Final Year Project

Bug Reporter

Project Final Report

Kevin Cusack

(14102730)

Author	Kevin Cusack
Course of Study	B.Sc. Computer Science and Information Technology (2019)
Academic Supervisor	Desmond Chambers

Table of Contents

1. Acknowledgements

I would like to thank most sincerely my supervisor, Desmond Chambers for all his help and input in producing the final year project and report.

2. Introduction

For my CS&IT Final Year Project, I created a Java application called Bug Reporter, used for tracking software localisation bugs/issues. I choose this project as from previous employment where I documented software localisation issues and bugs. I was asked to look into an application that could be used to store reported software localisation bugs, description along with an image (screenshot) of any localisation issues. I used what would be considered an out of date Microsoft Access Form for logging the localisation bugs. At the time it was just about sufficient for the tasks at hand. It was minimum and not very user friendly. So I will create an application that could be used for such a purpose.

A Micro-Service (REST API) was created, enabling Bug Reporter Clients to connect to this cloud based API in order to access, create, update and delete software localisation Bugs/Issues and uploading, downloading and delete Image and PDF files. The application will be compatible with Windows operating systems.

I will outline the setup steps taken to download and configure the software tools used in the development of this project, with an overview on what these tools will be used for. Comments will also be added on issues I faced and how I resolved them.

This document is structured in the way I initially coded for this project.

3. Deliverables and Location

The repositories for the Bug Reporter Client and the REST API commits are on GitHub and can be cloned to the local machine using the links below.

You will need to use the following login information for GitHub to make repositories visible as they are *currently set to private*.

GitHub Login Information:

Username: kev17404@yahoo.co.uk

Password: Cusask174

Windows Installer Setup:

You can install this application with the Windows Installer setup.exe. This can be downloaded from Dropbox in the following repository.

https://www.dropbox.com/sh/dsfdxra7scw8w63/AAD2QVwcXlB7iM9-lC7fhptea?dl=0

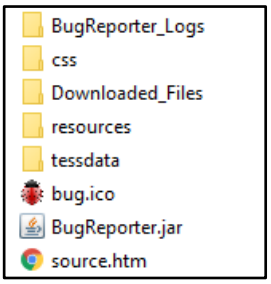
Installation Instructions for the Windows Installer package are in section 10 of this document.

Bug Reporter Client Source Files and Runnable Jar file:

https://www.dropbox.com/sh/fgy078o51o2k3fr/AAAsTsunaDzkAi3mcaqw6-mKa?dl=0

In order to avail of all features of this application, the runnable.jar file directory should also contain the following files and directories.

Note: You should apply permissions to the root folder in order to launch the application and allow this application to download files from the cloud.



BugReporter_Logs Folder: This folder will be created by the application.

CSS Folder: Contains html styles (cascading style sheets) to view a Bug in the browser window.

Downloaded_Files Folder: This folder will be created by the application.

Resources Folder: Contains image files for viewing a Bug in a browser window.

Tessdata Folder: Contains trained data for the Image to Text pattern recognition feature.

'BugReporter_Logs' & 'Downloaded_Files', directories are generated by the Windows Installer Setup.exe. If the source files are cloned from GitHub, then these directories exist and will be empty.

Bug Reporter Client Runnable Jar file and REST API War file:

Both these services can be downloaded by pasting the following links into browser.

Bug Reporter REST API Web Service Repo:

https://github.com/kev174/Bug_Reporter_REST_API_FYP

Bug Reporter REST API War File:

https://github.com/kev174/Bug-Reporter-REST-API_War

Bug Reporter Client Repo:

https://github.com/kev174/BugReporterClientMVN

Amazon AWS Web Services and Login Information:

The link to Amazon AWS web services and login information can be found in section 11 of this document.

4. Technology Overview

This section is used to describe the installation of software development tools and any configuration required to get them fully functioning for the development of the Bug Reporter Client and API.

4.1. Eclipse (IDE):

Eclipse will be the primary development environment used for developing the Bug Reporter Client and the Bug Reporter API. Both applications will be coded using Java language.

4.2. Google Window Builder Pro:

This allows me to build the Bug Reporter GUI (graphical user interface) with a user friendly Interface.

It can be downloaded by going to Eclipse > Help > Eclipse 'MarketPlace' and entering 'WindowBuilder' Pro and clicking search.

4.3. Apache Tomcat 9:

Tomcat 9 is used to run the Bug Reporter API web-service locally and was initially used to host the RESTful API.

Download Tomcat 9 from the Apache web site, and Install using default settings.

https://tomcat.apache.org/download-90.cgi

Adding Apache Tomcat 9 to Eclipse IDE:

Eclipse IDE > Window > Preferences > Server > Runtime Environment > ADD > Apache (Drop down menu) select Apache 9 > Next > Browse for Tomcat Folder and select what jre to use (1.8) alternatively (Workbench default JRE would work also) > Finish > OK.

