LibPrint
Design Specifications

by Chandler Griscom
Josh Powell
Dillon Hartsfield

presented to Dr. Brent Baas
COSC3403
24 Feb 2017

Table of Contents

1. Introduction

This document contains the initial design specifications for LibPrint. It is intended to provide a comprehensive basis and reference that developers may use to implement the final product.

LibPrint is a pair of software systems intended to act as a "middleman" between users attempting to print a document and staff members who will request payment and confirm the print action. On the client side, the software should act as a printer driver so that users will simply have to select LibPrint as their printer and click print. The document will then be sent to the host-operated webclient where a staff member can print the document normally.

2. Design Considerations

This section describes the issues encountered during the design process that resulted from unclear requirements or technical limitations.

2.1. Assumptions and Dependencies

- Windows 10 is the assumed operating system used by the library. This implies the following assumption:

  qvPDF must be compatible with the library's computers. IT will authorize the use of the custom print drivers and administrator access needed to set up the software.

- Requirements did not specify any detail regarding the network connections of the print server. The primary proposed solution assumes:
  - IT will create the LAN virtualhost called "libprint.letu.edu" for universal access
  - The computer running the print server is physically connected to the printers
  - The computer running the print server will have a static LAN IP address attached to the virtual host

2.2. Goals and Guidelines

- The printer interface must be as easy to use as a standard printer.
- LibPrint must improve upon the existing library practice.
- LibPrint must maintain tight security and ensure that system components may only be accessed by their intended users.

3. System Architecture and Design Strategies

The LibPrint architecture is divided into three segments: the printer driver, the client application, and the print server. Two communication channels are needed to facilitate data transfer between the segments: the driver / client file pipeline, and the client to server communication protocol.

3.1. Components

3.1.1. Printer Driver

To emulate standard printer behavior, the LibPrint execution sequence will be initiated by printing to a virtual Windows printer installed on the library computers. Due to the complexity of creating a printer driver from scratch, this program will simply reuse and configure the GPL-licensed qvPDF driver. It is capable of hiding configuration options from normal users, and can silently write PDF files and transfer execution to another program.

3.1.2. Client Application

Programming Language: Visual Studio C#
The client application will exist as a script-like executable on each library computer. It is executed by the printer driver when a user clicks 'Print' and must perform the following functions:

- Check if the remote print queue is full
- Display a message to the user containing their options and prices for printing (color vs. b&w)
- Display an error message if the remote queue is full
- Send authentication and the PDF document produced by the printer driver

3.1.3. Web Server

Environment: Apache Tomcat 8.0 (Java)
The web server will be accessed by the client application to communicate queue-related information, and to send PDF files to be printed. It will also be accessed through a web browser by library staff. It must perform the following functions:

- Respond to client application's requests for information
- Receive PDF files from the client applications
- Allow library staff to log in
- Maintain a queue of documents; allow the library staff to manage queue
- Distinguish between library public patrons and students, such that pricing may be accurately reported

3.2. *Communication Channels*

3.2.1. *Driver / Client File Pipeline*

A folder containing printed PDFs will be configured in qvPDF. The client application will be aware of this folder, and will retrieve PDFs from this folder if the remote print server gives it permission to send them. The folder should be emptied when the client exits so that files do not pile up on the client computer.

3.2.2. *Client to Server Protocol*

This protocol consists of two different requests and corresponding responses. Responses and requests are sent in plain text through HTTP POST requests initiated by the client. The "getInformation" request returns information about the queue and types of printing available. The "printPDF" request shall be used to actually send the PDF. Both requests will receive responses including the requested information, whether or not the operation was successful, and possibly an error message.

POST Request Format:

`POST Arguments:`	
`secToken={STRING}`	An encrypted token to verify that the source is legitimate
`request={STRING}`	"getInformation" or "printPDF"
`username={STRING}`	example: "chandlergriscom" or "LibraryGuest1"
`computer={STRING}`	The library computer's name, such as "S1".
`filename={STRING}`	Name of exported PDF file
`pages={INTEGER}`	Number of PDF pages (for calculating total cost)
`printerName={STRING}`	The printer's name (given by getInformation response)
`POST Body: {DATA}`	Empty for getInformation, or the binary file for printPDF

POST Response Format – A plaintext listing of response parameters

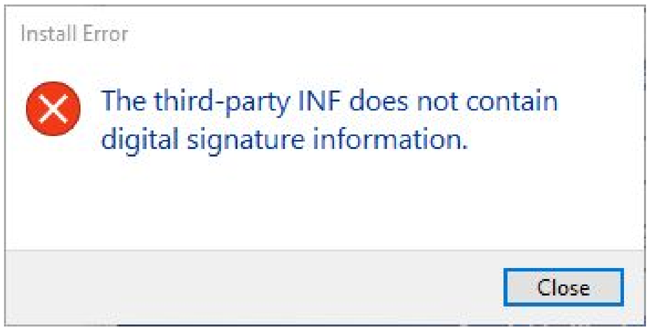
`Response: {STRING}`	Either "OK" or "Error"
`Error: {STRING}`	An error message that will be displayed to the user
`Printer0: {STRING}, {STRING}`	A printer and the cost, comma delimited. For example: "Color, $0.50 per page"
`Printer1: {STRING}, {STRING}`	Another name, price: such as "B&W, free"
`TotalCost: {STRING}`	The cost to be displayed to the user (pages times cost)

4. Policies and Tactics

LibPrint is designed to serve a large number of computers at the Margaret Estes Library. As such, special considerations were taken into account when designing the client and server components. These considerations include specifics regarding installation of the printer drivers and client software, configuration of the web server and library administrator access, and the type of hosting required to permit local access to all library computers.

4.1. Printer Driver Signature Verification on Windows 8-10

This policy is specific to the Windows 10 desktop computers used within the library. Newer Windows UEFI systems may prevent drivers from being installed which omit signatures from registered third-parties. Driver signing is a feature generally unavailable to student and open source developers, which presents a problem for Windows 8+ packaging. In order to install the printer driver that ships with qvPDF (and custom drivers in general), driver signature verification must be disabled on Windows 10. If the pictured error dialog box appears during driver installation, disabling signature enforcement is required before proceeding. On some computers, this may be as simple as running the following command as an administrator:



The third-party INF does not contain digital signature information.

