5COSC022W.2 Client Server Architectures

Tutorial Week 02: Client-Server Architecture Design Principles

In this tutorial, you will be given some scenarios and asked to identify the best architectural design, such as 1-tier, 2-tier, 3-tier, and N-tier architectures. You will also design and implement a Java application to simulate a tiered architecture.

Requirements

- Client server architectures concepts
- NetBeans IDE 18 or above

Exercise 1: System analysis

"BidStorm" is an online auction platform that allows users to buy and sell a wide variety of items. They started with a simple website and a small user base, but BidStorm has experienced a surge in popularity, leading to rapid growth in users, listed items, and bids.

Now, the platform is struggling to keep up with the demand. Users complain about slow loading times, website crashes during peak bidding hours, and occasional errors in processing bids and payments. BidStorm is also concerned about data security, as it stores sensitive user information and financial details.

Activity one: Investigate the problem (Group work)

- Each group analyses BidStorm's situation and identifies the potential issues caused by increased traffic and data volume.
- They should consider factors like website performance, database load, security risks, and user experience.
- Groups should brainstorm questions they would ask BidStorm to gather more information about their current infrastructure, technology, and pain points.

Activity two: Analyse and Recommend (Group Work)

Based on your investigation, each group analyses the suitability of different architectural approaches for BidStorm:

- **Two-tier:** Is it sufficient for the growing demands? Why or why not?
- **Three-tier:** Would it address the performance and scalability issues? How?
- **N-tier:** Is this level of complexity necessary for BidStorm at this stage? What benefits would it offer?

- **Microservices:** Could this be a long-term solution for BidStorm's growth? What are the potential challenges?

Groups must justify their choice of architecture by explaining how it addresses BidStorm's specific needs and challenges.

Activity three: Report and Share (Group Work)

- Each group compiles a short report summarizing their analysis, proposed architecture, and justifications.
- You share your reports on a collaborative platform like Padlet, allowing other groups to view and comment on their work.

Activity four: Class Discussion (Group Work)

The class engages in a discussion, comparing different groups' recommendations and debating the pros and cons of each architectural approach in the context of BidStorm's needs.

Exercise 2: Lab Activity

This exercise explores implementing a simplified tiered architecture for processing user data, similar to a client-server model. Tiered architectures divide an application into distinct layers, each with a specific responsibility. This separation of concerns improves code organisation, maintainability, and scalability.

In this exercise, we'll focus on a three-tier structure:

1. **Presentation Tier (Client):** This tier is responsible for interacting with the user. In our case, the Main class is a simplified client, taking user input from the console.
2. **Application/Business Logic Tier (Server/Middleware):** This tier handles the application's core logic, including data validation and processing. The UserValidator and UserProcessor classes represent this tier. The UserValidator ensures that user data meets specific criteria (e.g., valid age). The UserProcessor handles the actual processing of user data, including validation and storage. This tier also handles concurrency and synchronisation.
3. **Data Tier (Database/Data Store):** This tier is responsible for persistent data storage. The UserDataStore class simulates a database or data store by holding user data in memory using an ArrayList.

Benefits of Tiered Architecture Demonstrated in this Exercise:

- **Separation of Concerns:** Each tier has a clear responsibility. The Main class handles user interaction, the UserValidator and UserProcessor handle business logic, and the UserDataStore handles data storage. This makes the code more modular and easier to understand and maintain.
- **Modularity:** Changes to one tier are less likely to affect other tiers. For example, if we want to change how user data is stored (e.g., switch from an ArrayList to a database), we only need to modify the UserDataStore class.
- **Scalability (Simulated):** While this example is simplified, the use of threads in the UserProcessor demonstrates how the application tier can handle multiple user requests concurrently, a key aspect of scalability in client-server applications.

