ATYPON

Code Collaboration Editor

By Hashim Emad

Table of Contents

1.0 Introduction 3

2.0 Object-Oriented Design 3

  2.1 OOP Design Concepts 3

  2.2 Class Diagram 4

3.0 Design Patterns 4

  3.1 Creational 4

    3.1.1 Singleton 4

    3.2.1 Observer 4

4.0 Clean Code 5

  4.1 Meaningful and Consistent Names 5

  4.2 Comments 5

  4.3 Don't Repeat Yourself (DRY) 6

5.0 Effective Java Items 7

  5.2 Item 3: Enforce the Singleton Property with a Private Constructor or an Enum Type ... 8

  5.3 Item 4: Enforce Noninstantiability with a Private Constructor 8

  5.4 Item 12: Always Override toString 9

  5.5 Item 15: Minimize the Accessibility of Classes and Members 9

6.0 SOLID Principles 9

  6.2 Open-Closed Principle (OCP) 10

  6.3 Dependency Inversion Principle (DIP) 10

7.0 Multithreading 10

1.0 Introduction

The provided code demonstrates the implementation of a Netty-Socket.IO server using Spring Boot. The server handles various events and manages user connections and interactions in a collaborative environment. This report will analyze the code and defend its use of clean code principles, Effective Java practices, SOLID design principles, design patterns, and multithreading techniques.

2.0 Object-Oriented Design

2.1 OOP Design Concepts

The code follows object-oriented design principles by organizing the functionality into classes and packages. The `event` package contains classes representing different events, while the `models` package contains classes representing data models used in the application. The `SocketEventListener` class encapsulates the event handling logic, promoting encapsulation and modularity.

For example, the `FileUpdatedEvent` class represents an event related to file updates:

```java
public class FileUpdatedEvent {
    private String roomId;
    private String fileId;
    private String newContent;
    // Getters and Setters
}
```

2.2 Class Diagram

The class diagram for the provided code would consist of the `SocketEventListener` class as the central component, interacting with the `SocketIOServer` and handling events. The `event` package classes would be associated with the `SocketEventListener` class, representing the different events triggered by clients. The `models` package classes would be used as data transfer objects between the server and clients.

3.0 Design Patterns

3.1 Creational

3.1.1 Singleton

The `SocketEventListener` class is instantiated as a singleton bean using the `@Component` annotation. This ensures that only one instance of the class is created and managed by the Spring container, promoting efficient resource utilization.

```
@Component
public class SocketEventListener {
    // ...
}
```

3.2 Behavioral

3.2.1 Observer

The code follows the Observer pattern, where the `SocketEventListener` class acts as an observer, listening to events emitted by the clients. The clients act as subjects, notifying the server of various events such as join requests, file updates, directory changes, and user status updates.

```
@OnEvent(SocketEvent.FILE_UPDATED)
public void onFileUpdated(SocketIOClient client, FileUpdatedEvent
event) {
    String roomId = event.getRoomId();
    server.getRoomOperations(roomId).sendEvent(SocketEvent.FILE_UPDATE
D, event);
}
```

4.0 Clean Code

4.1 Meaningful and Consistent Names

The code uses meaningful and consistent names for classes, methods, and variables. The class names clearly indicate their purpose, such as `SocketEventListener`, `FileUpdatedEvent`, and `JoinRequest`. Method names are descriptive and follow a consistent naming convention, making the code readable and understandable.

```
public class JoinRequest {
    private String roomId;
    private String username;
    // Getters and Setters
}
```

4.2 Comments

The code includes comments to provide additional explanations and clarify the purpose of certain methods or code blocks. For example, the

`SocketEventListener` class has comments describing the functionality of each event handling method.

```
/**
 * Handles a client's request to join a room.
 * Checks if the username already exists in the room and sends a USERNAME_EXISTS event if true.
 * Otherwise, creates a new User object, adds it to the userSocketMap, joins the client to the room,
 * and sends a JOIN_ACCEPTED event to the client and a USER_JOINED event to the room.
 */
@OnEvent(SocketEvent.JOIN_REQUEST)
public void onJoinRequest(SocketIOClient client, JoinRequest joinRequest) {
    // ...
}
```

