Record Indexer Server

Contents

Introduction

The Record Indexer application is a Client/Server system. The Server is a Java program that assigns record batches to users, stores the indexed record data submitted by users, and allows the indexed records to be searched by key word. A single instance of the Server program executes on a machine somewhere on the network. The Server can be accessed simultaneously by any number of Clients over the network (Figure 1). The Client is a Java program that implements the graphical interface that allows users to download record batches, index the record data, and submit the indexed data to the Server. Of course, the Client program interacts heavily with the Server to implement its features.

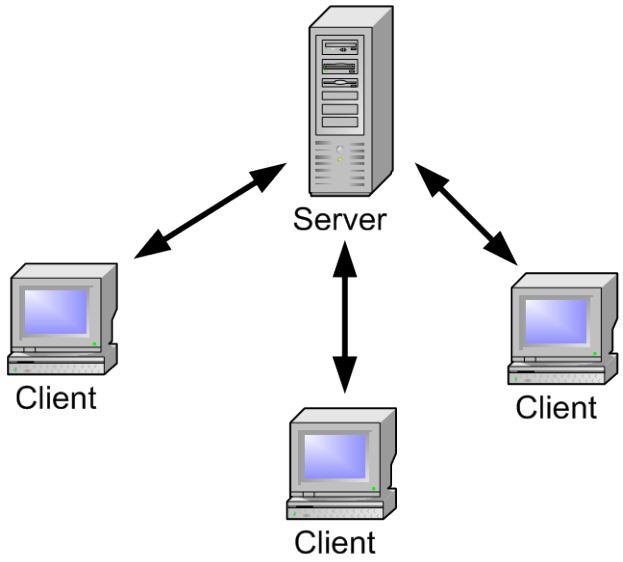

Figure 1 – Client/Server Architecture

For this project you will implement the Record Indexer Server program. By doing this project, you will gain knowledge and experience with the following concepts, skills, and technologies:

- Design, implementation, and testing of a relatively large, multi-faceted Java program
- Relational databases and Java's database APIs
- XML and Java's XML APIs
- Object serialization
- Simple Web Services using HTTP and Java's `HttpServer` class
- Automated testing using JUnit

Understanding this specification requires that you have previously read the Record Indexer Overview and Record Indexer Data Format documents.

Source Tree

The file named `record-indexer-source.zip` found on the course web site contains the source tree for this project. It contains several files that you will need, including an ANT script that automates many necessary tasks, the Java libraries needed for the project, and some testing support code. This same source tree will be used for both of the Server and Client projects.

Demo

The provided source tree contains a demo of the Record Indexer that you can run to see how the program works from the user's perspective. The demo runs as a single Java program and does not use a client/server architecture. While the operation of the demo GUI is realistic, the demo does not have a separate server program, and is different in this way from the system that you will build.

The demo can be executed by running the following command in the `record-indexer` directory (it requires that ANT has been previously installed):

```
ant demo
```

You can login using the following credentials: Username: **`sheila`**, Password: **`parker`**

Code Organization

The source tree contains several sub-directories, two of which are named `src` and `test`. The `src` directory will contain all of the "real" Java code for your project. The `test` directory will contain all of the testing Java code for your project.

To keep your Java code well organized, you should separate classes that serve different purposes into different Java packages. At the highest level, you should separate the code for the Server and Client into different packages. Server classes should be placed in appropriately named packages such as `server`, `server.this`, `server.that`, etc. Client classes should be placed in packages such as `client`, `client.this`, `client.that`, etc. If there are classes that are used by both the Server and the Client, these could be placed in packages such as `shared`, `shared.this`, `shared.that`, etc.

Server Architecture

The Server consists of a single Java program, but internally it is divided into several different pieces. In addition to the Server program itself, you will also create a few other programs to help you develop and test your Server. Figure 2 depicts the various components that you will need to create, and the dependencies between them.

