# Distributed File System

```java
package csueb.cs401.group2;

public class DistributedFileSystem {

    private String student1 = "Travis Cassell";
    private String student2 = "Bryan Graves";
    private String student3 = "Michael Nguyen";
    private String student4 = "Quang Nguyen";
    private String student5 = "Andrew Nowinski";

    public DistributedFileSystem() {

        this.group = "Group 2";
        this.course = "CS 401";
        this.semester = "Spring 2022"

    }
}
```

# Quang - Intro/Requirement

Server

- Will authenticate users
- Will send requested file to client
- Will keep log of users who pushed a request
- Will interact with persistence module
- Keeps record of files and  where files are stored

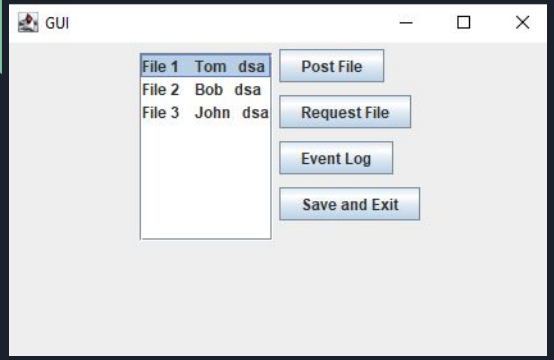
Client

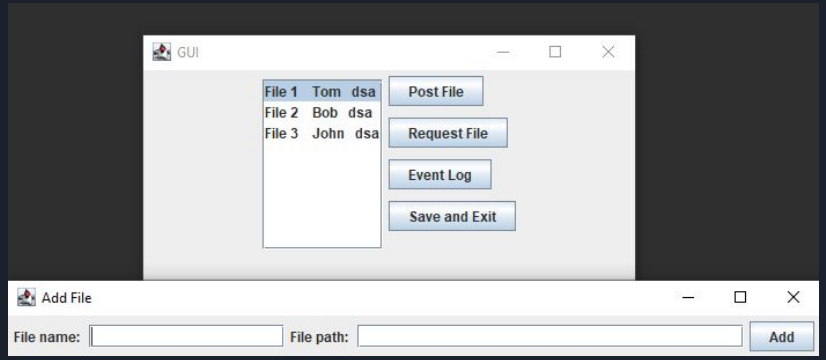
- Will be able to request files
- Will be able to look at list of files and choose
- Can upload file to node
- Can login using id and pin
- Provides GUI and interaction with Server

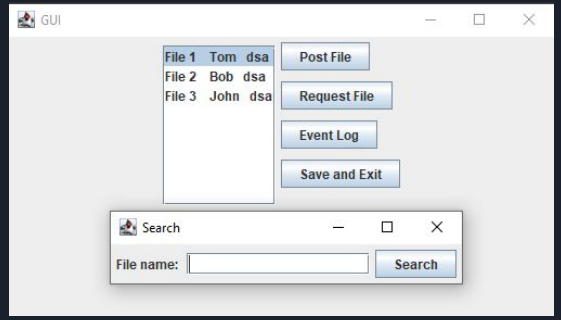
Persistence

- Logs all transactions
- Will persist log to file system
- Able to pull log
- Persists database to file system
- Maintains physical copy of metadata

# GUI


GUI
File 1 Tom dsa
File 2 Bob dsa
File 3 John dsa
Post File
Request File
Event Log
Save and Exit


GUI
File 1 Tom dsa
File 2 Bob dsa
File 3 John dsa
Post File
Request File
Event Log
Save and Exit
Add File
File name:
File path:
Add


GUI
File 1 Tom dsa
File 2 Bob dsa
File 3 John dsa
Post File
Request File
Event Log
Save and Exit
Search
File name:
Search

