Database Manager: Remote Updating and Repair

java-plugin download and repair web service

Final Report Document

Alan Champion
Adam Curtiss
Harrison Higgins
June 20, 2016
DrillingInfo Team

Table of Contents

1. Introduction

1.1. Client Description

DrillingInfo is an energy sector data analysis company that focuses on delivering high precision geotechnical data processing tools for clientele in the petroleum and investment fields. Among other services, they provide the Transform software that is used to present geophysical data for oil and gas interpretation, visualization, and analysis. Transform is a recent acquisition by DrillingInfo and it is still operating under the perpetual license model. They are currently in an exploratory phase where DrillingInfo would like to transition Transform to a subscription web service and integrate it into its existing web services.

1.2. Project Description

Our project was to create a web service that would update the Transform software on the user's machine. Transform is comprised of around 200 jars. Jars, in this context, can be thought of as compressed and compiled java classes. Each jar contains links to other jars and when one jar is updated all corresponding links must be updated as well. Handling the repair aspect was outside of the scope our project but we were provided a stub repair service to interface with. Eventually the two services would be melded into one user-friendly updater.

Our project included: finding the local jars; checking for updates on a remote server; downloading, validating, and processing the update jars; replacing the old jars with the new; and finally interfacing with the repair service. We also made a user interface to aid the user and connect our service with the repair service.

The reason for this project is that if DrillingInfo finds a bug and wants to update a jar, they currently have to release a new version of Transform. This takes a long time and requires a fresh install of the entire program on the client's computer. Our project would allow clients to receive updates as soon as they were made available.

As stated in Section 1.1, DrillingInfo is currently deciding if they want to transition Transform to a subscription based web service. DrillingInfo would store the jars that comprise Transform on their own servers and have a launcher service that the client could run to use Transform. If they decide to make this switch, our project would be retained as part of the launcher service.

2. Requirements

2.1. Functional

At the start of our project, DrillingInfo detailed the expected functionality of our service through various stages of development. The first three stages were mandatory for the proof of concept, the next three stages were not required.

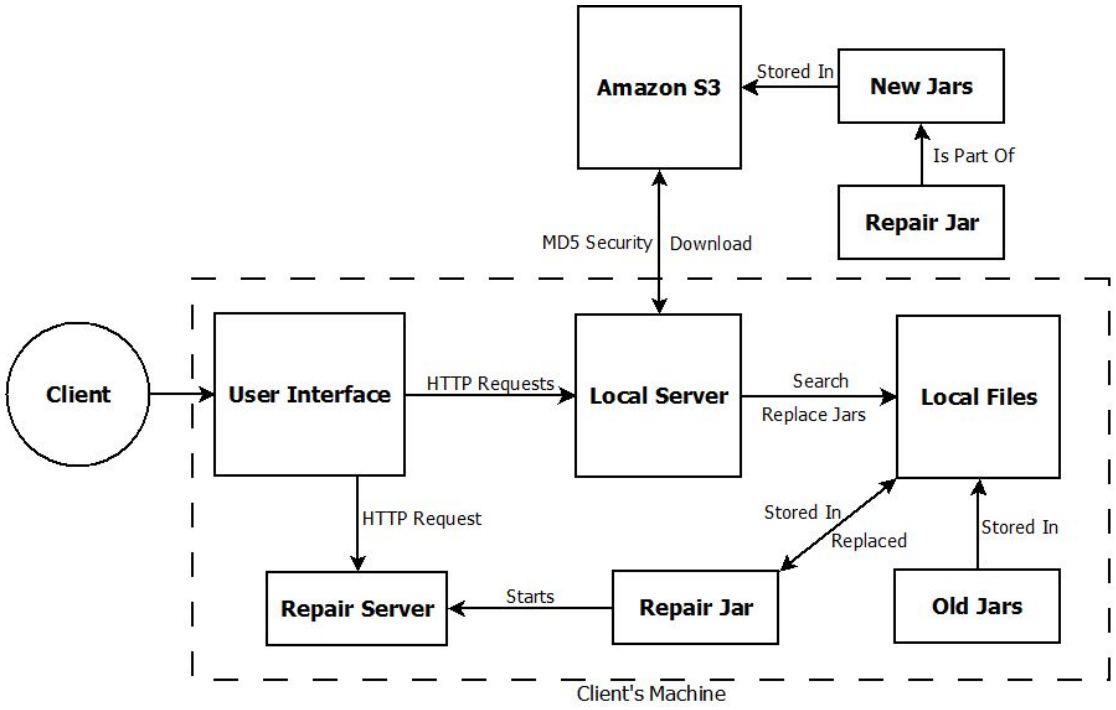
- Create Services: Familiarize ourselves with RESTful web service development using java. Understand the third party libraries.
- Versioned jar: Create and utilize web services locally. Be able to download a simple versioned jar.
- Transfer: Utilize web services to securely transfer jars to and from cloud storage.
- Transform installation: Integrate our service with Drilling Info's stub repair service.
- DBMS and user access: Create and implement a user interface to link both services. There should be an authentication layer excluding unauthorized user's access to updates (authentication was not completed).
- Run Repair: Prepare Transform and UI for full implementation and client access for DrillingInfo (not completed).