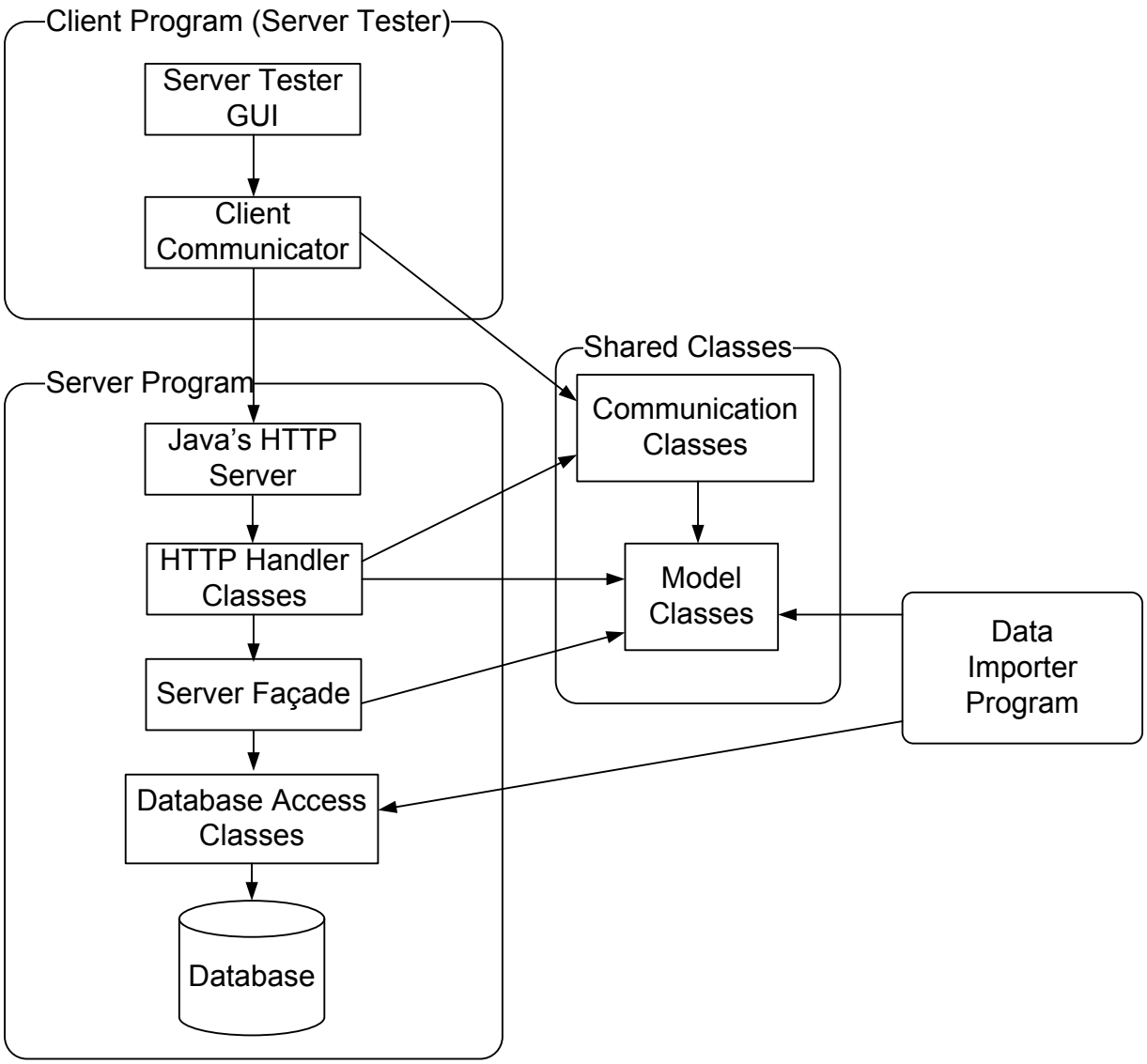

Figure 2 - Server Architecture

All of these components are described in the following sections.

Design Document

The first deliverable for this project is a design document.

The first step in developing a large program like Record Indexer is to spend some time understanding the problem you are trying to solve. Once you understand the problem, you can start to design the program by creating classes that perform each of the functions required by the program. For each class that you create, you should document what responsibilities the class has and how it interacts with the other classes in your design to perform its responsibilities. This exercise will help you determine what classes you need to write, and how those classes will work together to produce a working program.

Once you've thought through your design the best that you can without writing any code, you should make a first attempt at implementing your design. As your implementation progresses, you will probably find that your design was incomplete or faulty in some respects. This is to be expected because some insights only come from actually writing code. As you proceed, make necessary changes to your design, and incorporate them into your code.

To encourage you to follow this type of process, you will be required to submit a design document for your program. Your design document should include two parts:

1. Database Schema: Make a text file that contains all of the SQL CREATE TABLE statements needed to create all of the tables in your database schema.
2. Class Documentation: Create stub classes for your Model Classes, Database Access Classes, Communication Classes, and Client Communicator class. Document them with Javadoc comments. Run Javadoc to generate HTML documentation.