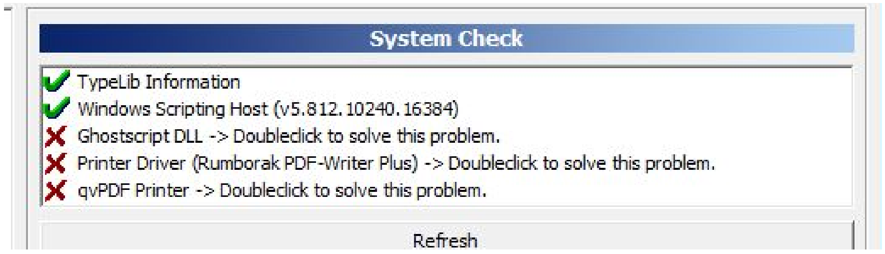
```
bcdedit /set testsigning on
```

On other systems, modifying UEFI boot options is necessary. This is a detail that must be worked out with LETU IT, as there is no way around it on some systems.

4.2. qvPDF Configuration

qvPDF contains a large feature set designed to allow administrators to configure printer drivers to hide unnecessary detail from users and execute the necessary software to process print commands. This section details all the steps needed to configure LibPrint for a library computer.

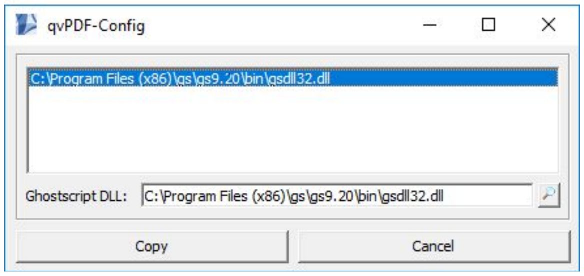
- Download and install Ghostscript x86 9.20 from the following location: https://github.com/ArtifexSoftware/ghostpdl-downloads/releases/download/gs920/gs920w32.exe
- Download and install qvPDF x86 (using all default options) from the following location: https://sourceforge.net/projects/qvpdf/files/latest/download
- Configure qvPDF using the configuration dialog that appears installation.



System Check

- ✓ TypeLib Information
- ✓ Windows Scripting Host (v5.812.10240.16384)
- ✗ Ghostscript DLL -> Doubleclick to solve this problem.
- ✗ Printer Driver (Rumborak PDF-Writer Plus) -> Doubleclick to solve this problem.
- ✗ qvPDF Printer -> Doubleclick to solve this problem.

Double click the Ghostscript list item to locate the necessary DLL file. The dialog below should automatically locate the file; click ‘copy’.

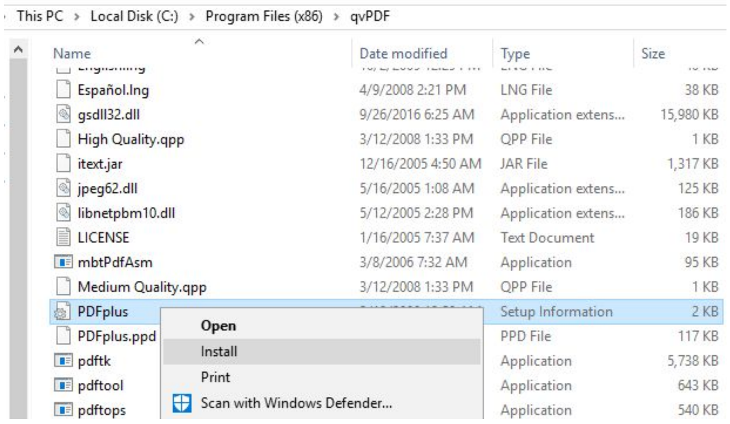


qvPDF-Config

C:\Program Files (x86)\gs\gs9.20\bin\gsdll32.dll

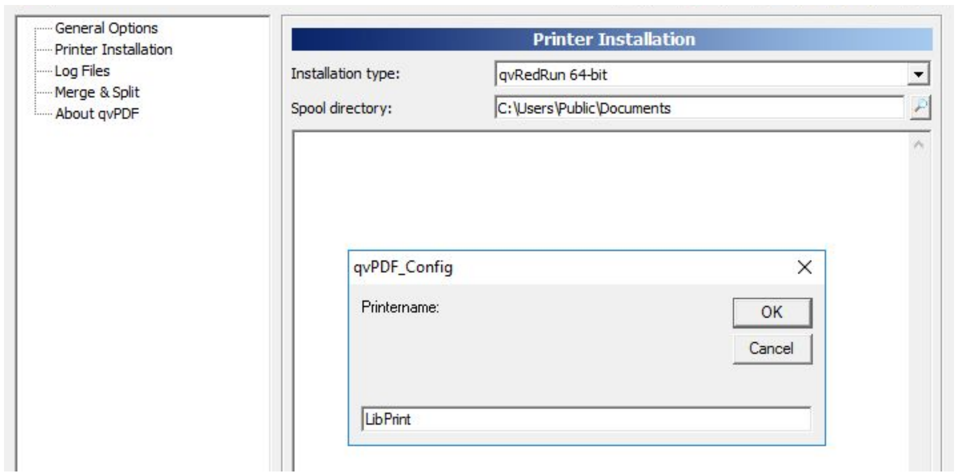
Ghostscript DLL: C:\Program Files (x86)\gs\gs9.20\bin\gsdll32.dll

Before double clicking the printer driver item, ensure that Windows 10 has allowed the driver to be installed. Navigate to the qvPDF installation directory, find the file named ‘PDFplus.inf’, right click it, and select ‘Install’. If the installation completes successfully, return to the qvPDF configuration window and double click the Printer Driver option. Otherwise, refer to the section 5.1 for disabling driver signature enforcement.