# Server Class

- Singleton pattern
- Holds shared data
  - Accessor methods for shared data is synchronized
  - Hashmaps because O(1)
- Holds list of services available

```java
 * The class holds a real time listing of active authenticated clients.
 *
 * The class holds a mapped distribution of file names with the associated nodes that
 * hold the files.
 *
 * The class holds a map of generated read file request tokens to the resolve state
 * of its response.
 */
public class Server {

    private static Server me;

    private ServerSocket socket;
    private int port;
    private int redundancyTarget; // target # of nodes that server will try to push file to

    private Logger LOGGER;
    private EventLog eventLog;

    private HashMap<String, User> registeredUsers = new HashMap<>();
    private HashMap<String, ClientHandler> activeAuthClients = new HashMap<>(); // userid to user
    private HashMap<String, List<FileNode>> fileDistribution = new HashMap<>(); // file name to FileNode
    private HashMap<String, Boolean> resolvedReadRequests = new HashMap<>(); // tokens to resolve state
    private Map<Message.Type, Service> services = Map.of(
            Message.Type.LOGIN, new ServiceLogin(),
            Message.Type.SEARCH, new ServiceSearch(),
            Message.Type.READ_FILE_REQUEST, new ServiceReadFileRequest(),
            Message.Type.READ_FILE_RESPONSE, new ServiceReadFileResponse(),
            Message.Type.POST_FILE_REQUEST, new ServicePostFileRequest(),
            Message.Type.POST_FILE_RESPONSE, new ServicePostFileResponse(),
            Message.Type.LOGS, new ServiceGetLogs()
        );

    private Server() {}

    public static Server getInstance() {
        if (me == null) {
            me = new Server();
            me.init();
        }
        return me;
    }
```

# Service Interface

```
public interface Service {


    /**
     * @return 0 if successful, any other number indicated failure
     */
    public int run(Message message, ClientHandler ref);
}
```

# Client Handler Class

```
Service service = Server.getInstance().getService(req.getType());
if (service != null) {
    try {
        int result = service.run(req, this);
        if (result != 0) {
```

- Decouple the invoker (client handler) with use case logic
- Can retrieve and handle the status output of the use case
- Much better than a bunch of if and else statements

# Service Login Class (Example)

- Isolated logic from other services
- Easy to create junit tests for each service

```java
/**
 * @author michaelvu
 * This class will receive a {@code Message} expecting a payload of type
 * {@code LoginBody} which requires
 *      1. username
 *      2. password
 *
 * This class will return a {@code Message} with no payload. The message will contain
 * the status, message, and timestamp for the request.
 *
 * It will create an id based on the concatenation of the username and password.
 * This id will be the key index.
 *
 * It will register a new user with the server if the credentials are not found
 * to existing map. Otherwise, it will use the server to retrieve the user details.
 * After either, the client handler is updated with user details and that client handler
 * is added to the map of active authenticated connections maintained in the server.
 *

 */
public class ServiceLogin implements Service{

    @Override
    public int run(Message message, ClientHandler ref) {

        LoginBody req = (LoginBody) message.getPayload();
        String id = req.getUsername().concat(req.getPassword());
        HashMap<String, User> registeredUsers = Server.getInstance().getRegisteredUsers();
        if (registeredUsers.containsKey(id)) {
            // found valid credentials
            // update client handler with user info
            ref.setAuthenticated(true);
            ref.setUser(registeredUsers.get(id));
        } else {
            // create a new user account since not found
            // update server with new user
            User user = new User();
            user.setId(id);
            user.setLoginUserName(req.getUsername());
            user.setLoginPassword(req.getPassword());

            // update server
```

# Client Class

Initialize port

Start function initializes socket, ObjectOutputStream, and ObjectInputStream

```java
private Client() {}

    public static Client getInstance() {
        if (me == null) {
            me = new Client();
            me.init();
        }
        return me;
    }

    private void init() {

        // init port
        if (System.getenv("port") != null) {
            port = Integer.valueOf(System.getenv("port"));
        } else {
            port = 8080;
        }

        // init logger
        System.setProperty("java.util.logging.SimpleFormatter.format",
                "[%1$tF %1$tT] [%4$-7s] %5$s %n");
        LOGGER = Logger.getLogger(Client.class.getName());
    }

    public static void start() {

        // Testing
        System.out.println("START RUNNING");

        try {

            socket = new Socket("localhost", port);
            oos = new ObjectOutputStream(socket.getOutputStream());
            ois = new ObjectInputStream(socket.getInputStream());

        } catch (IOException e) {
            LOGGER.warning(e.getLocalizedMessage());
        }
    }
```

