University of Victoria
Engineering & Computer Science/Math Co-op
Work Term Report
Spring 2017

Automated Auditing for OSCAR Service Providers

UVic L.E.A.D. Lab
Faculty of Engineering
Victoria, British Columbia, Canada

William H. Grosset
V00820930
Work Term 1
Computer Science
wgrosset@uvic.ca
April 27th, 2017

In partial fulfillment of the academic requirements of this co-op term

Supervisor's Approval: To be completed by Co-op Employer

This report will be handled by UVic Co-op staff and will be read by one assigned report marker who may be a co-op staff member within the Engineering and Computer Science/Math Co-operative Education Program, or a UVic faculty member or teaching assistant. The report will be either returned to the student or, subject to the student's right to appeal a grade, held for one year after which it will be destroyed.

I approve the release of this report to the University of Victoria for evaluation purposes only.

Signature: _____ Position: _____ Date: _____

Name (print): _____ E-Mail: _____

For (Company Name)_____

Table of Contents

Figures

Glossary

Shell	An interactive command line program that communicates directly with the operating system.
REST API	An application-programming interface that uses the HTTP protocol to send GET, PUT, POST, and DELETE requests for accessing and modifying resources.
Servlet	A Java class that is used to handle HTTP requests from the client, database, or another web application.
Encapsulation	A strategy used to hide the functionality and data of a Java program.
OAUTH	A protocol for handling user token-based authorization.
Serialization	Process of converting an object to a sequence of bits for storage.
Coupling	Interdependence between software components.
Widget	An interactive element of a graphical user interface.

Executive Summary

The ConCert (Continuous Certification) project was developed for the purpose of automatic auditing of an OSCAR (Open Source Clinical Application Resource) medical system. The auditing process of an OSCAR system can be automated to increase efficiency and extended to provide more feedback to an OSP (OSCAR Service Provider). OSPs now have the ability to use the automated auditing feature to aid in required system auditing. This report outlines the specification, design, implementation, and testing of the ConCert project. Also, the report will outline the recommendations for future development of the project.

The new feature provides auditing information for the OSCAR system environment, connected database, configuration files, and the Tomcat web container. All of the auditing information can be viewed on the corresponding JSP (JavaServer Page). The feature request was added to the source code after being reviewed via the Gerrit code-review tool on the Jenkins development server. Once the unit and integration tests for the audit web service are completed, users will be able to access the auditing information via the audit REST API (Representational State Transfer Application Programming Interface). The REST API has available URL endpoints for each of the categories described above. Recommendations for the ConCert project involve creating Bash scripts for auditing a non-live OSCAR application and extending the audit REST API for real-time feedback.

1. Introduction

OSCAR (Open Source Clinical Application Resource) is an EMR (Electronic Medical Record) system that is widely used in primary care offices throughout Canada. OSCAR was created and developed by the McMaster University in Ontario with the primary goal to help reduce management costs and increase the efficiency of a practitioner's day-to-day activities. Currently, OSCAR supports over 3.5 million Canadians and contains a growing developer community to help support the open-source platform [1]. Documentation, feature requests and bug tracking, and the source code for OSCAR are managed via the Atlassian project management tool. All Atlassian content is available to the public: https://oscaremr.atlassian.net.

This report will outline the current issues of required medical system auditing and the goals, design, implementation, and testing of the ConCert (Continuous Certification) project. Also, the report will outline the technical problems encountered, personal reflections, and recommendations for future additions to the ConCert project. The scope of the project is limited to the OSCAR system environment, with the goal to increase auditing efficiency and allow for remote system access. All work associated with OSCAR was completed with the University of Victoria L.E.A.D. Lab, under supervision from Dr. Jens Weber (Engineering Lead) and Dr. Raymond Rusk (Senior Software Engineer).

2. Problem Description

Since OSCAR is considered a medical device, regulations are required to ensure that the system environment of OSCAR is supported safely and securely. The current auditing process requires an OSP (OSCAR Service Provider) to visit an OSCAR hosted-location and manually verify system requirements. While verifying a system, OSPs are required to complete an auditing checklist and record notes regarding system behavior. The auditing checklist includes the OSCAR installation environment, interfaces to external providers, hardware/network environments, and peripheral devices.