This PC › Local Disk (C:) › Program Files (x86) › qvPDF

Name	Date modified	Type	Size
Español.lng	4/9/2008 2:21 PM	LNG File	38 KB
gsdll32.dll	9/26/2016 6:25 AM	Application extens...	15,980 KB
High Quality.qpp	3/12/2008 1:33 PM	QPP File	1 KB
itext.jar	12/16/2005 4:50 AM	JAR File	1,317 KB
jpeg62.dll	5/16/2005 1:08 AM	Application extens...	125 KB
libnetpbm10.dll	5/12/2005 2:28 PM	Application extens...	186 KB
LICENSE	1/16/2005 7:37 AM	Text Document	19 KB
mbtPdfAsm	3/8/2006 7:32 AM	Application	95 KB
Medium Quality.qpp	3/12/2008 1:33 PM	QPP File	1 KB
PDFplus		Setup Information	2 KB
PDFplus.ppd		PPD File	117 KB
pdftk		Application	5,738 KB
pdftool		Application	643 KB
pdftops		Application	540 KB



Printer Installation

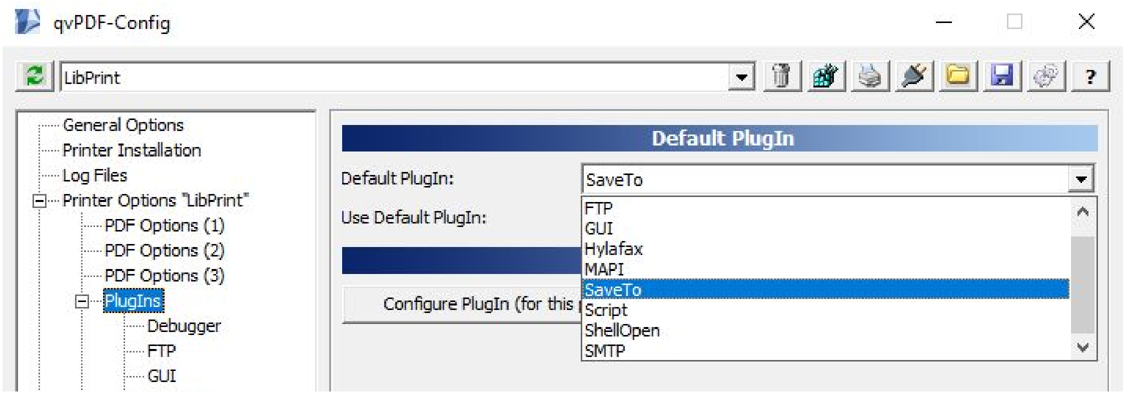
Installation type: qvRedRun 64-bit

Spool directory: C:\Users\Public\Documents

qvPDF_Config

Printername: LibPrint

Next, double click the ‘qvPDF Printer’ list item. Name the printer appropriately, then the application should automatically add the printer driver to Windows.


qvPDF-Config

LibPrint

- General Options
- Printer Installation
- Log Files
- Printer Options "LibPrint"
  - PDF Options (1)
  - PDF Options (2)
  - PDF Options (3)
  - PlugIns
    - Debugger
    - FTP
    - GUI

Default PlugIn

Default PlugIn: SaveTo

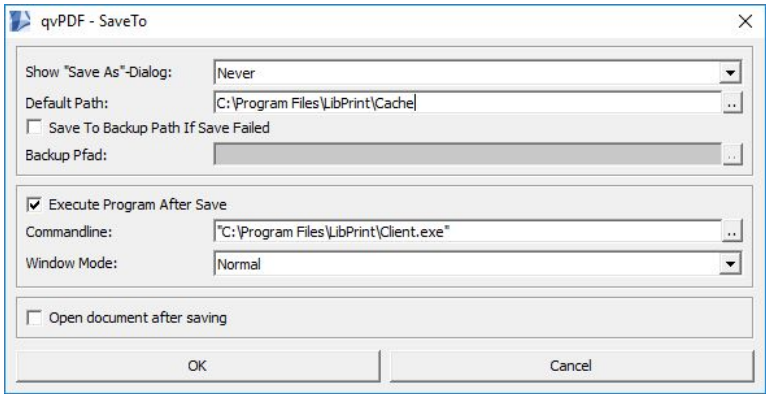
Use Default PlugIn:

FTP, GUI, Hylafax, MAPI, SaveTo, Script, ShellOpen, SMTP

Configure PlugIn (for this

In the list to the left, select 'PlugIns' and change the default plugin to 'SaveTo'. Change the 'Use Default PlugIn' option to 'Always.'

Finally, press 'Configure PlugIn' and a dialog will appear to configure the SaveTo plugin. It should be configured as pictured; 'Default Path' should be set to LibPrint's cache directory, and it should be set to execute the LibPrint client executable. The cache directory will contain exported PDFs, and after export, Client.exe will be executed.


qvPDF - SaveTo

Show "Save As"-Dialog: Never

Default Path: C:\Program Files\LibPrint\Cache

☐ Save To Backup Path If Save Failed

Backup Pfad:

☑ Execute Program After Save

Commandline: "C:\Program Files\LibPrint\Client.exe"

Window Mode: Normal

☐ Open document after saving

OK Cancel

4.3. Web Server Access Policies

The web server will need to run on the computer that is physically connected to the printers, yet also be accessible using a fixed HTTP address such as 'libprint.letu.edu'. There are a few ways to go about this, some of which must be left up to LETU IT. A good initial proposal that may need to be modified based on technical details of LETU's LAN would be to set up a locally accessible virtual host through Letnet.

In this configuration, the desktop computer serving as the print server must be configured with a fixed local IP address, and assigned the 'libprint.letu.edu' virtual host in the school's top-level domain configuration. It should be closed off to visitors that are not on LETU's local area network. This way, library administrators and patron computers may all access the LibPrint server, and it will still be blocked from external attacks. All access may be logged for security.