Your design document should be submitted electronically. To do so, create a web site containing: 1) the text file with your SQL CREATE TABLE statements, and 2) a directory containing the files generated by Javadoc. Send an email to the TAs containing: 1) your name, 2) the URL of your SQL text file, and 3) the URL of the `index.html` file for your Javadoc documentation. The TAs will click on these URLs to view your design document. It is your responsibility to ensure that this works properly. If it does not, the TAs will be unable to grade your work.

You might not be aware of this, but any files that you place in the `public_html` directory of your CS home directory are automatically published on the web. This allows you to easily create your own personal web site. For example, if a user has the CS login name `fred`, the URL for his web site is `http://students.cs.byu.edu/~fred/`. If he were to place a file named `database.sql` in his `public_html` directory, it would be accessible on the web at the URL `http://students.cs.byu.edu/~fred/database.sql`.

You may use your CS web site to submit your design document, or some other web site that you have.

Tasks

This section describes the steps you should follow in developing your Record Indexer Server.

1. Database Design

Based on the information in the Record Indexer Data Format document, decide how your Server will store all of the data for the application. It should use a SQLite relational database to store all data except the image, known data, and field help files, which will be stored in files in the Server's file system.

Decide what you will name your SQLite database file and where it will be stored. Decide what you will name the file system directories that will store the image, known data, and field help files and where they will be located.

Design your SQLite database schema (i.e., tables). The Firefox web browser has an excellent plugin named "SQLite Manager" that you may use to interactively create your SQLite database file, create tables, insert and query data, etc. Once you have settled on a particular schema, create a text file that contains all of the SQL statements necessary to create your schema. It should first contain statements to DROP all of the tables in your schema, and second it should contain CREATE TABLE statements to create all of the tables in your schema.

In order to support the creation of an automated pass-off driver for this project, we artificially impose the following constraints on your database design:

1) The terms "image" and "batch" are synonymous. We assume that you are storing information about all batches (i.e., images) in a database table, and that your batch table has a column which serves as a primary key (referred to herein as the "batch ID"). Batch IDs must be unique across ALL images, not just across the images for a particular project.

2) Fields have both an "ID" and a "number". We assume that you are storing information about all fields in a database table, and that your field table has a column which serves as a primary key (referred to herein as "field ID"). Field IDs must be unique across ALL fields for all projects, not just unique within a particular project. Two fields in different projects that happen to have the same name are considered to be different fields, and they should have different IDs.

3) In addition to having a "field ID", each field also has a "number", which specifies the physical position of the field on the images for its project. The leftmost field on the images for a project should have "number" one, the field to its right should have "number" two, etc. A field's "number" is entirely unrelated to its "ID".

2. Model Classes

Create a package of Java classes that model the core information manipulated by the application. You will need classes such as `User`, `Project`, `Field`, etc. These classes will largely store the same data as the tables in the database, and will be used to manipulate the data when it is in memory. These classes will primarily be data-holders, but they may also contain relevant algorithms as needed.

The model classes will be used for several important purposes in the program, including:

1) Storing and manipulating data when it is in memory
2) Transferring data back-and-forth between memory and the database
3) Transferring data back-and-forth over the network between the Server and its clients

Your model classes should be placed in their own package.

3. Database Access Classes

Create a package of Java classes that perform all database access needed by the Server. All code that directly accesses the SQLite database should be contained in these classes. They should provide operations for querying, inserting, updating, and deleting data in the database. Use Java's JDBC classes to implement all SQLite database access. Any other parts of the system that need to access information in the database should go through the database access classes.

The database access classes should be placed in their own package.

4. Database Access Unit Tests

As you develop your database access classes, use JUnit to write automated unit tests to verify that they are working properly. To help you get started with unit tests, the `test` directory already contains a class named `server.ServerUnitTests` that you can use for unit testing. You may create additional unit test classes as you see fit. If you add more unit test classes, add their names to the `testClasses` array in the `server.ServerUnitTests` `main` method. The TAs will run your unit tests by running the following ANT command:

```
ant server-tests
```

When you are done, all of your unit tests must run successfully (i.e., no failures).

5. Data Importer

In Java, write a data importer program that will take a data ZIP file (as described in the Record Indexer Data Format document) and import the data into your Server's database and directory structure. The `main` method for your importer program should accept a single command-line argument, which is the path (absolute or relative) to the XML file for the imported data set. Your importer should first delete all data from the Server's SQLite database and directory structure. Then, it should populate the database with the data in the XML file, and copy the image, known data, and field help files into the Server's directory structure. Use Java's XML classes to process the XML data file. Remember, your importer should replace any existing data rather than adding to it.