I ran the REST API locally through Eclipse by right clicking the project > Run As > Run on Server. The developer is prompted with the option to select any version of Apache Tomcat to use (must be installed with Eclipse IDE), and can set this to a default value so this option windows is not presented again when getting this hosting server running.

4.4. SourceTree:

SourceTree can be downloaded from the following link. Accept default values. You can login by using your Google credentials.

https://www.sourcetreeapp.com/

SourceTree is used to manage commits on the Bug Reporter REST API and the Bug Reporter Client to my GitHub account.

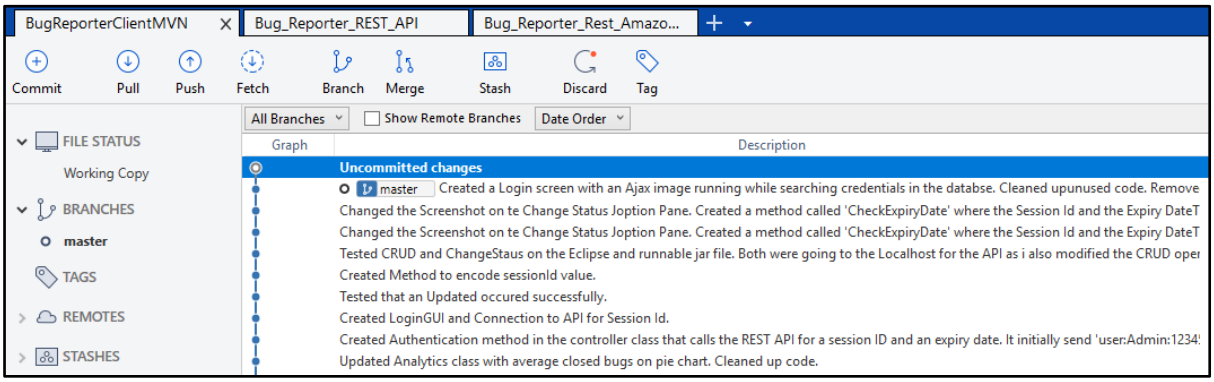


Figure 1: Commits for the Bug Reporter Client: SourceTree

4.5. WorkBench:

Workbench is used to make a connection to the Amazon RDS (Relational Database) and allows the user to add, read, update and delete entries from the RDS (CRUD operations).

Download the latest version of Workbench from the following web site and install using default settings.

https://dev.mysql.com/downloads/workbench/

Bug Reporter requires a Bug POJO (plain old Java objects) saved in a relational database. The table consists of the following columns

- Id – This will be the primary key and will be auto generated by the SQL database
- reporterName – Person recording the Bug
- testerName – Person who caught the localised error
- description – Description of the error
- severity – Priority of the error
- project – Company where the issue was caught
- screenshot – Image of the actual localised issue
- document – PDF documenting the issue caught
- startDate – Date issue was recorded by the reporterName
- endDate – Date Bug status was closed
- active – Is Bug open/closed
- bugClassification – Classification of the Bug i.e. Localisation, Graphical, Text…

This is the SQL statement used to create the Bug table within the Amazon RDS Database. 128 (128 characters length) is used to store the directory of the screenshot and document files on the local machine. Initially this was set to 64, but when a long directory was assigned, no entry was been saved in the database due to the directory length been longer than 64 characters.

The 'description' required larger amount of characters to be saved, so this was set to 4096 characters, due to this application been able to add sizable text or extract large volume of text from an image and added into the description text area of the GUI.

Every time a new Bug is created the ID will be auto incremented by one, by adding

"ENGINE = InnoDB AUTO_INCREMENTED = 1' at the end of the SQL Create statement.

```
CREATE TABLE `bug_reporter` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `reporterName` varchar(64) DEFAULT NULL,
  `testerName` varchar(64) DEFAULT NULL,
  `description` varchar(4096) DEFAULT NULL,
  `severity` int(11) DEFAULT NULL,
  `project` varchar(64) DEFAULT NULL,
  `screenshot` varchar(128) DEFAULT NULL,
  `document` varchar(128) DEFAULT NULL,
  `startDate` varchar(64) DEFAULT NULL,
  `endDate` varchar(64) DEFAULT NULL,
  `active` BOOL NOT NULL DEFAULT '0',
  `bugClassification` varchar(64) DEFAULT NULL,

  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1;
```

A table called 'credentials' was created in the AWS RDS to store users and their credentials used for login to this application. This relational database table consists of the following

- id – Primary key
- fullname = Full name of the User
- username – The users username requested for login
- password – The password requested for login
- role – The role of the user.

```
CREATE TABLE `credentials` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
```

```
  `fullname` varchar(64) DEFAULT NULL,
  `username` varchar(64) DEFAULT NULL,
  `password` varchar(64) DEFAULT NULL,
  `role` varchar(64) DEFAULT NULL,
  PRIMARY KEY (`id`)
)
```

TIP: To modify or add datatype properties in Workbench, right click the table and click 'Alter Table'.