Classes and Their Roles:

- **User:** Represents a user with a name and age (Data Transfer Object or DTO).
- **UserValidator:** Validates user data (Business Logic).
- **UserDataStore:** Stores user data (Data Tier).
- **UserProcessor:** Processes user data, including validation and storage, in a concurrent environment (Business Logic/Concurrency Handling).
- **Main:** Acts as a client, taking user input and initiating the processing (Presentation Tier).

Part 1: The User Class

What is a Class? A class is a blueprint that defines an object's properties (attributes) and functionalities (methods). It acts as a template for creating objects of that specific type.

Defining the User Class: Start by creating a file named User.java.

This line defines a public class named User. Public access allows other classes to interact with the User class.

ATTRIBUTES - DESCRIBING A USER

What are Attributes? Attributes represent the characteristics or properties of an object. In the context of a User class, attributes might be name, age, etc.

1. **Defining User Attributes:** Inside the User class definition, declare two private attributes:

- String name - This will store the user's name.
- int age - This will store the user's age.
- Using private restricts access to these attributes within the class itself.

METHODS - ACTIONS ON A USER

What are the Methods? Methods define the actions or functionalities that can be performed on an object. They allow us to access or change the values of the attributes and implement specific behaviour.

Constructor: The constructor is a special method with the same name as the class. It's responsible for initializing a new object's state when it's created.

Define a constructor for the User class that takes two arguments:

- String name - The user's name to be assigned.
- int age - The user's age to be assigned.
- Inside the constructor, use this.name and this.age to refer to the current object's attributes and assign the provided arguments to them.

GETTERS AND SETTERS - ACCESSING AND MODIFYING ATTRIBUTES

1. **Getter Methods:** Getter methods allow us to access the values of private attributes from outside the class. For each attribute, define a public method that returns its value.

Define two getter methods:

- public String getName() - Returns the user's name.
- public int getAge() - Returns the user's age.

2. **Setter Methods (Optional):** Setter methods allow us to modify the values of private attributes from outside the class. You can define setter methods for name and age if needed.

THE TOSTRING METHOD - REPRESENTING A USER AS A STRING

The toString method is a special method inherited from the Object class. It defines how an object should be represented as a string.

Override the toString method to return a formatted string representation of the User object.

Part 2: The UserValidator Class

What is Data Validation? Data validation is the process of ensuring that user-provided data adheres to specific rules. This helps prevent invalid data from entering the system and causing issues.

Creating the UserValidator Class: Create a new file named UserValidator.java. In this file, define a public class named UserValidator

VALIDATING USER AGE

Age Validation Method: Define a static method named isValidAge that takes an int age as input and returns a boolean value. This method will check if the provided age is valid.

IMPLEMENTING AGE VALIDATION

Inside the isValidAge method, add logic to check if the provided age is greater than zero. A user's age should be a positive value.

Part 3: The UserProcessor Class

This part introduces multithreading and user data processing with validation. You'll build a UserProcessor class that handles user data concurrently with proper synchronisation.

What is Multithreading? Multithreading allows a program to execute multiple parts (threads) concurrently. This can improve performance by utilising multiple CPU cores or handling tasks that don't require continuous processing.

Creating the UserProcessor Class: Create a new file named UserProcessor.java. Define a public class named UserProcessor that implements the Runnable interface. The Runnable interface signifies that objects of this class can be used in threads.

USERPROCESSOR PROPERTIES

Class Properties: The UserProcessor class holds information for processing a user:

- UserValidator validator - A reference to a UserValidator object for validation.
- UserDataStore dataStore - A reference to a UserDataStore object for storing user data.
- Object lock - An object used for synchronization (explained later).

- String name - The user's name.
- int age - The user's age.

