead of imposing a pre-set solution upon you. But that doesn’t mean that Flask isn’t versatile; it supports extensions and custom-coded modules that add application features as if they were created within Flask.

7) What do you mean by a dynamic URLs in flask? Explain with an example.

Dynamic URLs in Flask allow for the creation of variable parts within the URL routes. These variable parts can capture values from the URL and pass them as parameters to the associated view functions.

Example:

```
from flask import Flask
app = Flask(__name__)
@app.route('/home/<name>')
def home(name):
   return "Hello, " + name
if __name__ == "__main__":
   app.run(debug=True)
```

In this example, the **<name>** part of the URL **/home/<name>** is a dynamic variable. When a user accesses **/home/John**, Flask captures the value "John" and passes it to the **home()** function as the **name** parameter. The function then returns "Hello, John".

Flask provides converters to specify the data type of these variables. Converters such as **<int:age>** and **<float:age>** can be used to specify the data type of dynamic variables.

8) Explain URL building in Python Flask.

URL building in Python Flask refers to the process of dynamically constructing URLs for specific functions or routes within a Flask application. Flask provides the **url_for()** function for URL building. This function takes the name of the target function as its first argument and any additional keyword arguments representing variable parts of the URL. It then generates a URL corresponding to the specified function, incorporating provided values for variable parts.

Example:

```
from flask import Flask, redirect, url_for
app = Flask(__name__)

@app.route('/admin')
def admin():
   return 'admin'

@app.route('/student')
def student():
   return 'student'

@app.route('/user/<name>')
def user(name):
   if name == 'admin':
      return redirect(url_for('admin'))
   elif name == 'student':
      return redirect(url_for('student'))

if __name__ == '__main__':
   app.run(debug=True)
```

In this example, the **url_for()** function dynamically builds URLs for the **admin** and **student** routes based on the provided function names.

9) What are the HTTP methods in Python flask? Explain with an example.

Method	**Description**
GET	It is the most common method which can be used to send data in the unencrypted form to the server.
HEAD	It is similar to the GET but used without the response body.
POST	It is used to send the form data to the server. The server does not cache the data transmitted using the post method.
PUT	It is used to replace all the current representation of the target resource with the uploaded content.
DELETE	It is used to delete all the current representation of the target resource specified in the URL.

Example:

```
from flask import *
app = Flask(__name__)
@app.route('/login',methods = ['GET'])              //GET method
                        or
@app.route('/login',methods = ['POST'])             //POST method
def login():
    uname=request.form['uname']
    passwrd=request.form['pass']
    if uname=="ayush" and passwrd=="google":
        return "Welcome %s" %uname
if __name__ == '__main__':
   app.run(debug = True)
```

Output: Welcome ayush

10) How to set, access and delete cookies in python?

```
from flask import Flask, make_response, request
app = Flask(__name__)
//Set cookies
@app.route('/cookie')
def set_cookie():
    # Set a cookie named 'Name' with the value 'Evet'
    res = make_response("<h1>Cookie is set</h1>")
    res.set_cookie('Name', 'Evet')
    return res
//Get cookies
@app.route('/getcookie')
def get_cookie():
    # Access the value of the 'Name' cookie
    name = request.cookies.get('Name')
    return f"The name is: {name}"
//Delete cookies
@app.route('/deletecookie')
def delete_cookie():
    # Delete the 'Name' cookie by setting its expiration time to a past date
    res = make_response("<h1>Cookie is deleted</h1>")
    res.set_cookie('Name', '', expires=0)
    return res

if __name__ == '__main__':
    app.run(debug=True)
```

/cookie route sets a cookie named 'Name' with the value 'Evet'.
/getcookie route accesses the value of the 'Name' cookie.
/deletecookie route deletes the 'Name' cookie by setting its expiration time to a past date.

11) Explain request object properties with proper examples.

Attribute	Description
Form	Dictionary object with key-value pair of form parameters and their values.
args	Patr of the URL which is specified in the URL after question mark (?).
Cookies	It is the dictionary object containing cookie names and the values.
files	It contains the data related to the uploaded file.
method	It is the current request method (get or post).