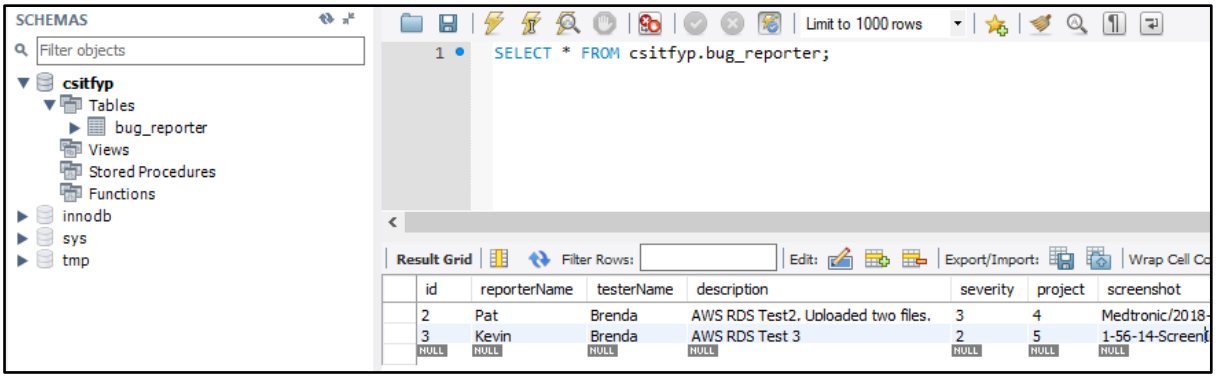


Figure 2: WorkBench containing Image of Bug objects in the Amazon RDS

4.6. SourceTree:

Create a Linux VM Client instance on the Amazon Elastic Beanstalk using EC2:

The REST API will be hosted on a Linux server instance on Amazon Elastic Beanstalk.

AWS > Login > Launch Management Console > Elastic Beanstalk > Create new Application (Top Right) > Enter name for service > Next > Create Web Server > From Predefined Configuration select Tomcat > Next > Next > Click Check Availability (Turns Green) > Next > Next.

4.7. Uploading WAR file to run on AWS EC2:

From the AWS Management Console click Elastic Beanstalk (Ensure EU (Ireland) is selected from location on the top right hand side) > Select 'bugreporterSunday_env' from the drop down menu in the upper left hand side > Click the Upload and Deploy button > Choose File: Browse to the .War file > Deploy (It takes 5/10 mins for the web service war file to be deployed).

4.8. Create a Bucket for Files on S3 (Non-Relational Database):

AWS Management Console S3 > Create Bucket > Name it a friendly name > Click Create > This Bucket is now available on the Left hand side. So click create folder and name it > Click Upload

and add files > Select files to upload > Click Start Upload - (Possibly upload video file if you like). To make files visible to everyone you select the file and on RHS click properties. At bottom you will see a link URL (end point) to access it > Click Permissions > Add more permissions > Drop down Menu - Everyone > Save.

4.9. AWS RDS:

AWS Management Console RDS > Dashboard > Crete Database > (Check Free Tier Options Only) Check Microsoft SQL Server checkbox (SQL Server Express Edition is automatically selected) > Next > db.t2.micro – 1 vCPU, 1 GiB RAM > DB Instance Identifier (Bug_Reporter), and enter master username and password (workbench will require this login information to make connection to the relational database).

Issue: Workbench was having an issue trying to connect to the RDS due to security settings. From RDS > Databases > csitfyp > Under Security Group Rules, select each security group (Inbound, Outbound) > Select either Inbound/Outbound, and click Edit > Under Source, select either My IP (It is set to custom and it works, but if you stop and restart this was changed to 'My IP').

4.10. Restoring RDS Instance from a previously saved Snapshot:

When the RDS is Started > Modify > Security > Change from default to rds-launch-wizard-2(sg-0589f2c3ec98f8c60a).

When deleting an Instance, click Delete > Select 'Yes' to Create final Snapshot > Delete.

If WorkBench prompts with the following error, 'The workbench cannot connect to this instance', you will have to make a change in AWS RDS as it needs to make a change under 'Security Groups'. In the upper right hand side click 'Instance Actions' tab, select Modify > from the 'Security Group' drop down menu select the correct group (default (sg-0c1c3270) (vpc-847a5ee2). Select apply immediately radio button > Click Modify Database.

4.11. POSTMAN:

POSTMAN was used to test the REST API, by hitting the endpoint with requests. This is due to having the backend running before the actual GUI was created.