However, the current auditing process is extremely tedious and inefficient. Checking over each required component in the audit checklist involves repetitive tasks of running shell commands inside the terminal window and manually viewing system configuration files. Currently, a standardized tool for OSCAR system auditing does not exist.

3. Continuous Certification Project

The goal of the ConCert project is to develop software that aids in automatic auditing of OSCAR medical systems that are subject to certification. The ConCert project's primary focus is to audit the OSCAR system environment, including the connected Drugref component. Drugref exposes drug information from the Health Canada Drug Product Database, which contains a configuration file to be audited (i.e. drugref2.properties). Currently, ConCert audits the following of a live OSCAR application: Linux distribution version, Java/JDK (Java Development Kit) version, connected database type and version, OSCAR configuration properties, Drugref configuration properties, and Tomcat information (see Section 3.1).

The following sections outline the technologies used and the three main development phases of the project: building low-level automation tools, the OSCAR audit web service, and merging changes into the OSCAR source code. The last section describes the technical problems and difficulties encountered during the development of the project.

3.1 Technologies & Programming Languages

Developing for OSCAR requires prior knowledge of web development, Java (7 or 8), and the associated Java libraries and frameworks that are used by the OSCAR application. Also, knowledge of common software industry tools for version control (Git), continuous integration (Jenkins), and unit/integration testing (JUnit) are highly recommended before contributing to OSCAR. The technologies outlined below describe their functionality and purpose within the ConCert project:

- **HTML/CSS**: Web-based languages used to create and style static content. Header elements, text formatting, and padding were used to create static web pages for OSPs to view.

- **Bash**: A Unix shell language. Bash scripts were created to interact with the OSCAR file system and output auditing information back to the user after being executed via the command line.

- **JSP**: JavaServer Pages allow developers to create dynamic web pages that use both static HTML content and JSP elements. Dynamic web pages are used to allow OSCAR Java back-end code to communicate and render data onto a corresponding JSP.

- **Java**: Java is an object-oriented and class-based programming language, and is the primary language used within OSCAR.

- **Struts**: An open-source MVC (Model View Controller) framework used for building maintainable and extensible Java web applications [2]. ConCert used the Struts framework to build Java programs to communicate with their corresponding JSPs for the user.

- **JAX-RS**: An available API (Application Programming Interface) in the Java Enterprise Edition to be used for building RESTful (Representational State Transfer) web services. ConCert utilized the API to build components, define endpoints, and label Java classes with annotations to define their roles for the audit REST API.

- **Spring**: The Spring framework applies the dependency injection pattern to help manage instantiation and dependency of Java objects. The audit REST API uses the framework to inject a class dependency during runtime for checking user privileges.

- **JUnit**: A simple and popular Java unit-testing framework. JUnit was used to test the behaviour and functions of each Java class within the ConCert project.

- **Tomcat**: An open-source Apache web container and HTTP server that allows users to deploy Java Servlet applications. Tomcat is used to deploy and power OSCAR application(s).

- **Maven**: Maven is an automated build tool for managing Java projects. Maven's convention over configuration, dependency management, and unit test reports allows developers to manage Java projects with a lot more ease.

- **Jenkins**: An open-source continuous integration platform for building and testing software projects.

- **Git**: A version control tool for managing and keeping track of project changes. OSCAR relies on Git for creating new features, handling bugs, and maintaining a log.

3.2 Phase 1: Low-Level Automation Tools

The initial time with OSCAR was spent getting familiar with the source code and system layout. Familiarizing myself with the OSCAR environment before beginning implementation was necessary to avoid any unnecessary complications. Implementation started with creating a Bash script that an OSP could use to retrieve auditing information about the current OSCAR instance. As shown in Figure A.1 (see Appendix), the example function within the Bash script navigates to the correct Tomcat directory and searches through the configuration file looking for a regular expression match. An OSP would execute the script via the command line, which would execute a series of functions and echo the results back onto the terminal window.

However, the Bash script was extremely limited due to not having access to an HTTP session or the OSCAR servlet context, which was necessary to retrieve the required auditing information. Due to these limitations, transitioning into using Java allowed for more flexibility. Converting the code from Bash to Java was straightforward since the logic of retrieving the auditing information and outputting to the terminal window remained the same (see Figure A.2 in the Appendix).