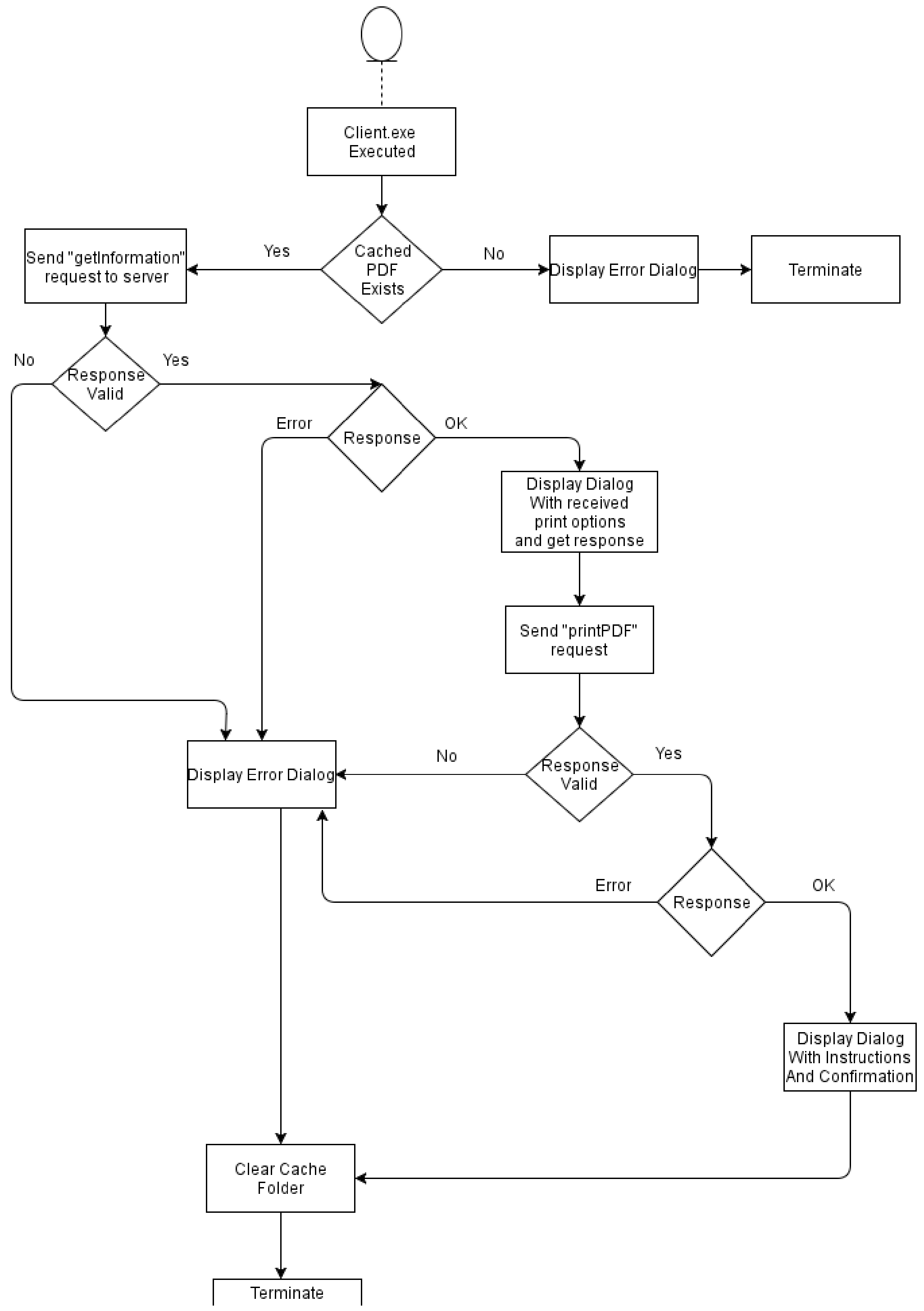
If this type of hosting is not possible, alternatives to investigate in the future would be:

- Use myfiles.letu.edu as an intermediary for file caching, then have the print server periodically poll the cache directory for submissions and respond appropriately.
- Let the library staff download the PDF and print it manually.

5. Detailed Component Descriptions

5.1. *Client Application*

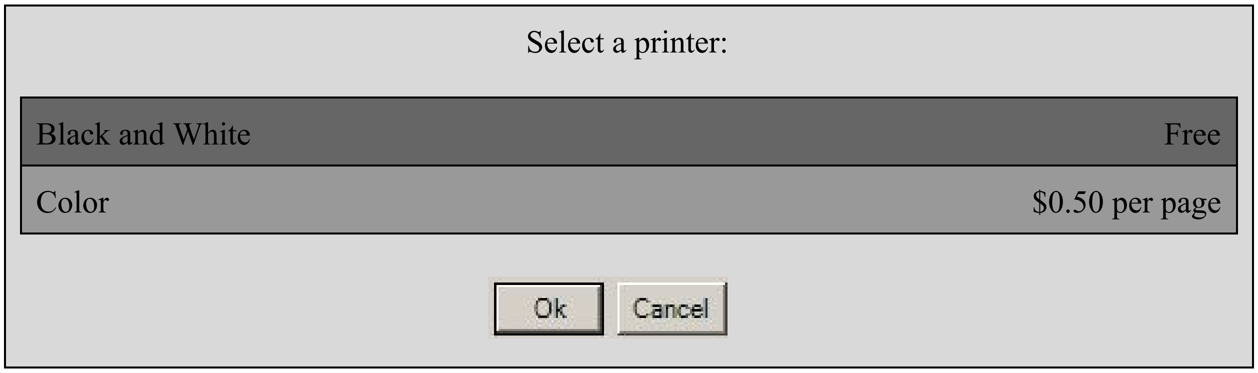
5.1.1. Flowchart



5.1.2. Dialog Descriptions

<u>*Print Options Dialog:*</u>

This dialog appears after the client has received a response following a getInformation request. If the response indicates OK to print, it parses the Option entries in the response to get a list of printer options and prices. These will be displayed in a list where the user is prompted to select which printer he wants.