Download the latest version of POSTMAN from the following web site and install using default settings. You will be prompted for Google login information.

https://www.getpostman.com/

4.12. Inno Setup:

Inno Setup is used to create software installer setup executable (setup.exe) for Windows computers. Users can install/uninstall this application on their Windows computer.

Inno Setup can be downloaded from the following link. Accept default installation setting prompts.

http://www.jrsoftware.org/isdl.php

See Section 10 of this document for installation instructions to install Bug Reporter Clients on a Windows computer.

4.13. Button Images:

For the GUI JButtons I used the web site 'Button Optamizer.com' to create custom button icons (buttonoptimizer.com, 2018). It allowed me to create a custom sized button with text and also add icons to the buttons which the site contains a library of.

4.14. *Issues with Bug Reporter Clients*:

Testing in Eclipse I was getting the application to function successfully, but when I exported the runnable.jar file, it failed as it was thrown the following exception.

"java.lang.ClassNotFoundException: javax.xml.bind.DatatypeConverter"

On investigating the issue, I found it was related to Eclipse compiling using Java 9. With Java 9 release, JAXB (java.xml.bind) was removed. I added the dependency in the POM file using the following POM dependency which downloaded the required 'jaxb.api.2.3.0.jar'file (Can also download the jaxb.api.2.3.0.jar file and add to build path).

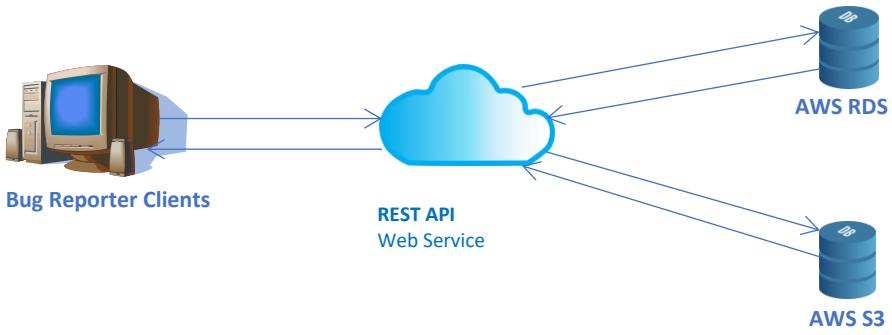
```
<dependency>
        <groupId>javax.xml.bind</groupId>
        <artifactId>jaxb-api</artifactId>
        <version>2.3.0</version>
</dependency>
```

4.15 *Bug Reporter Client Pre-Requisite*:

Bug Reporter Clients: Requires VC++ Redistributable SE2013 or VC++ Redistributable 2017 installed for OCR functionality.

5. Implementation Details: REST API

Bug Reporter Clients connect to the Amazon AWS which hosts the Bug Reporter REST API micro-service. The API web service will connect to Amazon RDS for relational database objects and to Amazon S3 for non-relational database files.



The Bug Reporter Client and REST API were built in the following order.

- Created a REST API service to listen and respond to HTML requests. This service runs locally on the localhost. POSTMAN was used to initially test that I can get a response back
- Created methods to receive and return requests from multiple @Post, @Get, @Update and @Delete objects (CRUD) operations. Testing on these methods was initially done using POSTMAN to send and receive requests (Bug objects and files)
- Create a relational database on Amazon RDS to store Bug POJO's (Plain Old Java Objects)
- Create a non-relational database on Amazon S3 to store files
- POSTMAN was used to test functionality for both the Amazon RDS and S3 cloud services
- Create Bug Reporter Client GUI where the end user will interact with cloud services to add, read, update and delete Bug objects. This section was vast as it contains numerous options for the end user, including options such as getting text from an image, rendering screenshot and PDF files to GUI buttons, and numerous warning messages should the user select incomplete operations and logging errors.

5.1. AMAZON REST API WEB-SERVICE:

Created a Maven, REST API service called 'Bug_Reporter_Rest_Amazon_Aws', whose function is to listen and respond to HTML requests from Bug Reporter Clients, and will be hosted on the Amazon Elastic Beanstalk E2 web service.

A class called 'BugReporterServiceImpl' was created, that will service all HTML requests i.e. getAll(), updateBug(), downloadFile()…

The 'Bug_Reporter_Rest_Amazon_Aws' API was initially run and tested locally. From here I coded the API to listen for requests on the following end point

http://localhost:8080/Bug_Reporter_Rest_Amazon_Aws/bugs/getAll

I set the function to return a string to ensure that the getAll() method would return a string object. POSTMAN was used to test the web service by hitting the above end point with a GET request. Testing was successful as the string from the API was returned.