You will use this program to import data into your server for testing. It will also be used by TAs when they pass off your Server. To run your importer, we will use the ANT target named `import`, as shown below.

```
ant import -Dfile=<PATH TO XML FILE>
```

To make the `import` target work properly, you will need to modify the `import` target in your ANT `build.xml` file to do whatever you want it to do. At a minimum, you will need to specify the full package name of the Java class that implements your importer program.

6. Web Server / Web Service API

Create a package of classes that implement your main Server program. Your Server should use Java's built-in `HttpServer` class to implement a web server that:

1. Implements a web service API that provides all operations needed by the Client (described in the following sections)
2. Allows Clients to download image, known data, and field help files.

This will require you to write several HTTP Handler classes that process the various web service requests that may be received from the Client (Validate User, Get Projects, Get Sample Image, etc.).

Your Server program should accept a single command-line argument which specifies the TCP port number on which it should run. If a port number is specified, your Server should run on that port. If the specified port is the empty string, your Server should run on a default port (i.e., whatever port you want, as long as you always use the same one).

The TAs will use the ANT target named `server` to run your Server at pass off, as shown below.

```
ant server –Dport=<PORT NUMBER>
```

To make the `server` target work properly, you will need to modify the `server` target in your ANT `build.xml` file to specify the full package name of the Java class that implements your Server.

7. Client Communicator & Communication Classes

Create a Client Communicator class that your Client will use to communicate with your Server. The Client Communicator will provide client-side access to the Server's web service API, as shown in Figure 3.

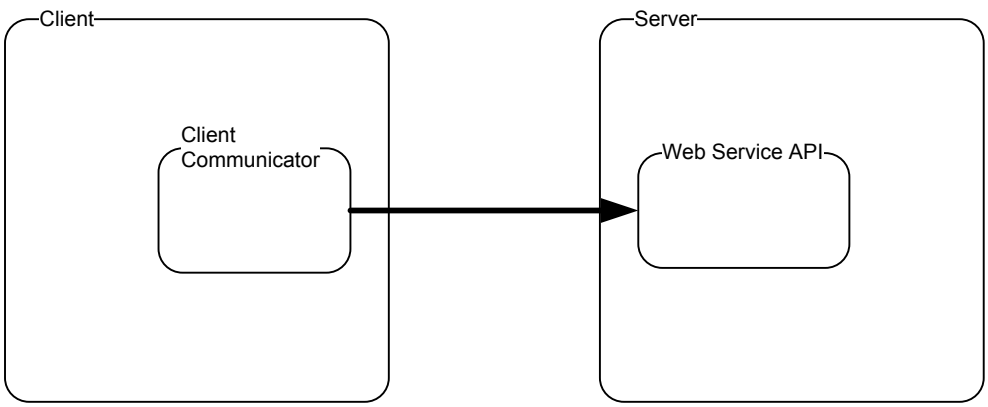

Figure 3 - Client Communicator

Your Client Communicator class should support at least the following operations (which are described in the following sections):

1. Validate User
2. Get Projects
3. Get Sample Image
4. Download Batch
5. Submit Batch
6. Get Fields
7. Search
8. Download File

As you design the method interface of your Client Communicator class, strongly consider creating additional "communication classes" that will make it easier to pass data in to and out of these operations. For example, if your Client Communicator class has an operation named `ValidateUser`, it might be useful to create classes named `ValidateUser_Params` and `ValidateUser_Result` to encapsulate the input parameters and return values of the operation. In this case, the signature of the `ValidateUser` operation would look like this:

```
ValidateUser_Result ValidateUser(ValidateUser_Params params);
```

While creating these additional "parameter" and "result" classes might seem like overkill, they make it much easier for both the client and server to pass data back and forth across the network. They are especially useful for operations that have complex inputs and/or outputs. For consistency, you may choose to create these classes for every operation on your Client Communicator class, or you may choose to create them only for operations with complex inputs and/or outputs.

8. Client Communicator Unit Tests

As you develop your Client Communicator, write automated unit tests to verify that it is working properly. Add your tests to the `server.ServerUnitTests` class, or create additional test classes of your own. If you add more unit test classes, add their names to the `testClasses` array in the `server.ServerUnitTests` `main` method. The TAs will run your unit tests by running the following ANT command:

```
ant server-tests
```

When you are done, all of your unit tests must run successfully (i.e., no failures).