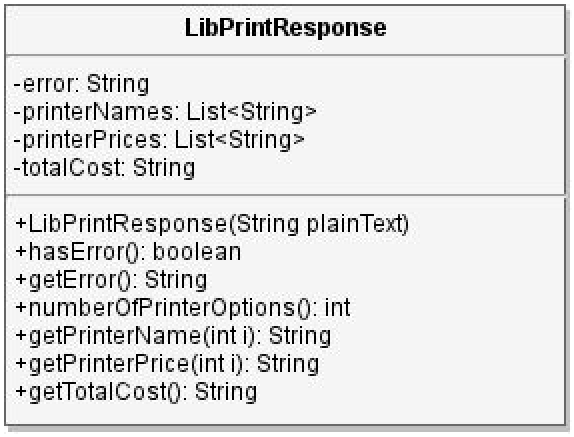


Select a printer:

Black and White	Free
Color	$0.50 per page

5.1.3. Class Diagrams

The LibPrintResponse class is a container that parses server responses and allows the application to easily access the returned data.



```
LibPrintResponse

-error: String
-printerNames: List<String>
-printerPrices: List<String>
-totalCost: String

+LibPrintResponse(String plainText)
+hasError(): boolean
+getError(): String
+numberOfPrinterOptions(): int
+getPrinterName(int i): String
+getPrinterPrice(int i): String
+getTotalCost(): String
```

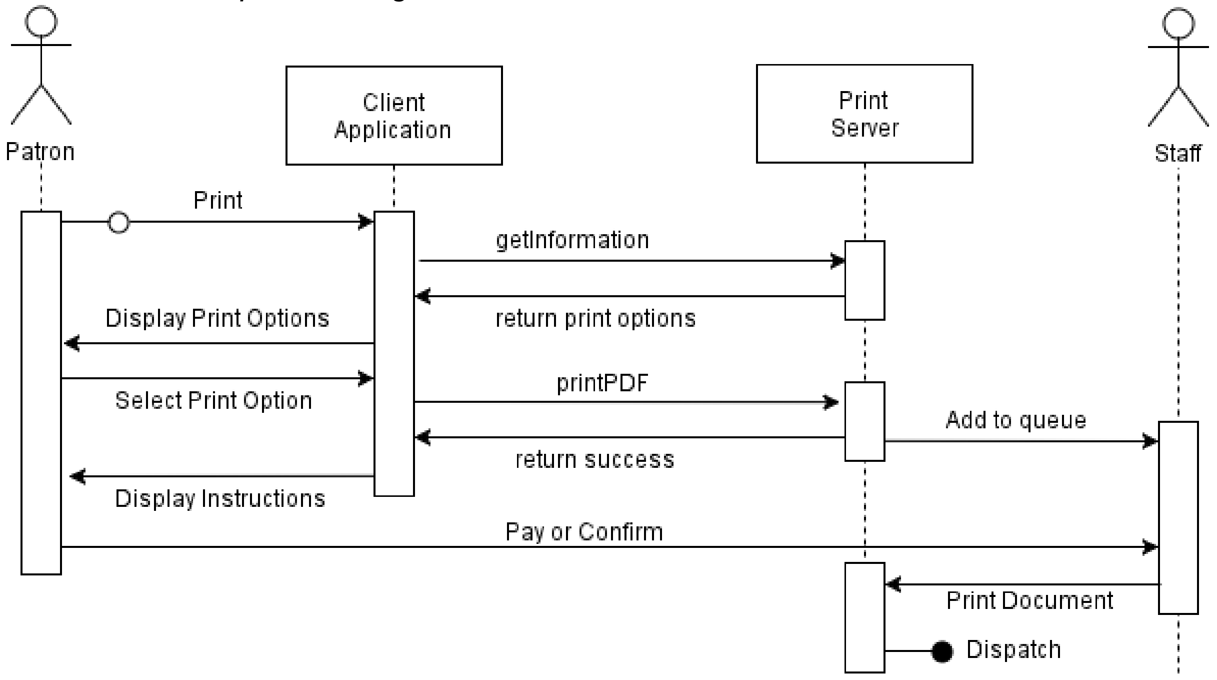
5.1.4. Pseudocode

The following pseudocode describes the basic operation of the client application. One omitted detail, the generateToken() method, may be implemented in different ways depending on the features available in Visual Studio, but it should generate an encrypted or hashed string calculated from the computer name, username, and filename, that the server will be able to independently verify as a security feature.