PROCESSING USER INPUT CONCURRENTLY

1. **The run Method:** The run method is the entry point for each thread created from a UserProcessor object. It defines the processing logic.

```
public class UserProcessor implements Runnable {
  // ... (previous code)

  @Override
  public void run() {
    // Process user input concurrently with proper synchronization
  }
}
```

2. **Synchronisation:** Concurrent access to shared resources (like the dataStore) can lead to data corruption. Here, we use a synchronised block to ensure only one thread accesses the dataStore at a time:

```
public class UserProcessor implements Runnable {
  // ... (previous code)

  @Override
  public void run() {
    synchronized (lock) {
      if (validator.isValidAge(age)) {
        // Process the user input (e.g., store in data store)
        dataStore.addUser(new User(name, age));
      } else {
        // Display error message for invalid input
        System.out.println("Invalid age for User " + name);
      }
    }
  }
}
```

PART 4: THE USERDATASTORE CLASS

What is a Data Store? A data store is a component responsible for holding and managing data within an application. In this case, it will store User objects.

Creating the UserDataStore Class: Create a new file named UserDataStore.java. Define a public class named UserDataStore:

STORING USER DATA

1. **Using a List:** Create an ArrayList to store the User objects. An ArrayList is a dynamic array that can grow as needed.
2. **Declaring the List:** Inside the UserDataStore class, declare a private List<User> named userList
3. **Initializing the List in the Constructor:** In the UserDataStore's constructor, initialise the userList with a new ArrayList:

ADDING AND RETRIEVING USERS

The addUser Method: Create a public method named addUser that takes a User object as input and adds it to the userList:

The getAllUsers Method: Create a public method named getAllUsers that returns a copy of the userList. This is important to prevent external code from directly modifying the internal list of the UserDataStore:

PART 5: CONCURRENT USER INPUT PROCESSING

The core concept demonstrated in the Main class, using UserProcessor and threads, is **concurrent user input processing**. This means handling multiple user inputs simultaneously, improving responsiveness and efficiency, especially in scenarios with multiple users or long-running tasks.

2. How it Works in the Code:

- **UserProcessor:** The UserProcessor class implements the Runnable interface. This makes its instances executable by threads. The *run()* method of UserProcessor contains the logic for validating and storing a single user's data.
- **Creating Threads:** In the Main class, inside the input loop, a new Thread object is created for each user input: *Thread t = new Thread(processor);*. The processor (a UserProcessor instance) is passed to the Thread constructor, specifying what code the thread should execute.
- **Starting Threads:** The *t.start()* method starts the thread's execution. This makes the run() method of the associated UserProcessor execute in a separate thread of control.
- **The lock Object and Synchronisation:** Because multiple threads might try to access and modify the shared dataStore concurrently, we use a synchronized block. The synchronized (lock) statement ensures that only one thread can execute the code within the block at any given time. This prevents race conditions

and data corruption. The lock object acts as a monitor, allowing only one thread to "hold the lock" and access the critical section.

- **join() Method:** The t.join() method in the Main class after the input loop is essential. It makes the main thread *wait* for each of the created threads to finish their execution before proceeding. This is crucial because we want to ensure that all user data has been processed and stored before we attempt to display the contents of the dataStore. Without join(), the main thread might print the data before the other threads have had a chance to add it. Here is the tree structure of the class:

TREE STRUCTURE FOR MAIN CLASS

Please carefully follow this tree structure to implement the whole main class.

```
├── 1. Setting Up the class and implementing main method
|   ├── 1. Create a Scanner object to read user input
|   ├── 2. Create instances of UserValidator and UserDataStore
|   ├── 3. Create a List to store Thread objects
|   ├── 4. Create a lock object for synchronisation
|   └── 5. Initialise a boolean variable to control the input loop
├── 2. User Input Loop (while addMoreUsers)
|   ├── 7. Simulate concurrent user input (3 users at a time using for loop)
|   |   ├── 8. Prompt the user to enter a name
|   |   ├── 9. Read the name from the input
|   |   ├── 10. Prompt the user to enter an age
|   |   ├── 11. Read the age from the input
|   |   ├── 12. Create a UserProcessor object
|   |   ├── 13. Create a Thread object with the UserProcessor object
|   |   ├── 14. Add the thread to the thread list
|   |   └── 15. Consume the newline character left by nextInt()
|   ├── 16. Ask the user if they want to add more users
|   ├── 17. Read the user's response using nextLine()
|   └── 18. Update the loop control variable based on the response
```