With this I began to code for the remaining methods. An Interface called 'BugReporterService' was created and 'BugReporterServiceImpl' implements that interface with the following methods

- Response uploadFile();
- Response downloadFile();
- Response addBugReport()
- Response updateBug(id, Bug);
- Response deleteBug(id);
- Response changeStatus(id);
- Response getAllBugsInDB();
- Bug getSpecificBug();
- Response getSessionId();
- Boolean validSessionId();

5.2. REST API Framework: Uses annotations for the development and deployment of web service clients and endpoints. The following annotations were used by the Bug Reporter Clients to map a request to a web service resource.

Annotation	Description
@COMSUMES	Data type that a resource can accept
@PRODUCES	Data type that a resource will return
@PATH	Relative path of the resource class/method. API Endpoint
@GET	HTTP Get request, used to fetch resource
@PUT	HTTP PUT request, used to create resource

@POST	HTTP POST request, used to create/update resource
@DELETE	HTTP DELETE request, used to delete resource

The @CONSUMES annotation is used to specify which data type a resource can accept, or consume, from the Bug Reporter Clients.
The @PRODUCES annotation is used to specify the data type a resource can produce and send back to the Bug Reporter Clients.

The following is the URL used, by the 'BugReporter' Clients when they want to access resources on the REST API web-service.

http://localhost:8080/Bug_Reporter_Rest_Amazon_Aws/bugs/

This request goes to the 'ConnectToDB.java' class containing methods that make the connection to the Amazon RDS through the use of a series of SQL prepared statements.
In the case of returning all Bug POJO's from the database, a List<Bug> objects is retrieved from the Amazon RDS service and returned back to the calling method on the clients.

5.3. Return All Bug Objects in the Database:

The @Get annotation in the code below states this is a GET request, and returns a List of Bug objects in Json format.
The @Path annotation will be appended to the URL below. The Bug Reporter Clients will hit the web service using the following link

http://localhost:8080/Bug_Reporter_Rest_Amazon_Aws/bugs/getAll/OTk2NzUzNTEw

@Produces: Generates a JSON object (APPLICATION_JSON) which is a List<Bug> converted to Json format and returned to the Bug Reporter Clients.
I have placed '@Consumes(MediaType.APPLICATION_JSON)' annotation on the main class of this implementation so all access to all resources will accept only Json object requests, this way I do not have to place @Consumes before each of the resources.
This method receives a 'sid' (session Id) from the Bug Reporter Clients and encoded in base64. The 'sid is forwarded to the 'validSessionId()' method, where it is decoded from base64 to a Long. The session Id is searched within a HashMap containing a long (session Id) as a key, and a value

DateTime as the expiry date of the session Id. If the session from the client has expired, then that session Id and DateTime is removed from the HashMap.

The code used to return all Bug objects in the implementation class.

```
@Override
@GET
@Path("/getAll/{sid}")
@Produces(MediaType.APPLICATION_JSON)
public Response getAllBugsInDB(@PathParam("sid") String sid) {

	if(validSessionId(sid)) {
		db = new ConnectToDB();
		bugList = db.getAllBugs();
		return Response.ok(bugList).build();
	} else {
		return Response.status(401).entity("Unauthorized
Request").build();
	}
}
```