9. Server Tester Program

Because you do not yet have a real Client to test your Server with, we provide a server test client for you to use. We call the test client the Server Tester. The Server Tester is implemented by the classes in the `servertester.*` packages in the `test` directory. The main class for the Server Tester is `servertester.GuiTester`, and it can be run using the following ANT command:

`ant server-gui`

The GUI of the Server Tester is shown in Figure 4. It shows all of the operations that your Server (and Client Communicator) should implement, and what parameters each operation should take.

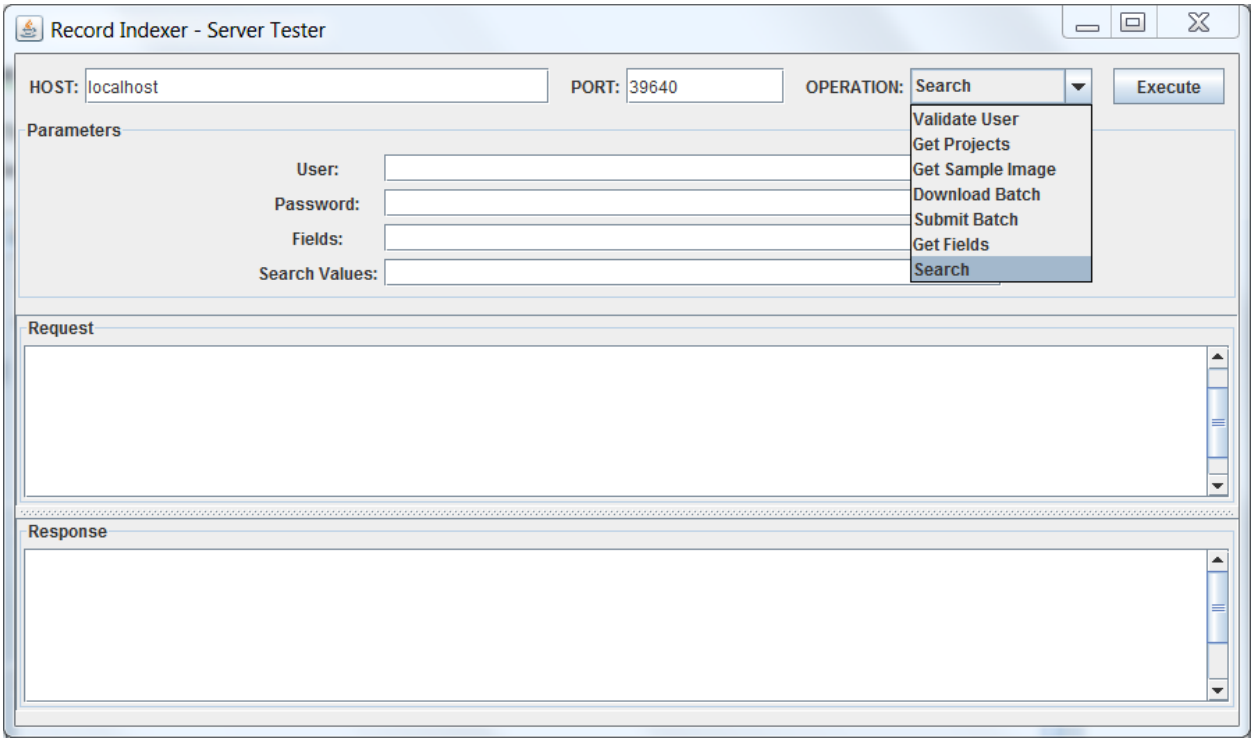

Figure 4 - Server Tester

The Server Tester lets you test each of the operations the Server supports. The OPERATION field lists each of the Server operations: Validate User, Get Projects, Get Sample Image, Download Batch, Get Fields, Submit Batch, and Search. The Server Tester is used as follows:

1. Specify the machine the Server is running on in the HOST field
2. Specify the port number the Server is running on in the PORT field
3. Select the operation to be called in the OPERATION box
4. In the PARAMETERS section, specify the parameter values to be passed to the operation
5. Click the EXECUTE button, which calls the Client Communicator to execute the operation. The Client Communicator assembles a request, which it then sends to the Server. The request is displayed in the REQUEST text field. When the Server's response is received by the Client Communicator, it is displayed in the RESPONSE text field.
6. By inspecting the REQUEST and RESPONSE text fields, you can see how your Server and Client Communicator are working.