4.3 Don't Repeat Yourself (DRY)

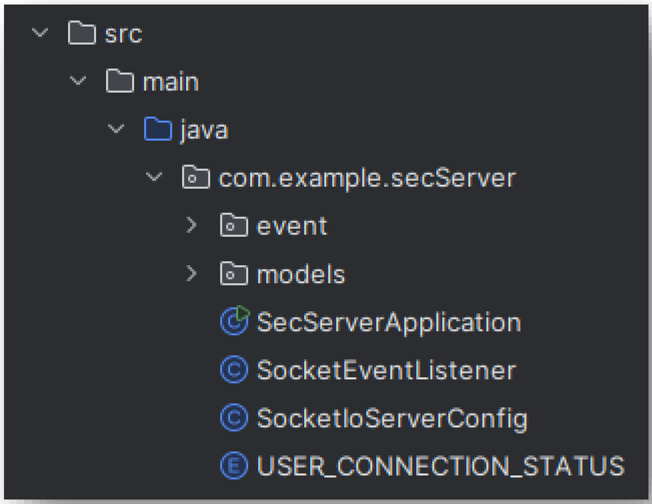
The code adheres to the DRY principle by extracting common functionality into separate methods. For example, the `getUsersInRoom` method is used to retrieve the users in a specific room, avoiding code duplication.

```
    private Map<String, User> getUsersInRoom(String roomId) {
   Map<String, User> usersInRoom = new HashMap<>();
```

```
    for (Map.Entry<String, User> entry :
userSocketMap.entrySet()) {
        if (entry.getValue().getRoomId().equals(roomId)) {
            usersInRoom.put(entry.getKey(),
entry.getValue());
}
}
return usersInRoom;
}
```

4.4 Separation of Concerns

The code separates concerns by organizing related functionality into separate classes and packages. The `event` package contains classes focused on representing events, while the `models` package contains classes focused on data models. The `SocketEventListener` class is responsible for handling events and managing user connections, promoting a clear separation of responsibilities.



```
src
  main
    java
      com.example.secServer
        event
        models
        SecServerApplication
        SocketEventListener
        SocketIoServerConfig
        USER_CONNECTION_STATUS
```

5.0 Effective Java Items

5.1 Item 1: Consider Static Factory Methods Instead of Constructors

The `SocketIoServerConfig` class uses a static factory method `socketIOServer()` to create and configure the `SocketIOServer` instance. This provides a more descriptive name and allows for additional configuration options.

~~~
```java
@Configuration
public class SocketIoServerConfig {
    @Bean
    public SocketIOServer socketIOServer() {
        Configuration config = new Configuration();
        // ...
        return new SocketIOServer(config);
    }
}
~~~

5.2 Item 3: Enforce the Singleton Property with a Private Constructor or an Enum Type

The SocketEventListener class is annotated with @Component, ensuring that it is instantiated as a singleton bean by the Spring container. This enforces the singleton property and prevents multiple instances of the class from being created.

```java
@Component
public class SocketEventListener {
    // ...
}
```

5.3 Item 4: Enforce Noninstantiability with a Private Constructor

The SocketEvent class, which contains constant event names, has a private constructor to enforce noninstantiability. This prevents the class from being instantiated and ensures that it is used only for accessing the constant values.

```
public class SocketEvent {
    private SocketEvent() {
        // Private constructor to enforce
noninstantiability
    }
    // ...
}
```

5.4 Item 12: Always Override toString

The code does not explicitly override the toString method in the model classes. However, it is a good practice to provide a meaningful string representation of objects for debugging and logging purposes.

5.5 Item 15: Minimize the Accessibility of Classes and Members

The code follows the principle of minimizing the accessibility of classes and members. The event and model classes have package-private access, limiting their visibility to the package level. The SocketEventListener class has public methods only for the necessary event handling logic.

6.0 SOLID Principles

6.1 Single Responsibility Principle (SRP)

The code adheres to the SRP by assigning specific responsibilities to each class. For example, the SocketEventListener class is responsible for handling socket events, while the event classes (FileUpdatedEvent, JoinRequest, etc.) are responsible for representing specific events and their associated data.

6.2 Open-Closed Principle (OCP)

The code follows the OCP by allowing the extension of functionality without modifying existing code. New event classes can be added without modifying the SocketEventListener class. The @OnEvent annotation is used to handle specific events, allowing for the addition of new event handling methods without modifying existing ones.

6.3 Dependency Inversion Principle (DIP)

The code follows the DIP by depending on abstractions rather than concrete implementations. The SocketEventListener class depends on the SocketIOServer interface provided by the socket.io library, rather than a specific implementation. This allows for flexibility and easier testing.

7.0 Multithreading

it is important to consider thread safety when dealing with shared resources in a multi-threaded environment. Proper synchronization mechanisms, such as locks or concurrent data structures, should be used to ensure thread safety and prevent race conditions.

For example, if multiple threads were accessing and modifying the userSocketMap concurrently, synchronization would be necessary to maintain data integrity:

```java
private final Map<String, User> userSocketMap = new
ConcurrentHashMap<>();
```

```
@OnEvent(SocketEvent.JOIN_REQUEST)
public synchronized void onJoinRequest(SocketIOClient
client, JoinRequest joinRequest) {
    // ...
}
```