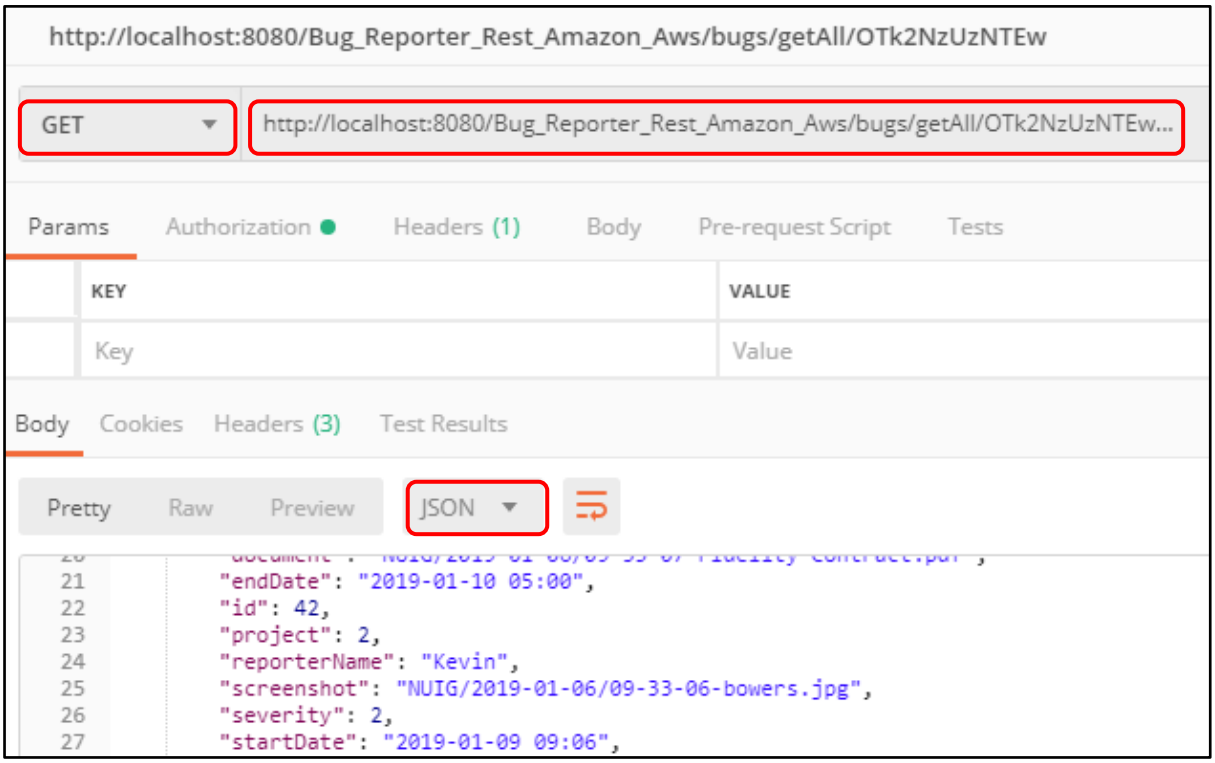


Figure 3: POSTMAN @GET Request

5.4. Get a Specific Bug:

In order to get a specific Bug for the search feature, the Bug Reporter Clients would have to append an ID to the URL. The ID is encoded in base64 on the client side. The API extracts the id from the URL and decoded from base 64 to an integer where it is used it to call the Amazon RDS and search for a Bug with that ID in the database.

The @Produces indicates, it must produce and return a Json object. The resource converts this Bug object to Json format and returns it back to the client.

This method receives a 'sid' (session Id) from the Bug Reporter Clients and encoded in base64. The 'sid is forwarded to the 'validSessionId()' method, where it is decoded from base64 to a Long. The session Id is searched with a HashMap containing a long (session Id) as a key, and a value DateTime as the expiry date of the session Id. If the session from the client has expired, then that session Id and DateTime is removed from the HashMap.

```
@GET
@Path("/{id}/getSpecificBug")
@Produces(MediaType.APPLICATION_JSON)
public Bug getSpecificBug(@PathParam("id") String id) {

        base64 = new Base64Coding();
        int bugId = Integer.parseInt(base64.decode(id));

        Bug bug = new Bug();
        db = new ConnectToDB();
        bug = db.searchForBug(bugId);

        Gson gsonBuilder = new GsonBuilder().create();
        String jsonFromBugPojo = gsonBuilder.toJson(bug);
        Gson gson = new Gson();
        Bug aBug = gson.fromJson(jsonFromBugPojo, Bug.class);

        return aBug;
}
```

5.5. Create a New Bug Object:

The @POST indicates this is a new request. In order to create a new Bug object, the API receives a Json object from the Bug Reporter Clients containing data of the actual Bug. When the API receives the Json object it sends to ConnectToDB().addEntry(Bug) method, where an automatically generated ID is generated in the Amazon RDS, and is used as the primary key. Every time a new Bug is created the Id is incremented, even thou there can be some unused Id with lesser values available due to deleted Bug objects.

The use of a setAutoCommit(false) is used to ensure that if the Bug object is not successfully created in the database that this service can return a failed to create entry in the database. If it can create this entry in the database then the myConnection.setAutoCommit(true) will apply the change to the database.

This method receives a 'sid' (session Id) from the Bug Reporter Clients and encoded in base64. The 'sid is forwarded to the 'validSessionId()' method, where it is decoded from base64 to a Long. The session Id is searched with a HashMap containing a long as a key, and a value DateTime as the expiry date of the session Id. If the session from the client has expired, then that session Id and DateTime is removed from the HashMap.

```
@Override
@POST
@Path("/{id}/addBug")
public CustomResponse addBugReport(@PathParam("id") int id, Bug bug) {

	CustomResponse customResponse = new CustomResponse();
	db = new ConnectToDB();
	boolean createdBugEntryInDatabase = false;

	try {
		createdBugEntryInDatabase = db.addEntry(bug);
	} catch (SQLException e) {
		log.error("General Exception at
			BugReporterServiceImpl.addBugReport(). " + e);
		customResponse.setStatus(true);
		customResponse.setMessage("Failed to Add a Bug with reporters
			name: " + bug.getReporterName());
		return customResponse;
	}

	customResponse.setStatus(true);
	customResponse.setMessage("Bug created successfully? " +
		createdBugEntryInDatabase + ", for reporter " +
		bug.getReporterName());
	return customResponse;
}
```

5.6. Update an Existing Bug Object:

The @PUT indicates this is an Update request. In order to update a new Bug object, the Bug Reporter Clients append an Id to the URL where it is encoded in base64. The API extracts this Id

from the URL where it is decoded from base64 to an integer and is used to call the Amazon RDS and search for a Bug with that ID in the database. If it exists, the object will be updated and a return code of 200 is retuned back to the clients. A return code of 200 indicates 'OK' or successful. If screenshot or pdf files need updating, then the S3 non-relational database will update these files. The REST API checks if there are previous files in the S3, and if newer file(s) need saving, then the previous file(s) will be deleted from the S3 database first.