To integrate your Client Communicator with the provided Server Tester code, you should implement the empty methods in the `servertester.controllers.Controller` class. These methods are called when the EXECUTE button is clicked. They should perform the following steps:

1. Retrieve the user-provided parameter values from the GUI. These parameters can be retrieved in the `Controller` class by calling `getView().getParameterValues()`. This returns an array of Strings. Each string is a parameter whose index corresponds to its position on the GUI for the current operation. For example, the username parameter will be at index 0 for the validate user operation, and the password will be at index 1.
2. Based on the selected OPERATION, call the corresponding method on your Client Communicator class, passing it the user-provided parameter values
3. Your Client Communicator will assemble a request to be sent to the Server. Display this request in the REQUEST text field. This field can be set by calling `getView().setRequest()`.
4. When your Client Communicator receives a response from the Server, display the response in the RESPONSE text field. This field can be set by calling `getView().setResponse()`

Automated Passoff

The Server Tester GUI defines the operations your Server needs to support, and helps you manually test those operations. We also use it to automate the pass off procedure for your Server. To make it possible for us to create an automated passoff driver, we need to standardize the inputs and outputs of each Server operation. With this goal in mind, this section precisely specifies the inputs and outputs of each Server operation. When integrating your Client Communicator with the Server Tester, you should follow this specification closely. If you don't, your Server will not be able to pass off.

Below we define the input parameters for each operation. These are the values provided by the user in the Server Tester GUI. We also define the output of each operation. This is the format you should display in the RESPONSE text field. (*NOTE: This output format is not the format you should use to communicate between your Server and Client Communicator. The format specified here is only for the purpose of making it possible for the TAs to write automated pass off drivers. Within your code, you will probably want to use an XML-based format to communicate between your Server and Client Communicator.*)

Operation Specifications

VALIDATE USER

Validates user credentials

INPUTS

`USER ::= String`	`User's name`
`PASSWORD ::= String`	`User's password`

OUTPUTS

If the user credentials are valid,

FORMAT	EXAMPLE
OUTPUT ::= TRUE\n <USER_FIRST_NAME>\n <USER_LAST_NAME>\n <NUM_RECORDS>\n USER_FIRST_NAME ::= String USER_LAST_NAME ::= String NUM_RECORDS ::= Integer	TRUE\n Sheila\n Parker\n 42\n

If the user credentials are invalid,

FORMAT	EXAMPLE
OUTPUT ::= FALSE\n	FALSE\n

If the operation fails for any reason (e.g., can't connect to the server, internal server error, etc.),

FORMAT	EXAMPLE
OUTPUT ::= FAILED\n	FAILED\n

GET PROJECTS

Returns information about all of the available projects

INPUTS

USER ::= String	User's name
PASSWORD ::= String	User's password

OUTPUTS

If the operation succeeds,

FORMAT	EXAMPLE
OUTPUT ::= <PROJECT_INFO>+ PROJECT_INFO ::= <PROJECT_ID>\n <PROJECT_TITLE>\n PROJECT_ID ::= Integer PROJECT_TITLE ::= String	3\n Draft Records\n 1\n 1890 Census\n 2\n 1900 Census\n

If the operation fails for any reason (e.g., invalid user name or password, can't connect to the server, internal server error, etc.),

FORMAT	EXAMPLE
OUTPUT ::= FAILED\n	FAILED\n

GET SAMPLE IMAGE

Returns a sample image for the specified project

INPUTS

USER ::= String	User's name
PASSWORD ::= String	User's password
PROJECT ::= Integer	Project ID

OUTPUTS

If the operation succeeds,

FORMAT	EXAMPLE
OUTPUT ::= <IMAGE_URL>\n IMAGE_URL ::= URL	http://x.byu.edu:1234/images/batch-1.png\n

If the operation fails for any reason (e.g., invalid project ID, invalid user name or password, can't connect to the server, internal server error, etc.),

FORMAT	EXAMPLE
OUTPUT ::= FAILED\n	FAILED\n

DOWNLOAD BATCH

Downloads a batch for the user to index

The Server should assign the user a batch from the requested project. The Server should not return batches that are already assigned to another user. If the user already has a batch assigned to them, this operation should fail (i.e., a user is not allowed to have multiple batches assigned to them at the same time).

Note that the known values URL may not be present for some fields.