# Client Functions

Post file request and response will upload file to specified owners node

Post file request causes the Client to request to push a File to the Server

Post file response notifies Client that the file has been processed

```java
public static class commands {
    // Post File Services

    public static void postFileRequest() {
        Message msg = new Message(Message.Type.POST_FILE_REQUEST);
        File file = new File(null, null);
        file.setContent(null);
        file.setFileName(null);

        try {
            oos.writeObject(msg);
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    public static void postFileResponse() {
        try {
            Message msg = (Message) ois.readObject();
            if (msg.getType().equals(Message.Type.POST_FILE_RESPONSE)) {
                File file = (File) msg.getPayload();
                // persisted and everything fine

                // Update to get type
                Message postMsg = new Message(Message.Type.POST_FILE_RESPONSE);

                FileNode fn = new FileNode();
                fn.setOwner(file.getOwner());
                fn.setFileName(file.getFileName());

                // update repo
                postMsg.setPayload(fn);

                oos.writeObject(postMsg);
            }
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }

    }
```

# Client Functions

Read file request and response will interact with server to read file from specified owners node

Read file request pushes a request to all nodes with the desired File

Read file response checks to see if the request has been handled, and if so the User will receive the requested File

```java
// Read File Services
public static void readFileRequest() {

    Message msg = new Message(Message.Type.READ_FILE_REQUEST);


    File file = new File(null, null);
    file.setContent(null);
    file.setFileName(searchFile.getText());

    try {
        oos.writeObject(msg);
    } catch (IOException e) {
        e.printStackTrace();
    }

}

public void readFileResponse() {
    try {

        Message msg = (Message) ois.readObject();
        if (msg.getType().equals(Message.Type.READ_FILE_RESPONSE)) {
            File file = (File) msg.getPayload();

            FileNode fn = new FileNode();
            fn.setOwner(file.getOwner());
            fn.setFileName(file.getFileName());

        }

    } catch (ClassNotFoundException | IOException e) {
        e.printStackTrace();
    }

}
```

# Client Functions

Login function that takes user input for username and password that are used to allow for User login

A payLoad containing the Users username and password is sent along with the login message

If User is verified, Users details are fetched

```java
public static void clientLogin() {
    try {
        Message msg = (Message) ois.readObject();
        if (msg.getType().equals(Message.Type.LOGIN)) {

            LoginBody req = (LoginBody) msg.getPayload();
            req.setUsername(txuser.getText());
            req.setPassword(pass.getText());

            msg.setPayload(req);

            oos.writeObject(msg);

        }
    } catch (ClassNotFoundException | IOException e) {
        e.printStackTrace();
    }
}
```

# Client Functions

EventLog function returns the event log as a string

```java
////////////////////////////////
// Request event log from server
// for display in GUI
////////////////////////////////
public static String eventLog() {
    // Send message type LOGS to server
    // to request event log
    Message msg = new Message(Message.Type.LOGS);

    try {
        oos.writeObject(msg);
    } catch (IOException e) {
        e.printStackTrace();
    }

    // Receive event log from server
    try {
        Message msgRec = (Message) ois.readObject();
        if(msgRec != null && msgRec.getMessage() != null) {

            return msgRec.getMessage();
        }

    } catch (ClassNotFoundException | IOException e) {
        e.printStackTrace();
    }

    return null;
}
```