However, not having access to the servlet context or an HTTP session remained an issue. Utilizing the Struts framework within OSCAR allows the Java auditing class to inherit properties of an Action class. An Action class must override the execute method, which handles the HTTP request and prepares either a successful response or an error. After successfully compiling the Action class, a JSP page needed to be created for a user to view the auditing information.

When a user accesses the JSP page, the corresponding Action object is instantiated and allows the JSP page to access the auditing information through the Action object's attributes, as shown in Figure 3.1. Since mixing Java code with HTML markup is not recommended, developers can use JSTL (JavaServer Pages Standard Tag Library) to encapsulate the functionality within the Java program and avoid using scriptlets (i.e. `<%=request.getAttribute("serverVersion")%>`) within the JSP (see Figure 3.2).

```
…
  <body>
     <h5>Server Information:</h5>
     <pre><%=request.getAttribute("serverVersion")%></pre>
  </body>
…
```

Figure 3.1. Example JSP page accessing an Action object.

```
…
  <body>
     <h5>Server Information:</h5>
     <pre>${serverVersion}</pre>
  </body>
…
```

Figure 3.2. Example JSP page using JSTL to access an Action object.

OSCAR and Drugref both load their respective configuration properties files when Tomcat successfully deploys the OSCAR application. Auditing configuration files required creating a Java program that could read through a file and output the configured property values. Since property tags within a file can be overwritten sequentially, an optimization was made to read the file in reverse and only use the first property value found. Echoing the auditing information onto a web page significantly increases the efficiency of required OSCAR auditing, but the information still needed to be more accessible.

3.3 Phase 2: OSCAR Audit Web Service

Initially, research begun with looking at existing frameworks and libraries within OSCAR that could be extended for auditing purposes. OSCAR uses the Java Melody framework to monitor application performance, such as CPU/memory usage, HTTP sessions, database errors, and more (see Figure A.3 in Appendix). However, the Melody framework could not be extended for auditing purposes, and the conclusion was to build an OSCAR audit web service.

Currently, OSCAR contains web services for billing, consultations, patient details, and more. All available web services can be viewed via the OSCAR Atlassian Confluence page. OSCAR allows users with privileges to access the web services through two different methods: a user session or OAUTH. Resources within these web services are accessible through their respective REST APIs. When a user makes a successful request to the API, a JSON (JavaScript Object Notation) object will be returned.

Exploring the design and implementation of current OSCAR web services helped tremendously before beginning the development of the new audit web service. Also, researching into best software design principles for the consistency of fieldnames, response objects, and error handling contributed to the design of the audit REST API. There are 4 types of Java classes that needed to be created to follow the JAX-RS API specification for creating RESTful web services. Also, a Java class exists for the handling

the internal auditing information. The following describes a high-level overview of the classes involved in the audit web service:

- **AuditService.class**: Handles all related web service requests. Request handlers will take in arguments that match the HTTP request parameters and return a response object.

- **AuditManager.class**: Provides access to relevant data and business logic classes that are required by the AuditService route handlers (directly accesses the Audit.class).

- **AuditResponseTo1.class**: JSON wrapper object that inherits behavior of the OSCAR GenericRESTResponse class. All OSCAR response objects inherit the GenericRESTResponse class for setting the status (i.e. success or failure) and message of the request.

- **AuditTo1.class**: A serializable model object that contains the fieldnames for the JSON object.

- **Audit.class**: Handles the internal logic of retrieving the auditing information specified in Section 3.0.

Instead of creating a single response object for retrieving all the auditing information, a response object was created for each of the following audit categories: system, database, Tomcat, OSCAR, and Drugref. As the functionality of the audit REST API extends, the auditing information that is returned for each category will also extend. Also, this design choice reduces coupling within the web service. Example responses for each available endpoint in the web service can found in the Appendix (see Figure A.4).

The following describes the workflow of making a successful request to the audit REST API:

1. An authorized client will make a request to an available route handler in **AuditService.class**.

2. **AuditService.class** will check user permissions using the SecurityInfoManager class. If permission is granted, **AuditManager.class** will retrieve the data for the request by directly accessing the **Audit.class**.

3. Once this data is received, an **AuditTo1.class** object will be created to represent the model object and contain the relevant data for the request.

4. **AuditService.class** will return an **AuditResponseTo1.class** wrapper object, which will be sent back to the client as JSON.