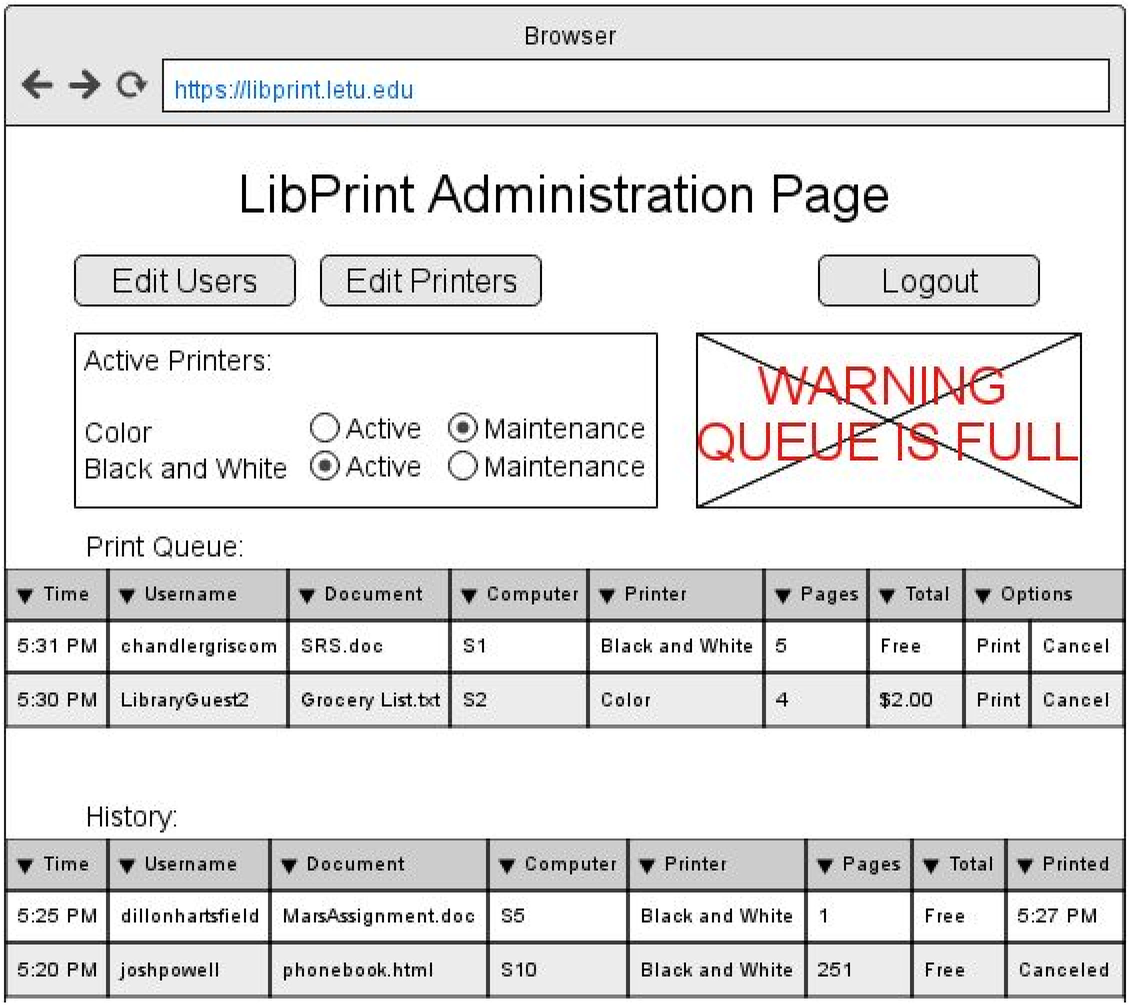
```
method init() // Runs as soon as program is started
  filename = null
  nFiles = 0
  for each file in listFilesInCacheDirectory() // Retrieve the filename
    filename = file.name()
    nFiles++
  end for
  if (nFiles != 1) // The cache directory should contain exactly 1 file
    displayErrorDialog(“An unexpected error occurred.  Please try again.”)
    clearCacheDirectory()
    quit()
  end if
  responseRaw = sendHTTP_POST(“libprint.letu.edu/RequestHandler”, null, // No body
                mapPostArguments(request -> “getInformation”,
                secToken -> generateToken(), username -> getLoggedInUser(),
                computer -> getComputerName(), filename -> filename
                pages -> getNumberOfPDFPages(filename)))
  if !validResponse(responseRaw) then
    displayErrorDialog(“Could not contact the print server...”)
    clearCacheDirectory()
    quit()
  end if
  responseObject = new LibPrintResponse(responseRaw) // Construct object
  if (responseObject.hasError()) then
    displayErrorDialog(responseObject.getErrorMessage())
    clearCacheDirectory()
    quit()
  end if

  int printerID = displayPrintOptionsDialog(responseObject)

  responseRaw = sendHTTP_POST(“libprint.letu.edu/RequestHandler”,
                getStreamFromFile(filename), // Body of request is the PDF
                mapPostArguments(request -> “printPDF”,
                secToken -> generateToken(), username -> getLoggedInUser(),
                computer -> getComputerName(), filename -> filename
                pages -> getNumberOfPDFPages(filename),
                printerID -> printerID))

  if !validResponse(responseRaw) then
    displayErrorDialog(“Could not contact the print server...”)
    clearCacheDirectory()
    quit()
  end if
  responseObject = new LibPrintResponse(responseRaw)

  // Check for errors again
  // Print server queue could still theoretically fill up at this point
  if (responseObject.hasError()) then
    displayErrorDialog(responseObject.getErrorMessage())
    clearCacheDirectory()
    quit()
  end if

  displayInfoDialog("Success! Go to the front desk to confirm and pay...")
  clearCacheDirectory()
  quit()
end method
```

5.2. Print Server

5.2.1. Sequence Diagram for a Successful Print



5.2.2. GUI Mockup



LibPrint Administration Page

Active Printers:

Color	Active	[✓ Maintenance]
Black and White	[✓ Active]	Maintenance

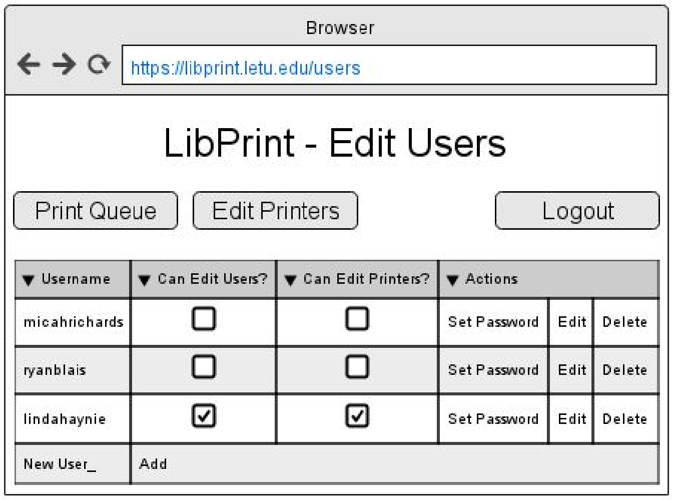
WARNING QUEUE IS FULL

Print Queue:

▼ Time	▼ Username	▼ Document	▼ Computer	▼ Printer	▼ Pages	▼ Total	▼ Options	
5:31 PM	chandlergriscom	SRS.doc	S1	Black and White	5	Free	Print	Cancel
5:30 PM	LibraryGuest2	Grocery List.txt	S2	Color	4	$2.00	Print	Cancel