# Client Functions

Search function returns a List of the File Nodes to allow the user to view a list of available files to choose from

```java
public static List<FileNode> search() {
    Message msg = new Message(Message.Type.SEARCH);
    Message msgRec = null;

    try {
        oos.writeObject(msg);
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    try {
        msgRec = (Message) ois.readObject();
        //
        // GET EVENT LOG
        //

    } catch (ClassNotFoundException | IOException e) {
        e.printStackTrace();
    }
    if (msgRec != null && msgRec.getPayload() != null && msgRec.getPayload() instanceof SearchBody){

        return ((SearchBody) msgRec.getPayload()).getNodes();
    }
    else {
        return null;
    }
}
```

# Client Functions

Save and Exit function exits the client and saves a copy of the eventlog

```
////////////////////////////////////
// Save event log pulled from
// server and exit program
////////////////////////////////////
public void saveAndExit() {
    String eLog = eventLog();
    try {

        PrintWriter outputFile = new PrintWriter("EventLog");
        outputFile.print(eLog);
        outputFile.close();

        } catch (Exception e) {

            e.printStackTrace();

        }

    System.exit(0);
}
```

# Event Class

Event Class:

A simple class to create Event objects when interactions take place between Client and Server

Attributes include Date object, a String description, and a String for user ID. These are stored to keep track of relevant interactions within the system.

Event class applied within the  persistence package.

```java
import java.util.Date;

public class Event {

    private Date eventDate;
    private String eventDescription;
    private String userID;



    //*********************************************************
    // Default constructor
    //*********************************************************
    public Event() {
        eventDate = new Date();
        eventDescription = new String(original: "");
        userID = new String(original: "");
    }



    //*********************************************************
    // Creates an event with the description and user given
    // Input: string description of event and string userID
    // Output: an event set to current date and time with
    // values specified.
    //*********************************************************
    public Event(String inDescription, String inUser) {
        eventDate = new Date();
        this.eventDescription = inDescription;
        this.userID = inUser;
    }



    //*********************************************************
    // Creates an event with the description and user given
    // Input: string description of event and string userID
    // Output: bool weather event was created and an event
    // set to current date and time with values specified.
    //*********************************************************
    public boolean createEvent(String inDescription, String inUser) {
        eventDate = new Date();
        this.eventDescription = inDescription;
```

# Event Log | | Persistence

Persist Package:

EventLog class:

Attributes include a List of events and a String filename used to record interactions between Client and Server. When an event occurs (such as a request to the Server for a file), an event object is created which is stored in a list of events. If the List exists, the object is added, otherwise a List is created. The list is written to a text file.

File updated with each added event.

Load data reads event log text file and returns List of events

```java
public class EventLog {

        private Event event = new Event();;
        private List<Event> events = new LinkedList<>();
        private String filename = "Events.txt";



        //*******************************************************
        // Default constructor
        //*******************************************************
        public void EventLog() {
        }



        //*******************************************************
        // Author: Travis Cassell
        // Saves the event log to a system file
        // Input: none
        // Output: file filename with all events written to OS
        //*******************************************************
        private void Save() {
            if(this.events == null) {
                this.events = new LinkedList<Event>();
                events.add(event);
            }

            else {
                events.add(event);
                doSaveToFile(event);
            }

        }

```

# Bryan - Persistence/Close Out

Persistence log file

Report of Events

Each module has been unit tested

Left to do:

Integration testing of the various modules

Performance testing

User acceptance testing

Events.txt - Notepad

File Edit View

```
0,Added file Financial Results.xlsx,gravesb,2022-05-02T01:03:07.254Z
1,Retrieved file Financial Results.xlsx,gravesb,2022-05-02T01:03:07.364Z
2,Added file Financial Results rev2.xlsx,gravesb,2022-05-02T01:03:07.365Z
```

```
0: Desc:Added file Financial Results.xlsx, User:gravesb, Date:01-5-2022 06:03:07
1: Desc:Retrieved file Financial Results.xlsx, User:gravesb, Date:01-5-2022 06:03:07
2: Desc:Added file Financial Results rev2.xlsx, User:gravesb, Date:01-5-2022 06:03:07
```