This method receives a 'sid' (session Id) from the Bug Reporter Clients and encoded in base64. The 'sid is forwarded to the 'validSessionId()' method, where it is decoded from base64 to a Long. The session Id is searched with a HashMap containing a long (session Id) as a key, and a value as the expiry date of the session Id. If the session from the client has expired, then that session Id and DateTime is removed from the HashMap.

```
@PUT
@Path("/{id}/updateBug/{sid}")
public Response updateBug(@PathParam("id") String id, @PathParam("sid") String
        sid, Bug bug) {

        db = new ConnectToDB();
        base64 = new Base64Coding();
        int bugId = Integer.parseInt(base64.decode(id));
        String updateBugEntryInDatabase = "";

        if (validSessionId(sid)) {
                try {
                        updateBugEntryInDatabase = db.updateDB(bug, bugId);
                } catch (SQLException e) {
                        log.error("General Exception at
                                BugReporterServiceImpl.updateBug(). " + e);
                        return Response.status(400).entity("Failed to Add a Bug
                        with reporters name: " + bug.getReporterName()).build();
                }
        }

        return Response.status(200).entity(updateBugEntryInDatabase).build();
}
```

5.7. Delete a Specific Bug:

The @DELETE indicates this is a delete request. In order to delete a specific Bug, the Bug Reporter Clients append an ID to the URL. The API extracts this id from the URL and uses it to call the Amazon RDS to delete a Bug with the primary key of Id.

The method deleteBugAndFiles(Id); is used to check if the Bug contains any files within the POJO, and if so, it calls the method deleteFileIfExists(file directory) which deletes the file in the Amazon S3 database should it exist.

This method returns a Json object containing a value of true if deletion was successful.

This method receives a 'sid' (session Id) from the Bug Reporter Clients and encoded in base64. The 'sid is forwarded to the 'validSessionId()' method, where it is decoded from base64 to a Long. The session Id is searched with a HashMap containing a long (session Id) as a key, and a value as the expiry date of the session Id. If the session from the client has expired, then that session Id and DateTime is removed from the HashMap.

```
@Override
@DELETE
@Path("/deletebug/{bugid}/{sid}")
@Produces(MediaType.APPLICATION_JSON)
public Response deleteBug(@PathParam("bugid") String Id, @PathParam("sid") String
                sid) {

        db = new ConnectToDB();
        base64 = new Base64Coding();
        int bugId = Integer.parseInt(base64.decode(Id));

        if (validSessionId(sid)) {
                try {
                        db.deleteBugAndFiles(bugId);
                } catch (SQLException e) {
                        log.error("General Exception at
                                BugReporterServiceImpl.deleteBug(). " + e);

                        return Response.status(400).entity("400").build();
                }
        }

        return Response.status(200).entity("200").build();
}
```

5.8. Change Status of an Existing Bug Object:

The @PUT indicates this is an Update request. In order to update a new Bug object, the Bug Reporter Clients append an ID to the URL where it is encoded in to base64.

http://localhost:8080/Bug_Reporter_Rest_Amazon_Aws/bugs/changeBugStatus/OA==/OTk2NzUzNTEw

The API extracts this id from the URL and decodes the Id back from base64 to an integer, and is used to call the Amazon RDS and search for a Bug with a primary key that matches the Id value. If it exists, the object will be updated to change the status from opened to closed, and the tuple's 'endDate' cell will be updated to contain the date this transaction was closed.

This method receives a 'sid' (session Id) from the Bug Reporter Clients and encoded in base64. The 'sid is forwarded to the 'validSessionId()' method, where it is decoded from base64 to a Long. The session Id is searched with a HashMap containing a long (session Id) as a key, and a value DateTime as the expiry date of the session Id. If the session from the client has expired, then that session Id and DateTime is removed from the HashMap.

```
@Override
@PUT
@Path("/changeBugStatus/{bugid}/{sid}")
@Produces(MediaType.APPLICATION_JSON)
public Response changeBugStatus(@PathParam("bugid") String Id, @PathParam("sid")
        String sid) {

    db = new ConnectToDB();
    base64 = new Base64Coding();
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
    String todaysDate = dtf.format(LocalDateTime.now());
    int bugId = Integer.parseInt(base64.decode(Id));

    if (validSessionId(sid)) {
        try {
            db.changeBugStatus(bugId, todaysDate);
        } catch (SQLException e) {
            log.error("General Exception at
                    BugReporterServiceImpl.changeBugStatus(). " + e);
            return Response.status(400).entity("Changed Status
                    successfull: " + bugId).build();
        }
    }

    return Response.status(200).entity("Changed Status successful: " +
```

```
                bugId).build();
        }
```