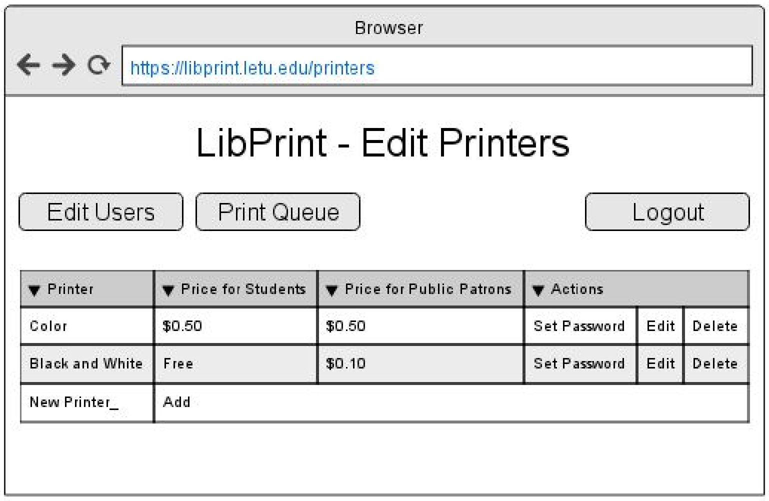
History:

▼ Time	▼ Username	▼ Document	▼ Computer	▼ Printer	▼ Pages	▼ Total	▼ Printed
5:25 PM	dillonhartsfield	MarsAssignment.doc	S5	Black and White	1	Free	5:27 PM
5:20 PM	joshpowell	phonebook.html	S10	Black and White	251	Free	Canceled



LibPrint - Edit Users

▼ Username	▼ Can Edit Users?	▼ Can Edit Printers?	▼ Actions		
micahrichards	☐	☐	Set Password	Edit	Delete
ryanblais	☐	☐	Set Password	Edit	Delete
lindahaynie	☑	☑	Set Password	Edit	Delete
New User_	Add				



LibPrint - Edit Printers

▼ Printer	▼ Price for Students	▼ Price for Public Patrons	▼ Actions		
Color	$0.50	$0.50	Set Password	Edit	Delete
Black and White	Free	$0.10	Set Password	Edit	Delete
New Printer_	Add				

5.2.3. *Web Interface Details*

As stated in section 3, Apache Tomcat will be used to implement the web server. Tomcat allows for authenticated remote administration and deployment, which will simplify the testing and maintenance processes.

A lightweight persistent database will be needed to store user and printer details. For simplicity (again), this project will use a Java serializing database called PropertyDB (https://github.com/cjgriscom/PropertyDB). It requires no external services to run on the host OS, can store its metadata within the Windows application data folder, and runs unobtrusively by periodically synchronizing selected Java objects to the file system whenever they are modified.

The web interface will be written using JSP, HTML 5, jQuery, and CSS. The jQuery components will be used to allow the administrator's browser to periodically poll the server to check if the print queue and history have updated.

5.2.4. *Java, Pseudocode, and Request Handling Descriptions*

PropertyDB Initialization Sequence for Tomcat Servlet (Java)

```
package edu.letu.libprint;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

import com.quirkygaming.errorlib.ErrorHandler;
import com.quirkygaming.propertydb.InitializationToken;
import com.quirkygaming.propertydb.PropertyDB;

/* Application Lifecycle Listener Class */
@WebListener
public class PropertyDBListener implements ServletContextListener {
        private static int PERIOD = 1000; // Sync every second if objects were modified
        private static InitializationToken token = null;

        // For testing purposes, throw all exceptions. This should be changed for formal release.
        // ErrorHandler.logAll may be used to direct errors to an arbitrary stream.
        public static ErrorHandler<RuntimeException> pdb_handler = ErrorHandler.throwAll();

        // This runs once on startup; initialize PropertyDB
        public void contextInitialized(ServletContextEvent c) {
                if (PropertyDB.initialized()) {
                        throw new IllegalStateException(
                                "PropertyDB has already been initialized!");
                } else {
                        // Initialize DB with given I/O-cycle time
                        token = PropertyDB.initializeDB(PERIOD);
                        Database.init(); // Initialize database properties
                }
        }
        // When server is closing, exit PropertyDB (ensures everything is cleaned up)
        public void contextDestroyed(ServletContextEvent c) {
                if (token != null) {
                        PropertyDB.closeDatabase(token);
                }
        }
}
```