INPUTS

USER ::= String	User's name
PASSWORD ::= String	User's password
PROJECT ::= Integer	Project ID

OUTPUTS

If the operation succeeds,

FORMAT	EXAMPLE
OUTPUT ::= <BATCH_ID>\n <PROJECT_ID>\n <IMAGE_URL>\n <FIRST_Y_COORD>\n <RECORD_HEIGHT>\n <NUM_RECORDS>\n <NUM_FIELDS>\n <FIELD>+ FIELD ::= <FIELD_ID>\n <FIELD_NUM>\n <FIELD_TITLE>\n <HELP_URL>\n <X_COORD>\n <PIXEL_WIDTH>\n (<KNOWN_VALUES_URL>\n)? BATCH_ID ::= Integer PROJECT_ID ::= Integer FIELD_ID ::= Integer IMAGE_URL ::= URL FIRST_Y_COORD ::= Integer RECORD_HEIGHT ::= Integer NUM_RECORDS ::= Integer NUM_FIELDS ::= Integer FIELD_NUM ::= Integer (>= 1) FIELD_TITLE ::= String HELP_URL ::= URL X_COORD ::= Integer PIXEL_WIDTH ::= Integer KNOWN_VALUES_URL ::= URL	4\n 1\n http://x.byu.edu:1234/images/img-4.png\n 100\n 50\n 7\n 2\n 23\n 1\n Last Name\n http://x.byu.edu:1234/help/last.html\n 125\n 250\n http://x.byu.edu:1234/known/last.txt\n 16\n 2\n First Name\n http://x.byu.edu:1234/help/first.html\n 375\n 250\n http://x.byu.edu:1234/known/first.txt\n 20\n 3\n Age\n http://x.byu.edu:1234/help/age.html\n 625\n 100\n

If the operation fails for any reason (e.g., invalid project ID, invalid user name or password, the user already has a batch assigned to them, can't connect to the server, internal server error, etc.),

FORMAT	EXAMPLE
OUTPUT ::= FAILED\n	FAILED\n

SUBMIT BATCH

Submits the indexed record field values for a batch to the Server

The Server should un-assign the user from the submitted batch. The Server should increment the total number of records indexed by the user so that the system can tell the user how many records they have indexed each time they log in. (NOTE: This is the number of individual names the user has indexed, not the number of batches. To simplify this calculation, when a batch is submitted, give the user credit for indexing all records on the batch, even if they didn't do them all.)

After a batch has been submitted, the Server should allow the batch to be searched by key word.

INPUTS

USER ::= String	User's name
PASSWORD ::= String	User's password
BATCH ::= Integer	Batch ID
FIELD_VALUES ::= RECORD_VALUES(;RECORD_VALUES)*	The field values for the batch. The values are ordered by doing a left-to-right, top-to-bottom traversal of the batch image. The values within a record are delimited by commas. Records are delimited by semicolons. Empty fields are represented with empty strings (see the example below).
RECORD_VALUES ::= String(,String)*	

Example field values: "Jones,Fred,13;Rogers,Susan,42;,,;,,;Van Fleet,Bill,23"

OUTPUTS

If the operation succeeds,

FORMAT	EXAMPLE
OUTPUT ::= TRUE\n	TRUE\n

If the operation fails for any reason (e.g., invalid batch ID, invalid user name or password, user doesn't own the submitted batch, wrong number of values, can't connect to the server, internal server error, etc.),

FORMAT	EXAMPLE
OUTPUT ::= FAILED\n	FAILED\n

GET FIELDS

Returns information about all of the fields for the specified project
If no project is specified, returns information about all of the fields for all projects in the system

INPUTS

USER ::= String	User's name
PASSWORD ::= String	User's password
PROJECT ::= Integer | Empty	Project ID | Empty String

OUTPUTS

If the operation succeeds,

FORMAT	EXAMPLE
OUTPUT ::= <FIELD_INFO>+ FIELD_INFO ::= <PROJECT_ID>\n <FIELD_ID>\n <FIELD_TITLE>\n PROJECT_ID ::= Integer FIELD_ID ::= Integer FIELD_TITLE ::= String	2\n 1\n Last Name\n 2\n 2\n First Name\n 2\n 3\n Gender\n 2\n 4\n Age\n

If the operation fails for any reason (e.g., invalid project ID, invalid user name or password, can't connect to the server, internal server error, etc.),

FORMAT	EXAMPLE
OUTPUT ::= FAILED\n	FAILED\n

SEARCH

Searches the indexed records for the specified strings

The user specifies one or more fields to be searched, and one or more strings to search for. The fields to be searched are specified by "field ID". (Note, field IDs are unique across all fields in the system.)

The Server searches all indexed records containing the specified fields for the specified strings, and returns a list of all matches. In order to constitute a match, a value must appear in one of the search fields, and be exactly equal (ignoring case) to one of the search strings.