The application contains three files, i.e., script.py, customer.html, and result_data.html.

script.py

```
from flask import *
app = Flask(__name__)
@app.route('/')
def customer():
  return render_template('customer.html')
@app.route('/success',methods = ['POST', 'GET'])
def print_data():
  if request.method == 'POST':
    result = request.form
    return render_template("result_data.html",result = result)
if __name__ == '__main__':
  app.run(debug = True)
```

customer.html: Form is created to input user data.

```
<html>
  <body>
    <h3>Register the customer, fill the following form.</h3>
    <form action = "http://localhost:5000/success" method = "POST">
      <p>Name <input type = "text" name = "name" /></p>
      <p>Email <input type = "email" name = "email" /></p>
      <p><input type = "submit" value = "submit" /></p>
    </form>
  </body>
</html>
```

result_data.html:

```
<!doctype html>
<html>
  <body>
    <table border = 1>
      {% for key, value in result.items() %}
        <tr>
          <th> {{ key }} </th>
          <td> {{ value }} </td>
        </tr>
      {% endfor %}
    </table>
  </body>
</html>
```

Output:

name	abc
email	abc@gmail.com

12) How to handle file uploading in flask?

Handling file uploading in Flask involves the following steps:

1. **Create Frontend:** Create an HTML file **index.html** where the user can select a file and upload it by clicking the upload buttons.

```
<html>
<head>
    <title>upload the file : GFG</title>
</head>
<body>
    <form action = "/success" method = "post" enctype="multipart/form-data">
            <input type="file" name="file" />
            <input type = "submit" value="Upload">
    </form>
</body>
</html>
```

2. Create another HTML file **response.html** to check the response.

```
<html>
<head>
    <title>success</title>
</head>
<body>
    <p>File uploaded successfully</p>
    <p>File Name: {{name}}</p>
</body>
</html>
```

3. Now create a Python file **fileupload.py** and write the following code.

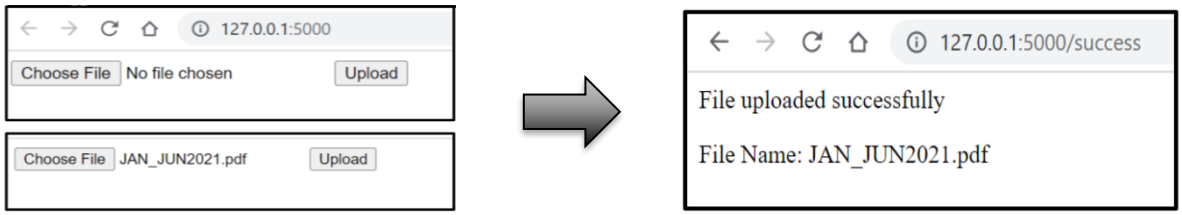
```
from flask import *
app = Flask(__name__)

@app.route('/')
def main():
        return render_template("index.html")

@app.route('/success', methods = ['POST'])
def success():
        if request.method == 'POST':
                f = request.files['file']
                f.save(f.filename)
                return render_template("response.html", name = f.filename)

if __name__ == '__main__':
        app.run(debug=True)
```

Output:



13) Explain Flask templates with example.

Flask templates are HTML files that include placeholders for dynamic content. These templates allow you to generate dynamic web pages by injecting data from your Python Flask application.

Example:

1. **Create a Flask Application**: First, create a Flask application in Python.

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html', title='Home', message='Welcome to Flask App!')

if __name__ == '__main__':
    app.run(debug=True)
```

2. **Create a Template**: Next, create an HTML template file in the **templates** directory of your Flask application. Let's name it **index.html**.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ message }}</h1>
</body>
</html>
```

3. **Render the Template**: In your Flask application, use the **render_template** function to render the HTML template and pass any dynamic data as keyword arguments.

```
from flask import render_template

@app.route('/')
def index():
    return render_template('index.html', title='Home', message='Welcome to Flask App!')
```

4. **Run the Application**: Run your Flask application. Run the command “python app.py”.

Now, when you visit the root URL of your Flask application (**http://localhost:5000**), Flask will render the **index.html** template and inject the dynamic data provided in the **render_template** function. In this example, the title will be "Home" and the message will be "Welcome to Flask App!"