3.4 Phase 3: Merging into OSCAR Source Code

The OSCAR repository is available to the public on the Atlassian BitBucket page. Any developer can download the repository and make their desired changes; however, all changes must be verified and reviewed before being merged into the OSCAR source code. After completing the changes locally, JUnit tests for the auditing logic had to be completed before the new feature could be added to the OSCAR source code. Unit tests cover the specific behavior of a method or function. To reach complete code coverage of a function, multiple unit tests must be created to fully cover a function that contains control statements. For example, Audit.class handles the internal auditing logic and contains 9 functions with a total of 33 corresponding JUnit tests (see Figure 3.3 for an example JUnit test).

```
@Test
public void isMatchTrueServerVersion() throws IOException {
    FileUtils.writeStringToFile((File)lsbRelease.get(audit),
"DISTRIB_DESCRIPTION=\"Ubuntu 14.04.5 LTS\"");

   String expectedResult = "Version: \"Ubuntu 14.04.5 LTS\"";
   assertEquals(expectedResult, serverVersion.invoke(audit));
}
```

Figure 3.3. JUnit test for successfully detecting the operating system.

The OSCAR source code is organized as a Maven project. The project can be downloaded locally via the Git command line tool. After adding the new Java classes (including the JUnit tests) to the specific directories within the Maven project, Maven must build successfully into a WAR file with no test conflicts or errors out of the 3000+ unit tests that exist.

Subsequently, the project must be able to build and successfully run all tests on the Jenkins development server. Using Git, the updated repository can be pushed to the Jenkins server where it will automatically start the build process and verify all tests. If the above process is successful, the integrated Gerrit code-review tool will be used to review all the new files and changes. Once verified and reviewed, the feature request will be merged into the main OSCAR project. All work completed in phase 1 of ConCert was successfully added to the OSCAR project on March 17th, 2017 under the feature request name, "Automated auditing for OSPs." The feature request can be found on the Atlassian JIRA page here: https://oscaremr.atlassian.net/browse/OSCAREMR-6051.

3.5 Problems Encountered

A few problems encountered involve using the wrong JUnit version and feedback from code review. Initially, all 33 unit tests were written using version 4.11 of JUnit. However, the dependency in the Maven project was version 4.4. This resulted in major issues due to all 33 JUnit tests relying on functions of the API that were only available in version 4.7 and above.

There were two possible solutions to this problem: update the dependency in the Maven project or find another method to mock the OSCAR file system. Due to OSCAR being a large legacy project, updating dependencies can cause many complications, such as failure of previously written tests. After discussing the issue with Dr. Raymond Rusk, the solution was to use the File and Files classes found in the Java Standard Edition to create mock folders and files for the test suite.

After submitting the audit feature request for code review, feedback was given from Marc Dumontier, the project lead for OSCAR. Before the feature request could be merged into the source code, a few changes had to be completed and reviewed again. Fortunately, the only major change that needed to be made was to include another permission check for user privileges. Due to security reasons, only users with the correct privileges should be allowed to view the auditing information. The SecurityInfoManager class that is available in OSCAR for checking user privileges solved this issue by checking privileges of the logged in user, as shown in Figure 3.4 below.

```
private void checkPrivileges(LoggedInInfo info) {
  if (!securityInfoManager.hasPrivilege(info, "_admin", "r", null)) {
    throw new SecurityException("Missing required security object
(_admin)");
  }
}
```

Figure 3.4. Private function to check user permissions with SecurityInfoManager.

4. Conclusion

Automating the required auditing process significantly increases the efficiency due to OSPs no longer having to manually view the configuration files or execute commands within the terminal. Currently, OSPs have the option to view auditing information via the JSP page when logged into an OSCAR application. The new audit web service needs each Java class to be thoroughly tested with unit and integration tests. After the web service is reviewed as a feature request and merged into the source code, OSPs will have the option to send a request to the available endpoints in the audit REST API. Provided that the development of the ConCert project will continue, automated auditing will prove useful to OSPs who are interested in quickening the process of required medical system auditing.