For each match found, the Server returns a tuple of the following form:

(Batch ID, Image URL, Record Number, Field ID)

Batch ID is the ID of the batch containing the match.
Image URL is the URL of the batch's image file on the Server.
Record Number is the number of the record (or row) on the batch that contains the match (top-most record is number one, the one below it is number two, etc.).
Field ID is the ID of the field in the record that contains the match (this is the field's "ID", not its "number").

Intuitively, Search works by OR-ing the requirements together. For example, if the user searches fields 1, 7 for values "a", "b", "c", the result contains all matches for which the field is 1 OR 7 and the value is "a" OR "b" OR "c".

Alternatively, we could say that the result is equivalent to the union of the following searches:
Field 1 for "a"
Field 1 for "b"
Field 1 for "c"
Field 7 for "a"
Field 7 for "b"
Field 7 for "c"

INPUTS

USER ::= String	User's name
PASSWORD ::= String	User's password
FIELDS ::= <FIELD_ID>(,<FIELD_ID>)* FIELD_ID ::= Integer	Comma-separated list of fields to be searched
SEARCH_VALUES ::= String(,String)*	Comma-separated list of strings to search for

OUTPUTS

If the operation succeeds,

FORMAT	EXAMPLE
OUTPUT ::= <SEARCH_RESULT>* SEARCH_RESULT ::= <BATCH_ID>\n <IMAGE_URL>\n <RECORD_NUM>\n <FIELD_ID>\n BATCH_ID ::= Integer IMAGE_URL ::= URL RECORD_NUM ::= Integer (>= 1) FIELD_ID ::= Integer	4\n http://x.byu.edu:1234/images/img-4.png\n 3\n 14\n 7\n http://x.byu.edu:1234/images/img-7.png\n 5\n 9\n

If the operation fails for any reason (e.g., invalid user name or password, invalid field ID, no search values, can't connect to the server, internal server error, etc.),

FORMAT	EXAMPLE
`OUTPUT ::= FAILED\n`	`FAILED\n`

10. File Downloads

The previous section defines the web service operations that your Server (and Client Communicator) must support. Some of these operations return URLs that reference image, known data, or field help files that reside in the Server's file system. When a Client receives such a URL, it must download the contents (i.e., bytes) of the file referenced by the URL. For example, suppose the Client receives the following image URL from the Server:

```
http://x.byu.edu:1234/images/batch-1.png
```

Before the Client can display the image to the user, it must first retrieve (i.e., download) the bytes of the image file from the Server. Similarly, if the Client receives the URL of a known data file or a field help file from the Server, it must download the content of those files before it can use them.

Clients will use HTTP GET requests to download files from your Server. To make this work, your Server should implement an HTTP Handler that does the following:

1) Extracts the requested URL from the HTTP GET request (i.e., from the `HttpExchange`)
2) Maps the Path portion of the requested URL to the path of the requested file in the Server's file system. (For example, if the requested URL is `http://x.byu.edu:1234/images/batch-1.png`, and your Server stores image files in a directory named `/users/fred/cs240/record-indexer/images`, the Path from the requested URL (`/images/batch-1.png`) would map to the file path `/users/fred/cs240/record-indexer/images/batch-1.png`)
3) Opens the requested file, and passes back the contents (i.e., bytes) of the file back to the client through the HTTP response body (i.e., through the `HttpExchange`)

NOTE: The file download HTTP Handler is different from the other HTTP Handlers in your Server. The other HTTP Handlers process web service requests from the Client, and use object serialization to receive inputs from and pass outputs to the Client. However, the file download HTTP Handler does not use object serialization at all. It simply returns the bytes of the requested file back to the client through the `HttpExchange`.

To ensure that your file download HTTP Handler is working properly, you can test it with a regular web browser. Just type the URL of the file you want to download into your web browser's URL field, and enter return. The browser will use an HTTP GET request to download the file from your Server, and display the contents of the file in your browser. If this doesn't work, something is wrong with your file download.

On the Client side, you should add an operation to your Client Communicator class that will download a file from your Server. It should take a URL as input, and pass back the contents (i.e., bytes) of the requested file as output. Internally, this operation should use an HTTP GET request to download the contents of the file from your Server.

Source Code Evaluation

After you pass off your project, your source code will be graded by a TA on how well you followed the good programming practices discussed in class. The following criteria will be used to evaluate your source code:

- (15%) Effective class, method, and variable names
- (20%) Effective decomposition of classes and methods
- (20%) Code layout is readable and consistent
- (15%) Effective organization of classes into Java packages
- (30%) High-quality unit test cases implemented with JUnit