2.2. Non-Functional

- Web service written in the java programming language
- User interface written using HTML, CSS, and JavaScript
- Eclipse Integrated Development Environment
- Third party libraries Grizzly, Jackson and Jersey for server creation and RESTful API
- The service links clients to appropriate jars for their version of Transform
- Backend data storage is Amazon S3 cloud storage
- MD5 checksums for security
- Google Chrome Postman app or RESTclient for testing requests and reading responses
- Service operates on Linux and Windows
- User interface is functional on Chrome, Firefox, and Internet Explorer

3. System architecture

3.1. Figure: System Diagram



3.2. Explanation of Components

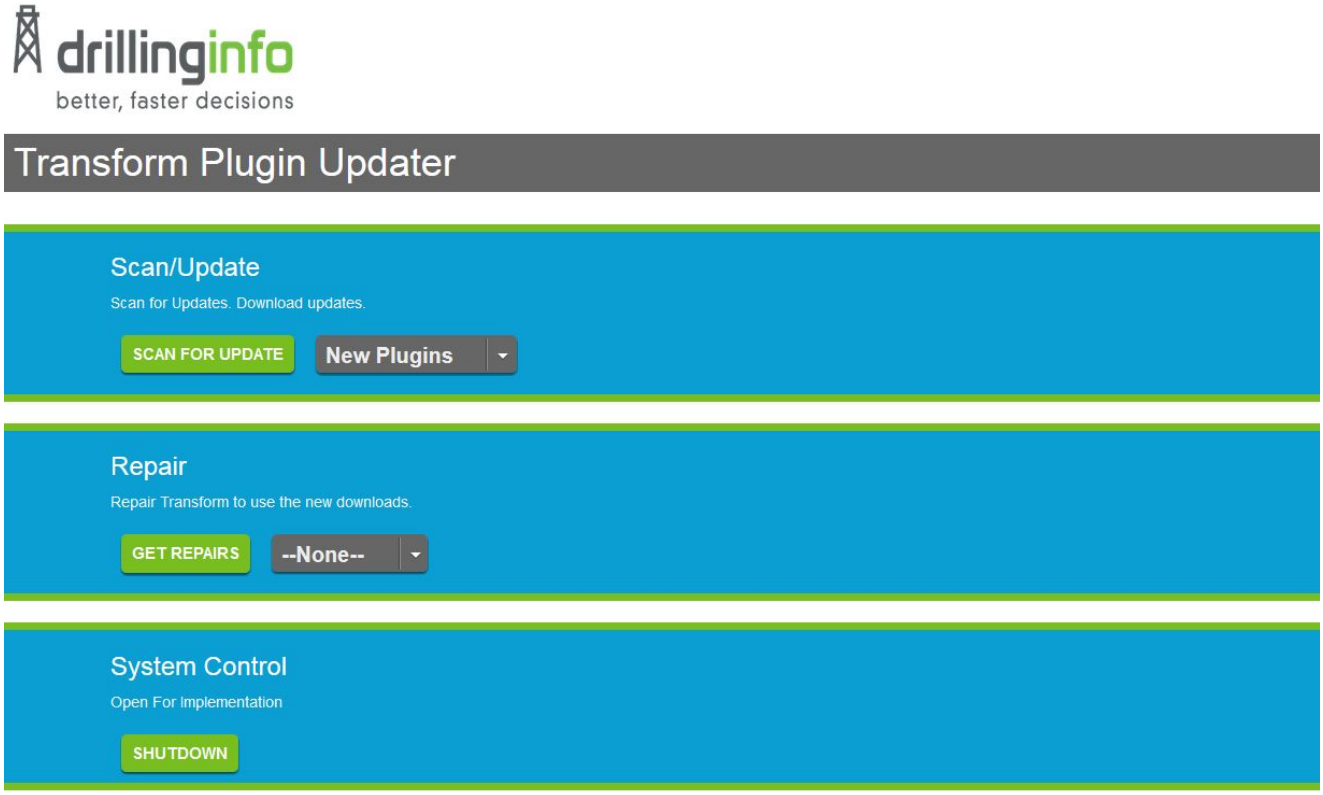
The Client is the actual user that will be using our service. This symbol represents companies that have a local version of Transform that they are looking to update.