5. Recommendations

A recommendation for the ConCert project is to extend the functionality of the auditing Bash script(s). Currently, ConCert offers auditing solutions via the JSP page and audit web service, which both require a live instance of OSCAR. If an OSCAR application cannot start due to an internal error, an OSP may still be interested in retrieving the available auditing information from the system. Another recommendation is to extend the ConCert auditing services to allow for continuous monitoring of an application. The extended feature would include a widget on the top navigation bar of the OSCAR home screen and would display a signifier based on the severity of the issue. The continuous monitoring would be able to give real-time feedback when a value from the auditing information has been changed or modified.

6. Appendix

```
oscarVersionBuild() {
  cd $CATALINA_HOME
  file="oscar.properties"
  flag1=false
  string=""

  while read -r line; do
      [[ "$line" =~ ^#.*$ ]] && continue
      if [[ $line =~ ^"buildtag=" ]]; then
        str=$line
        string="${str:9}"
        flag1=true
      fi
    done < "$file"

    if [ "$flag1" == true ]; then
      echo ">> OSCAR version and build: $string"
    else
      echo ">> Check OSCAR version and/or build."
    fi
    cd $HOME
}
```

Figure A.1. Bash function to retrieve OSCAR version and build.

```
private String oscarBuild(String fileName) {
  try {
    String output = "";
    String line = "";
    File oscar = new File(fileName);
    ReversedLinesFileReader rf = new ReversedLinesFileReader(oscar);
    boolean isMatch1 = false;
    boolean isMatch2 = false;
    boolean flag1 = false;
    boolean flag2 = false;

    while ((line = rf.readLine()) != null) {
      if (Pattern.matches("^(#).*", line)) continue;
      isMatch1 = Pattern.matches("^(buildtag=).*", line);
      isMatch2 = Pattern.matches("^(buildDateTime=).*", line);

      if (!flag1) {
        if (isMatch1) { // buildtag=
          flag1 = true;
          output += "Oscar build and version: " + line.substring(9);
        }
      }
      if (!flag2) {
```

```
        if (isMatch2) { // buildDateTime=
          flag2 = true;
          output += "Oscar build date and time: " + line.substring(14);
        }
      }
      if (flag1 && flag2)
        break;

      if (!flag1)
        output += "Could not detect Oscar build tag.";
      if (!flag2)
        output += "Could not detect Oscar build date and time.";
      return output;
    } catch (Exception e) {
      return "Could not read properties file to detect Oscar build.";
    }
  }
}
```

Figure A.2. Java function to retrieve OSCAR version and build.