├── 3. Thread Execution (for Thread t : threadList)

│   ├── 19. Iterate through the thread list and start/join each thread

│   │   ├── 20. Start the thread

│   │   └── 21. Wait for the thread to finish (join)

│   ├── 21.1 Handle potential InterruptedException

└── 4. Displaying Users

    ├── 22. Display all users stored in the data store

APPENDIX

MULTI-THREADING IN JAVA

Multithreading in Java refers to the concurrent execution of two or more threads within the same program, allowing different parts of the program to execute independently. A thread is a lightweight process that runs within the context of a larger program and shares the same resources, such as memory space, with other threads. Multithreading is a powerful concept that enables more efficient utilization of system resources and can lead to improved performance in certain scenarios.

Here's an overview of how multithreading works in Java, the essential components, and key concepts:

1. Thread Class:

- The primary component for multithreading in Java is the **Thread** class, which is part of the **java.lang** package.
- To create a new thread, you can either extend the **Thread** class and override its **run** method or implement the **Runnable** interface and pass an instance of your class to a **Thread** object.

```
class MyThread extends Thread {
 public void run() {
  // Code to be executed in the new thread
 }
}
```

OR

```
class MyRunnable implements Runnable {
  public void run() {
    // Code to be executed in the new thread
  }
}
// Create a Thread and pass an instance of MyRunnable
Thread thread = new Thread(new MyRunnable());
```

2. Thread Lifecycle:

- A thread goes through various states during its lifecycle, including **NEW**, **RUNNABLE**, **BLOCKED**, **WAITING**, **TIMED_WAITING**, and **TERMINATED**.
- The **start** method of the **Thread** class is used to begin the execution of a thread, transitioning it from the **NEW** to the **RUNNABLE** state.

3. Concurrency vs. Parallelism:

- Concurrency involves making progress on more than one task at the same time, but not necessarily simultaneously.
- Parallelism involves the simultaneous execution of multiple tasks.

4. Synchronization:

- Threads share the same memory space, which can lead to data inconsistency issues when multiple threads access and modify shared data simultaneously.
- Synchronization mechanisms, such as the **synchronized** keyword or explicit locks, are used to control access to shared resources and prevent data corruption.

5. Thread Safety:

- Ensuring thread safety is crucial when dealing with shared data. It involves designing code in a way that allows it to be safely accessed by multiple threads without causing data corruption or unexpected behavior.

6. Daemon Threads:

- Daemon threads are background threads that do not prevent the program from exiting if they are the only threads running.
- They are typically used for tasks like garbage collection or monitoring.

7. Thread Pools:

- Creating a new thread for every task can be inefficient. Thread pools, managed by the **Executor** framework, provide a reusable pool of worker threads that can be used to execute tasks.

8. Interrupts:

- The **interrupt** mechanism is used to interrupt the execution of a thread. It is commonly used for graceful thread termination.

9. Wait and Notify:

- The **wait** and **notify** methods, along with the **synchronized** keyword, are used for inter-thread communication and coordination.

Example of Multithreading in Java:

Here's a simple example to illustrate the basics of multithreading:

In this example, two threads (**thread1** and **thread2**) are created and started. The **run** method of **MyThread** will be executed concurrently by both threads, leading to interleaved output.

```
class MyThread extends Thread {
  public void run() {
    for (int i = 0; i < 5; i++) {
      System.out.println(Thread.currentThread().getId() + " Value " + i);
    }
  }
}

public class MultithreadingExample {
  public static void main(String[] args) {
    MyThread thread1 = new MyThread();
    MyThread thread2 = new MyThread();

    thread1.start();
    thread2.start();
  }
}
```

WHAT IS THREAD-SAFE LIST IN JAVA?