The User Interface is the web page that they will access our services through. A more detailed look at this piece is provided in section 4. The User Interface communicates with the two different servers via HTTP Requests.

The main server, depicted as Local Server in the above figure, and its associated resources and methods comprise the bulk of our project. We discuss these components in depth in section 5. This server talks with Amazon S3 through the S3 API. In it are the jars that we want to download and check with the MD5 verification. The Local Server also talks with the Local Files on the user's machine. It searches for the files we want to replace and then replaces them when the new ones have been verified and downloaded.

The Repair Server represents the repair service that was provided by DrillingInfo. It handles repairing Transform to work with the new jars that have been downloaded.

3.3. Figure: Screenshot of User Interface



3.4. UI Description

The overall layout and color scheme were created based on the DrillingInfo homepage http://info.drillinginfo.com/. Buttons operate the services. Drop down menus display the jars that can be downloaded and the repair options available. If no jars are available, the drop down lists will not be populated. Upon population, hidden buttons for the appropriate services are made visible.

4. Technical Design

4.1. RESTful Web Service

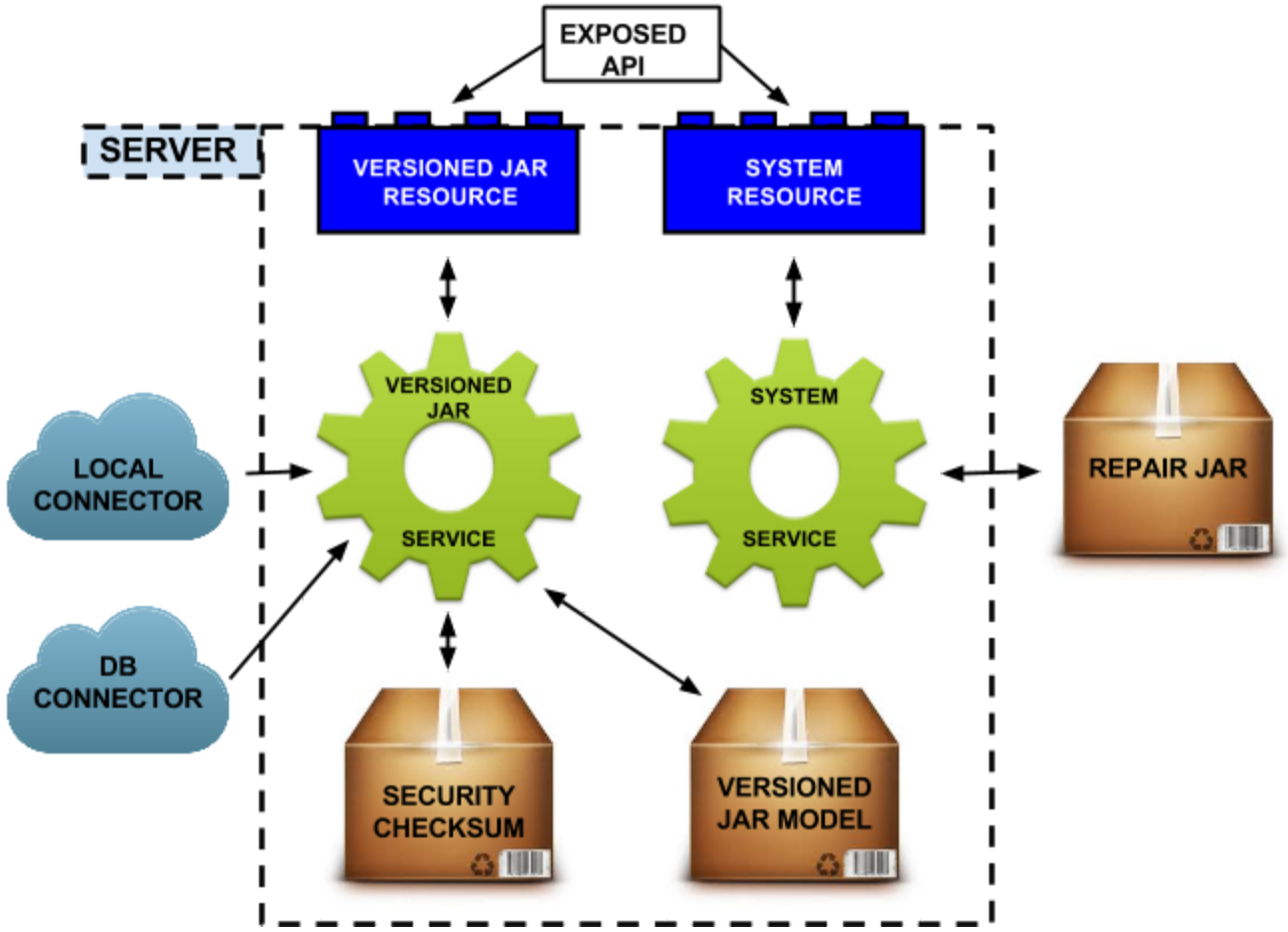
A large portion of this project was designing a RESTful web service. The three of us had no prior experience with web programming so we had to spend the first week coming up to speed on REST. REST stands for representational state transfer and is an architectural style that is most commonly applied to web services.

To be a RESTful web service the software must follow a few key principles. The service exposes a set of resources that can be accessed through unique resource identifiers (URI). The client can then use four types of HTTP requests on the resources: GET, PUT, POST, and DELETE. GET returns the requested resource. PUT replaces the resource with a new one. POST adds a new resource. DELETE removes the resource. A RESTful service can then supply the requested resource in a variety of formats based on what the client wants or can handle. The final principle is that the service uses hypermedia as the engine of application state (HATEOAS). This means that each requested resource will also supply hypermedia to the other resources of the service. This allows the client to navigate the service without needing to know the URIs in advance.

REST is often compared to SOAP which is a protocol specification for web services. Comparing the two is outside of the scope of this paper but someone who is familiar with web design would likely want to know why we chose one over the other. The simple answer is that DrillingInfo told us to make it RESTful. The better answer is that it will be easier to integrate our service into their existing code base, our service is more flexible because it can handle multiple file formats, and JSON objects can be parsed more quickly than XML.

To put our newfound knowledge into application, we exposed our VersionedJar and System resources (Appendices A and B). A client's request would then be mapped to the appropriate method in the corresponding service classes which did the heavy lifting (Appendices C and D).