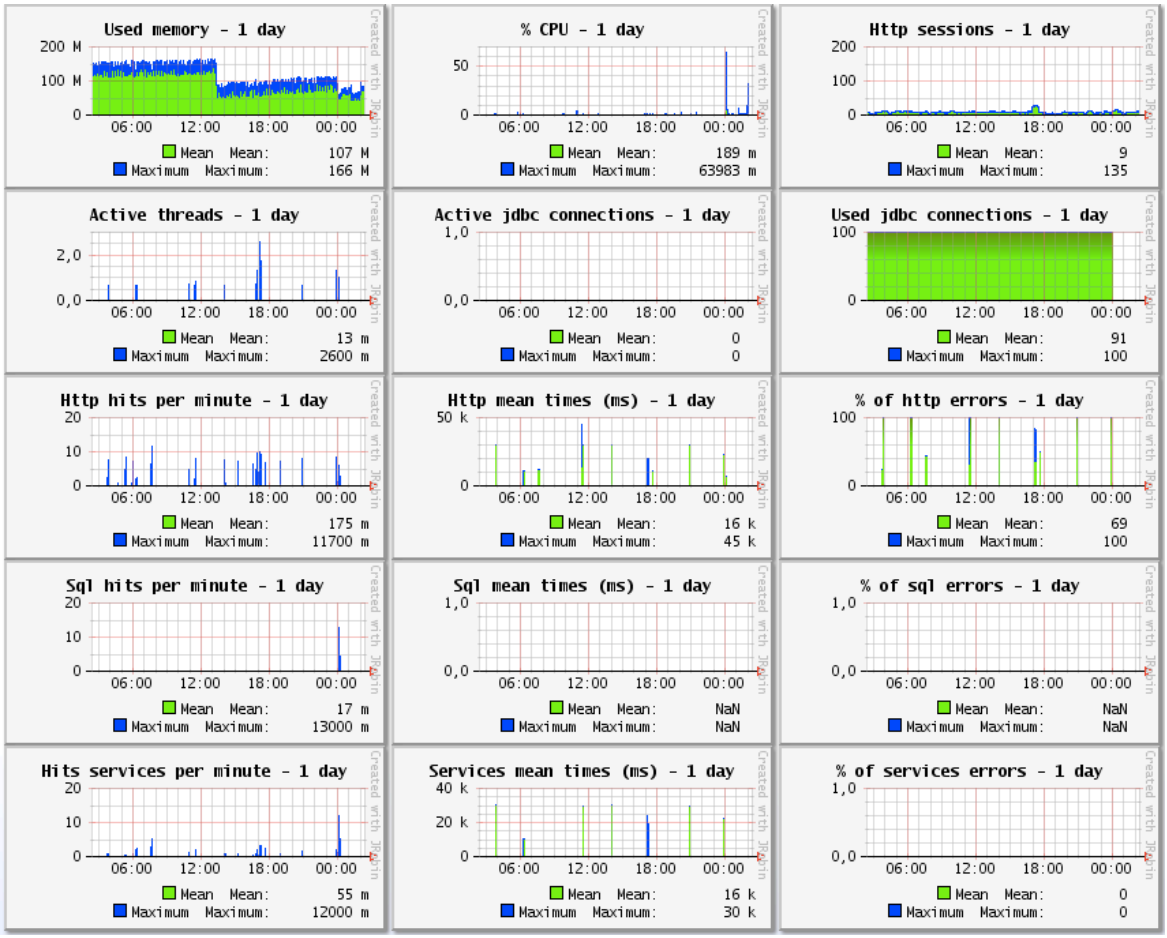

Figure A.3. Example Java Melody statistics.

- ***GET /audit/systemInfo***

  Returns the system (Linux distribution) and JVM version that the OSCAR application is live on.

  Example response**:**

```
...
"audit": {
  "timestamp": "<yyyy/MM/dd HH:mm:ss>",
  "systemVersion": "Ubuntu 14.04",
  "jvmVersion": "1.7.0_111"
}
```

- ***GET /audit/databaseInfo***

  Returns the connected OSCAR database type and version.

  Example response:

```
...
"audit": {
  "timestamp": "<yyyy/MM/dd HH:mm:ss>",
  "dbType": "MySQL",
  "dbVersion": "5.5.53"
}
```

- ***GET /audit/tomcatInfo***

  Returns the Tomcat version, and maximum/minimum (xmx/xms) heap size for Tomcat memory allocation.

  Example response:

```
...
"audit": {
  "timestamp": "<yyyy/MM/dd HH:mm:ss>",
  "tomcatVersion": "Apache Tomcat/7.0.52",
  "xmx": "1024m",
  "xms": "1024m"
}
```

- ***GET /audit/oscarInfo***

  Returns OSCAR web application name, build tag, build date, and property values for HL7TEXT_LABS, SINGLE_PAGE_CHART, TMP_DIR, and drugref_url.

  Example response:

```
...
"audit": {
  "timestamp": "<yyyy/MM/dd HH:mm:ss>",
  "webAppName": "oscar14",
  "build": "Gerrit_OSCAR-697",
```

```
  "buildDate": "2017-05-01 1:20AM",
  "hl7TextLabs": "no",
  "singlePageChart": "false",
  "tmpDir": "/etc/tmp/",
  "drugrefUrl": "http://<ip_address>:<port_number>
}
```

- ***GET /audit/drugrefInfo***

  Returns Drugref property values for db_user, db_url, and db_driver.

  Example response:

```
...
"audit": {
  "timestamp": "<yyyy/MM/dd HH:mm:ss>",
  "dbUser": "oscar",
  "dbUrl": "jdbc:mysql://127.0.0.1:drugref2",
  "dbDriver": "com.mysql.jdbc.Driver"
}
```

Figure A.4. Example responses for the audit REST API.

7. Bibliography

[1] OSCAR, "OSCAR EMRConnecting Care, Creating Community.," Who We Are | OSCAR EMR. [Online]. Available at: https://oscar-emr.com/about-us/.

[2] NetBeans, "Introduction to the Struts Web Framework," Introduction to the Struts Web Framework - NetBeans IDE Tutorial. [Online]. Available at: https://netbeans.org/kb/docs/web/quickstart-webapps-struts.html.