In Java, a thread-safe list refers to a data structure that is designed to be safely accessed and modified by multiple threads concurrently, without leading to data inconsistencies or race conditions. Java provides several thread-safe list implementations that ensure proper synchronization and integrity of data when accessed by multiple threads. Two common thread-safe list implementations in Java are:

1. **CopyOnWriteArrayList:**

- **CopyOnWriteArrayList** is a part of the Java Collections Framework and is a thread-safe variant of **ArrayList**.
- It achieves thread-safety by creating a new copy of the underlying array whenever the list is modified (e.g., adding or removing elements). This copy is then used for subsequent read operations.
- While it provides thread-safety, it might not be the best choice for scenarios where write operations are more frequent than read operations due to the overhead of copying the array.

```
import java.util.List;

import java.util.concurrent.CopyOnWriteArrayList;

List<String> threadSafeList = new CopyOnWriteArrayList<>();
```

2. **Collections.synchronizedList:**

- **Collections.synchronizedList** is a utility method provided by the **java.util.Collections** class to create a synchronized (thread-safe) wrapper around an existing list.
- It wraps the original list, providing synchronization on each method call, ensuring that only one thread can modify the list at a time.
- It's a good choice when the list needs to be accessed and modified by multiple threads, but keep in mind that while individual operations are synchronized, compound operations (like iteration) still require external synchronization.

```
import java.util.List;

import java.util.Collections;

import java.util.ArrayList;

List<String> originalList = new ArrayList<>();

List<String> threadSafeList = Collections.synchronizedList(originalList);
```

JAVA CUONCURRENCY

ExecutorService and **Executors** are part of the Java Concurrency Framework, introduced to simplify the process of managing and executing concurrent tasks. They provide a higher-level abstraction for working with threads compared to directly working with **Thread** objects.

EXECUTORS

Executors is a utility class in the **java.util.concurrent** package that provides factory and utility methods for creating instances of **ExecutorService**. It simplifies the process of creating different types of thread pools.

KEY METHODS:

1. **newFixedThreadPool(int nThreads):**
    - Creates a fixed-size thread pool where the number of threads remains constant.
2. **newCachedThreadPool():**

- Creates a thread pool that creates new threads as needed but reuses existing ones. Unused threads are terminated after a specified idle timeout.

3. **newSingleThreadExecutor():**
    - Creates a single-threaded executor that uses a single worker thread operating off an unbounded queue.
4. **newScheduledThreadPool(int corePoolSize):**
    - Creates a thread pool that can schedule commands to run after a given delay or at fixed intervals.
5. **newSingleThreadScheduledExecutor():**
    - Creates a single-threaded executor that can schedule commands to run after a given delay or at fixed intervals.

EXECUTORSERVICE

It is an interface in the **java.util.concurrent** package that represents an asynchronous execution service. It extends the **Executor** interface and provides more advanced features for managing and controlling the execution of tasks.

KEY METHODS:

1. **submit(Runnable task):**
    - Submits a Runnable task for execution and returns a **Future** representing that task.
2. **submit(Callable<T> task):**
    - Submits a Callable task for execution and returns a **Future** representing that task. The **Future** can be used to retrieve the result of the computation.
3. **shutdown():**
    - Initiates an orderly shutdown of the **ExecutorService**, allowing previously submitted tasks to complete. After shutdown, no new tasks will be accepted.
4. **shutdownNow():**
    - Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were waiting to be executed.
5. **invokeAll(Collection<? extends Callable<T>> tasks):**
    - Executes all tasks in the given collection, returning a list of **Future** objects representing the tasks.
6. **invokeAny(Collection<? extends Callable<T>> tasks):**

- Executes all tasks in the given collection and returns the result of the first successfully completed task (or throws an exception if no tasks complete successfully).