4.2. Figure: REST Diagram



4.3. Downloading Process for Jars

The first thing that occurs when downloading jars is finding the user's jars. This process is explained in more detail in section 5. Once the correct file path is found, we look at the jars that are on the user's machine and go through each jar's manifest file. The manifest is a required part of a jar and contains a symbolic name and version number for the jar. We then populate a map with the symbolic-name and absolute local path to each jar in the so that we only access the local files once.

From there we open a connection to Amazon Web Services S3, which is a cloud storage service that holds all of the most recent update jars for Transform. Instead of downloading the jars to compare the manifests, S3 allows for headers to be added to files. What this lets us do is have the headers parallel the symbolic name and version number in that jar. Our service downloads this header data and compares it to the local jar map. We store a list of headers that

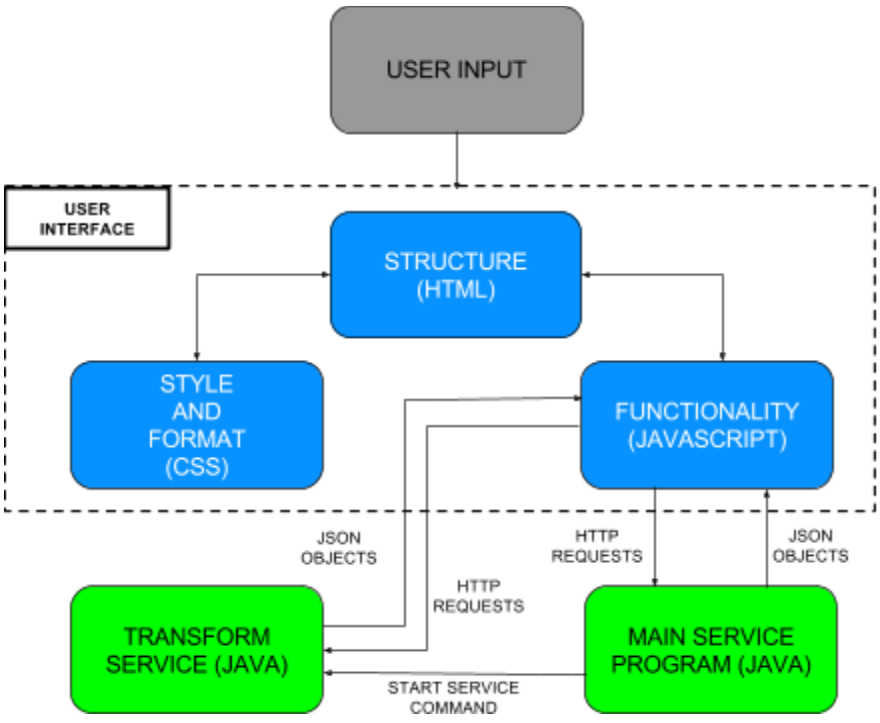
correspond with the new jars to download. We then download the new jars from S3 ensuring the least amount of data transfer possible.

Once we download the new jars, but before we delete the old jars, we generate MD5 checksums, to ensure that everything was downloaded correctly. The MD5 checksum is a hashing algorithm that is unique to a file and will be the same every time the same file is passed. Once the jar has passed this test, it is written to the file system and the old jar is deleted. Thus the downloading process is completed.

4.4. User Interface

The user interface is a web page that links the user to our download service and DrillingInfo's stub repair service. Being unfamiliar with web design, we had to learn HTML, CSS, and JavaScript. For people who are unfamiliar with these languages, HTML provides the structure of the web page, CSS adds the styling, and JavaScript handles the functionality. Our page is set up as three sections as requested by DrillingInfo. The first section interfaces with our VersionedJar resource, using JavaScript to send the appropriate HTTP requests to our server. Similarly, the second section interfaces with the repair service. The third section interfaces with our System resource. The section below is a visual representation of our UI design.