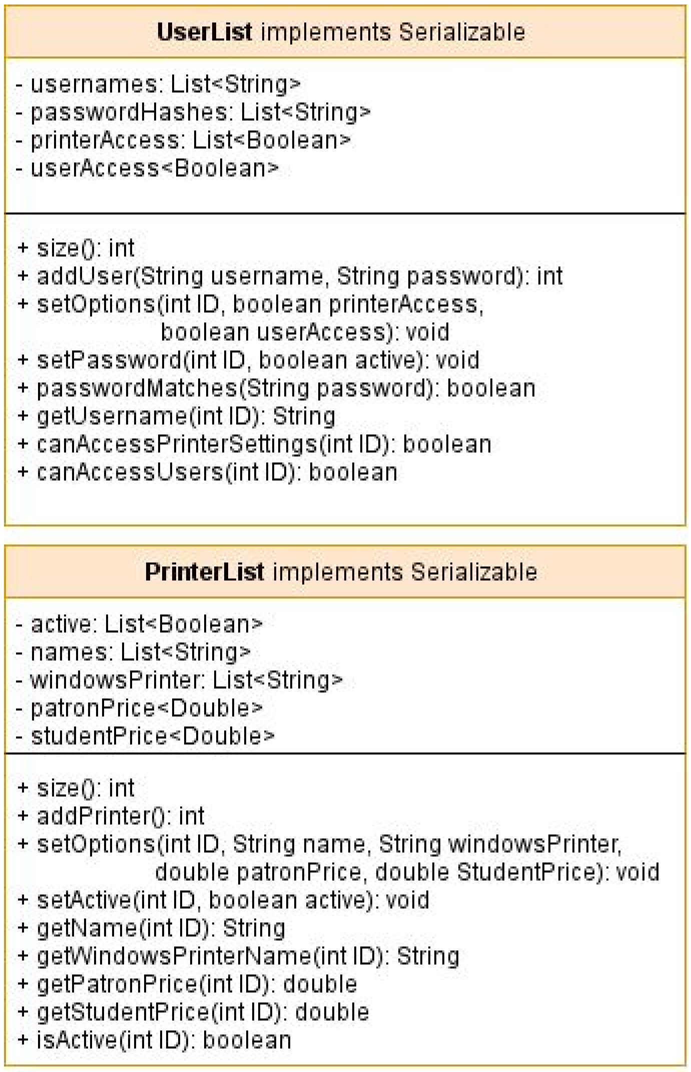
Synchronized Database Accessor Code (Java)

```
package edu.letu.libprint;

import java.io.File;
import java.io.Serializable;
import java.util.function.Consumer;

import com.quirkygaming.errorlib.ErrorHandler;
import com.quirkygaming.propertydb.PropertyDB;
import com.quirkygaming.propertylib.MutableProperty;

public class Database { // Persistent Synchronized Database Accessor

        private static final long CONFIG_VERSION = 1L;

        public static final ErrorHandler<RuntimeException> pdb_handler =
                        PropertyDBListener.pdb_handler;

        private static File storageDirectory;

        private static MutableProperty<UserList> userList;
        private static MutableProperty<PrinterList> printerList;

        private Database() {}

        static void init() { // Called by PropertyDBListener
                // Get Windows data folder
                String appdata = System.getenv("APPDATA");
                // Use APPDATA/LibPrint as data storage
                storageDirectory = new File(appdata, "LibPrint/");
                storageDirectory.mkdirs(); // Make sure it exists

                // Initialize the UserList object
                userList = PropertyDB.initiateProperty(
                                storageDirectory, "UserList", CONFIG_VERSION,
                                new UserList(), pdb_handler);

                // Initialize the PrinterList object
                printerList = PropertyDB.initiateProperty(
                                storageDirectory, "PrinterList", CONFIG_VERSION,
                                new PrinterList(), pdb_handler);
        }

        // Use these methods to perform synchronized read/writes
        // Supply the modifier consumer with a lambda expression like (obj) -> {...}
        // ---- ACCESSOR METHODS ----

        public static synchronized void accessUserList(
                                        Consumer<UserList> accessor, boolean modify) {
                accessor.accept(userList.get());
                if (modify) userList.update(); // Tell PropertyDB to sync the object if modified
        }

        public static synchronized void accessPrinterList(
                                        Consumer<PrinterList> accessor, boolean modify) {
                accessor.accept(printerList.get());
                if (modify) printerList.update(); //Tell PropertyDB to sync the object if modified
        }
}
```

Request and Response Pseudocode (must be implemented in the RequestHandler servlet)

```
method validateClient(...)
  if secToken is not valid then return “Response: Error\nError: Invalid security token\n”
  if !printerExists(printerID) or printerInMaintenanceMode(printerID) then
           return “Response: Error\nError: The selected printer is not currently available.\n”
  if queue.numElements >= 30 then return “Response: Error\nError: Queue is full\n”
  if queue.numElements(username) >= 3 then
           return “Response: Error\nError: You have already have 3 documents queued to print.\n”
End method

Event: POST to RequestHandler where request=getInformation // (by Client Application)
  validateClient(...)
  // Otherwise
  return “Response: OK\n” + getPrinterTextOptions() + “TotalCost: ” + getTotalCost() + “\n”
End Event

Event: POST to RequestHandler where request=printPDF // (by Client Application)
  validateClient(...)
  // Otherwise
  copyStreamToFile(postBody, cacheDir + username + “_” + filename + “.pdf”)
  addPDFToQueue(...)
  return “Response: OK\n” + getPrinterTextOptions() + “TotalCost: ” + getTotalCost() + “\n”
End Event

Event: POST to RequestHandler where request=login // (by jQuery frontend)
  if loggedInUsers is greater than 3 return “The number of administrators has exceeded the maximum.”
  return getNewAdministratorLoginToken(EXPIRATION_TIME)
End Event

Event: POST to RequestHandler where request=login // (by jQuery frontend)
  expireLoginToken(token)
  return “User successfully logged out.”
End Event

Event: GET to RequestHandler where request=getPrinterOptions // (by jQuery frontend)
  if loginToken is not valid return “”
  return getJSONPrinterList()
End Event

Event: GET to RequestHandler where request=getQueue // (by jQuery frontend)
  if loginToken is not valid return “”
  return getJSONPrintQueue()
End Event

Event: GET to RequestHandler where request=getHistory // (by jQuery frontend)
  if loginToken is not valid return “”
  return getJSONPrintHistory()
End Event

Event: POST to RequestHandler where request=editPrinter // (by jQuery frontend)
  if loginToken is not valid return “”
  ...// Access database and handle appropriately
End Event

Event: POST to RequestHandler where request=editUser // (by jQuery frontend)
  if loginToken is not valid return “”
  ...// Access database and handle appropriately
End Event

Event every_minute // Execute this event every minute
  expire30MinuteOldQueueItems()
  expireInactiveLoginTokens()
End Event
```

5.2.5. Class Diagrams for PrinterList and UserList



```
UserList implements Serializable
- usernames: List<String>
- passwordHashes: List<String>
- printerAccess: List<Boolean>
- userAccess<Boolean>

+ size(): int
+ addUser(String username, String password): int
+ setOptions(int ID, boolean printerAccess,
             boolean userAccess): void
+ setPassword(int ID, boolean active): void
+ passwordMatches(String password): boolean
+ getUsername(int ID): String
+ canAccessPrinterSettings(int ID): boolean
+ canAccessUsers(int ID): boolean
```

```
PrinterList implements Serializable
- active: List<Boolean>
- names: List<String>
- windowsPrinter: List<String>
- patronPrice<Double>
- studentPrice<Double>

+ size(): int
+ addPrinter(): int
+ setOptions(int ID, String name, String windowsPrinter,
            double patronPrice, double StudentPrice): void
+ setActive(int ID, boolean active): void
+ getName(int ID): String
+ getWindowsPrinterName(int ID): String
+ getPatronPrice(int ID): double
+ getStudentPrice(int ID): double
+ isActive(int ID): boolean
```

6. Glossary

IT – Information Technology (i.e. LeTourneau's IT department)
LAN – Local Area Network
LETU – LeTourneau University
Patron – A public (non-student) user of the Library
qvPDF – A configurable print driver that is capable of automatically exporting PDF documents