4.5. Figure: UI Diagram



5. Design and Implementation Decisions

5.1. Local File Locator

Our client needed our service to be able to find the local path of their application software directory to function correctly. We chose to write a recursive method that would search through the local files. This allowed us to put in our own criteria as to how it searched through files. This includes aborting the scan when the folder is found and ignoring folders that we don't have access to or that don't exist physically. This is likely our weakest section of code because it needs to be system independent but it also should not take too long to search. For our code to work we need to have a base directory to begin the search. We use a system call to user.home but this does not work when Transform is not found beneath this part of the directory tree.

A potential improvement to our code would be to tell our service exactly where Transform is located locally so that the search is instantaneous and having the service installed on a different machine or external drive does not break our code.

If having the user specify where Transform sits is not desirable then we could possibly restrict the user from moving anything from Transform/plugins directory. We could then see where our jar is at runtime and know where the plugins are.

5.2. Cloud Storage

We used Amazon S3 to store the update jars because it was highly recommended by DrillingInfo. DrillingInfo will be using S3 to host the update jars when this service is implemented.

5.3. Versioning

The version and name checking for the files is done with the manifest file in the jars. Using the manifests will always work because all of the jars that DrillingInfo uses will have manifests. Using manifests is more secure than checking for updates by name. An additional benefit is that it is possible to pull a manifest file without unpacking the jar file in java. This cuts down the amount of data transferred.

5.4. Security

We elected to use MD5 to generate checksums to ensure secure downloads. This method was chosen because java has a built-in tool for MD5 making it simple to implement. Also, the Amazon S3 service auto-generates a checksum in MD5 format so it is simple to check against the uploaded files. Lastly, the file sizes are large enough for MD5 to generate a secure checksum. If the file sizes were to considerably decrease a different security method should be implemented.

5.5. User Interface

We used HTML, CSS and JavaScript to create the user interface for our web service. We could have created the UI within java, but it would have been difficult to implement the design elements and style that our client desires. Additionally, we wanted the challenge and learning experience of using unfamiliar languages.

We initially attempted the UI using AngularJS but it proved unnecessarily complex for our simple design. We decided that we didn't have the time to learn AngularJS on top of everything else and chose to simplify for time.

6. Results

6.1. Features

The project has two main pieces: the java server for downloading and replacing the versioned jars; and the user interface that allows the user to simply perform these tasks. The java server searches through the user's local files to find the Transform jars. It then compares these jars to the ones stored in Amazon S3 to determine if there was an update for any of them. When an update is found, it will allow the user to download that jar to replace the local version of that jar. At this point, the user can get the repair options that that jar offers. These repairs are hosted on the downloaded jar and are not part of this project.

6.2. Unimplemented Features

There were a few features that we were unable to implement within the time constraints, the major one being a client side security layer that excludes users from updating Transform. This feature would be useful for a large company that only wanted managers and senior level engineers to be able to update their modelling software.

We would have liked to have made our server a remote server but we were unable to access the local jars without being a local server. This might not be a big issue though as Transform transitions to a web service because no files will be local and the search code will be unnecessary. The biggest downside is that our service is not as 'packaged-up' as we would like. The user needs to run our jar before accessing the UI. We would prefer to have the server start on demand and reduce the user's mandated activities for our service to function.

It was also suggested that we check compatibility of the update jars with the user's version of Transform. The project requirements were modified so that only the newest updates would be hosted on S3 and we could assume that they were backwards-compatible to any version of Transform.

6.3. Testing

We tested our final project with a mockup version of the Transform software that was provided by DrillingInfo. This was a simple collection of jars that had the same basic properties of all the Transform jars that our service would see. Our service interfaced with these jars and was able to perform an update to Transform. With these files, we were able to test the downloading

and replacing of jars to Transform, using MD5 checksums for validation, and to see the actual results as we restarted their service. This simple test proves that our service could be integrated into the full Transform service.

6.4. Summary of Testing

We tested our service on Linux and Windows, since these are the two platforms that Transform is released on. The UI was tested using Firefox, Chrome, and Internet Explorer. The weak spots are when we perform system commands to start the other service and the previously discussed issues with finding the correct directory for Transform. Without considering malicious use, our service is fully functional.

With respect to the user interface, we feel that our design is what was detailed to us by the client. The UI looks better on Firefox than Chrome or IE due to system commands that we cannot override without using a plugin. We felt that including a plugin was unnecessary since DrillingInfo will be giving our UI a makeover when our service gets integrated into Transform.

6.5. Future Work

DrillingInfo has expressed interest in transitioning our project to a launcher application where no software is on the user's local machine. As DrillingInfo starts to transition Transform to a subscription based service, our service could take a bigger role in this transition. Our project would be a major part of an application launcher, being used to find and implement updates. This would require a good bit of work to integrate our service into the launcher.

If DrillingInfo does not decide to go the web service route with Transform then the file location would need to be made more robust. A few options for work on this front were discussed in Section 5.1.

DrillingInfo will need to clean up our UI to meet their standards. They will also need to finish the repair service and integrate the finished components into Transform.

Another feature that could be implemented in the future is client side security access. For large companies using Transform, the feature would only allow select users to install updates. This would be important for large companies that invested time and money into its employees to learn a specific version of Transform. They could make sure they want the new software before an intern unknowingly downloads the new updates and causes a week long training session.

6.6. Lessons Learned

Our first challenge was overcoming the learning curve of the project. As developers, we must identify which skills and which pieces of the program are needed for basic functionality and develop that understanding first. We were forced to decide which elements were the most important and quickly learn them. This is a good skill to develop because it is applicable to many working environments. There is a vast amount of information readily available to software developers but most of it is inessential to the task at hand.

Among the computer science concepts we needed to learn for the project were: REST, MD5 security, client/server relationships, and API usage. These concepts, besides being useful to understand in general, were necessary to understand in order to complete the project.

Some project specific things we had to learn included java libraries (Including Jackson, Grizzly, and Jersey) and how to interact with Amazon Web Services. These were project specific tools that made coding and formatting far easier. We also really came to understand the benefit of using such libraries rather than trying to write our own code.

HTML, CSS and JavaScript are powerful tools for web development. We knew almost nothing when we got started on the user interface and had to learn these tools as we went. They proved to be useful and powerful but also easy to misuse and end up with incorrect UI elements. When developing for the web, these tools are invaluable.

7. Appendices

A. VersionedJarResource

```
@Path("/")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class VersionedJarResource {

    private static VersionedJarService versionedJarService = new VersionedJarService();

    /**
     * The base path that detects the jars that can be updated.
     *
     * @param uriInfo the UriInfo
     * @return the response from the HTTP request
     */
    @GET
    public Response detectAll(@Context UriInfo uriInfo) {
        ArrayList<VersionedJar> response = versionedJarService.detectAll();

        if (response == null) {
            return Response.status(Response.Status.NO_CONTENT).build();
        }
        else {
            return Response.status(Response.Status.OK).entity(response).build();
        }
    }

    /**
     * The path that replaces the jars that can be updated.
     *
     * @param uriInfo the UriInfo
     * @return the response from the HTTP request
     */
    @GET
    @Path("replace/")
    public Response replaceAll(@Context UriInfo uriInfo) {

        boolean response = versionedJarService.replaceAll();

        if (!response) {
            return Response.status(Response.Status.NO_CONTENT).build();
        }
        else {
            return Response.status(Response.Status.OK).entity(response).build();
        }
    }
}
```

B. SystemResource

```
@Path("system")
public class SystemResource
{
    static SystemService systemService = new SystemService();

    /**
     * This is the path for shutdown
     *
     * @param system the command that is passed in the URI
     * @return the response from the HTTP request
     *
     * @throws Exception
     */
    @GET
    @Path("{command}")
    public Response getDatatype(@PathParam("command") String system) throws Exception
    {
        if (system.equals("shutdown")) {
            System.exit(0);
        }
        return Response.status(Response.Status.OK).entity("Unknown Command").build();
    }

    /**
     * The path to restart the DI Server.
     *
     * @return the response from the HTTP request
     * @throws Exception
     */
    @GET
    @Path("restart")
    public Response restart() throws Exception
    {
        return systemService.restartDIServer();
    }
}
```

C. Snippets of VersionedJarService

```
 * This will find all the local files that can be updated with files on S3.
public ArrayList<VersionedJar> detectAll() {
    Map<String, ArrayList<String>> localJars = getLocalJars();
    Map<String, ObjectMetadata> dbJarMap = dbConnector.getAllJars();
    ArrayList<VersionedJar> updates = new ArrayList<VersionedJar>();

    Iterator<Entry<String, ObjectMetadata>> it = dbJarMap.entrySet().iterator();
    while (it.hasNext()) {
        Map.Entry<String, ObjectMetadata> pair = (Map.Entry<String, ObjectMetadata>)it.next();

        String s3jar = pair.getKey().toString();
        String symName = dbJarMap.get(s3jar).getUserMetaDataOf("bundle-symbolicname");

        if(localJars.containsKey(symName)){
            if (compareVersionNumbers(localJars.get(symName).get(0), dbJarMap.get(s3jar).getUserMetaDataOf("version")) <= 0) {
                localJars.remove(symName);
                it.remove();
            }
            else {
                System.out.println("Matched S3Jar : " + s3jar + " To Local Jar : " + localJars.get(symName).get(1));
                String localPath = localJars.get(symName).get(1);
                jarsOldToNew.put(localPath, s3jar);
                updates.add(new VersionedJar(s3jar));
            }
        }
        else {
            System.out.println("Local Jars didn't have " + symName);
            //TODO What if there is a file in S3 but not in the users files.
        }
    }

    if (jarsOldToNew.size() > 0) {
        System.out.println("Detected updates for " + jarsOldToNew.size() + " jars.");
        return updates;
    }
    else {
        return null;
    }
}


/**
 * This replaces the files found by the detecting for updates.
 * This checks the MD5 checksum on the download as well.
 * Pushes the file to the location of the old file and deletes the old file.
 *
 * @return If the replace worked or not.
 */
public boolean replaceAll() {
    boolean fileCreated = false;

    Iterator<Entry<String, String>> it = jarsOldToNew.entrySet().iterator();
    while (it.hasNext()) {
        Map.Entry<String, String> pair = (Map.Entry<String, String>)it.next();

        String localNew = pair.getKey().substring(0, pair.getKey().lastIndexOf(File.separatorChar) + 1);
        localNew = localNew + pair.getValue();

        S3Object s3object = dbConnector.downloadFile(pair.getValue());
        fileCreated = writeJar(localNew, s3object);

        //Remove if we don't want to delete the old file
        if(fileCreated){
            deleteFile(pair.getKey());
            jarsOldToNew.remove(pair.getKey());
        }
        else {
            deleteFile(localNew);
            throw new SecurityError("MD5 Checksums did not match.");
        }
    }
    return fileCreated;
}
```

D. SystemService

```
public class SystemService {

    /**
     * This restarts the DI Server so that it can take in the new updates.
     * If there is an error in the program it is either here or in the
     * LocalConnector.java file (In the client package) in the getPath method.
     *
     * @return the response from the HTTP request
     */
    public Response restartDIServer() {
        try {
            System.out.println(System.getProperty("user.dir"));

            /*
             * We used this "if else" because the school computers (Linux) didn't have the correct jre but
             * my laptop (Windows) does.
             *
             * Replace the if else with this :
             * Runtime.getRuntime().exec("java -cp * com.drillinginfo.rest.DIServer");
             */
            String OS = System.getProperty("os.name");
            if (OS.startsWith("Windows")) {
                Runtime.getRuntime().exec("java -cp * com.drillinginfo.rest.DIServer");
            }
            else {
                Runtime.getRuntime().exec("../jre/bin/java -cp * com.drillinginfo.rest.DIServer");
            }
        } catch (Exception e) {
            Response.status(Response.Status.INTERNAL_SERVER_ERROR).entity("Unable to start DI Server").build();
            e.printStackTrace();
        }

        return Response.status(Response.Status.OK).entity("Started DI Server").build();
    }
}
```