5.9. Uploading Files to Amazon S3 non-relational database

It is possible that some files been uploaded will have the same names. In order to differentiate one file from another, the directory and file name will have a date and time appended to them. The directory within the Amazon S3 will contain the name of the company where the issue was recorded and a date the file(s) were recorded on, in the format 'yyyy-mm-dd'. The filename will have the time recorded in 'hh-mm-ss'. This ensures that any files the Bug Reporter Clients upload can have the same name but stored in a different directory i.e.

Medtronic/2019-1-13/01-37-58-maxresdefault.jpg

The @POST indicates this is a new request. This method receives a file object and the file is saved in the Amazon S3 database. If successful it returns the directory path back to the Bug Reporter Clients where this directory is added to the Bug object and then saved in the Amazon RDS.

```
@POST
@Path("/upload")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.APPLICATION_JSON)
public Response uploadFile(
        @FormDataParam("file") File uploadedInputStream,
        @FormDataParam("file") FormDataContentDisposition fileDetail) {

        base64 = new Base64Coding();
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd");
        String decodedFileName = base64.decode(fileDetail.getFileName());
        String temp = decodedFileName;
        String[] dirs = temp.split("/+");
        String finalFileDirectory = dirs[0] + "/" + dtf.format(LocalDateTime.now()) + "/"
                + dirs[1];

        if (uploadedInputStream == null || fileDetail == null) {
                return Response.status(415).entity("No").build();
        }

        amazonS3Api as3 = new amazonS3Api();
        as3.saveFileToS3(uploadedInputStream, finalFileDirectory);
```

```
        return Response.status(200).entity(finalFileDirectory).build();
    }
```

5.10. Downloading Files from Amazon S3 Non-Relational Database

The @GET indicates this is a get request. In order to get a specific file, the Bug Reporter Clients would have to append the filename to the URL. The API would extract this filename from the URL and use it to call the Amazon S3 and search and retrieve that file.

I had an Issue regarding downloading the file from the database, this was due to saving the file on the local machine and trying to send if back to the Bug Reporter Clients. After researching this issue it came down to not storing the file locally, but instead place in a temporary location. I used File.createTempFile() method for this purpose. I believe this would also have been an issue if running on a Linux machine on Amazon E2 due to not knowing the directory of the Linux server.

The file object is attached to the return request header, along with its name to the calling method.

```
@GET
@Path("/getFileNamed/{fn}")
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public Response downloadFile(@PathParam("fn") String fn) {

        base64 = new Base64Coding();
        String decodedFileName = base64.decode(fn);

        File file;
        String fileDir ="";
        db = new ConnectToDB();
        bugList = db.getAllBugs();

        for(Bug bug : bugList) {
                file = new File(bug.getScreenshot());
                String scr = file.getName();
                file = new File(bug.getDocument());
                String pdf = file.getName();

                if(scr.equals(decodedFileName)) {
                        fileDir = bug.getScreenshot();
                        break;
                } else if (pdf.equals(decodedFileName)) {
                        fileDir = bug.getDocument();
```

```
                break;
        } else {
                fileDir = null;
        }
    }

    File fileDownloadedFromS3 = null;

    if (fileDir != null) {
        try {
                fileDownloadedFromS3 = amazonS3Api.getFileFromS3(fileDir);
                File fileForDownload = new File(fileDir);
                ResponseBuilder response = Response.ok(fileDownloadedFromS3);
                response.header("Content-Disposition", "attachment; filename=" +
                        fileForDownload.getName());
                return response.build();

        } catch (Exception e) {
                log.error("General Exception at
                        BugReporterServiceImpl.downloadFile(). " + e);
        }
    }

    File fileForDownload = new File(fileDir);
    ResponseBuilder response = Response.ok(fileDownloadedFromS3);
    response.header("Content-Disposition", "attachment; filename=" +
        fileForDownload.getName());

   return null;
}
```

5.11. Base64 Encoding:

I did not want to hard code Ids, session Id's and filenames that would be sent over the internet on a URL. So a service called 'Base64Coding' was created, which given a string would return the string in base64. When the client also sends a request, a session Id from that client is also appended to the URL where that too is encoded in base64.

On the Web Service API this service also exists, but instead would decode the string.

On the client side, the Id and filenames are encoded in base 64, and on the REST API side they are decoded back to ASCII text.

Base64 coding is used for the following operations

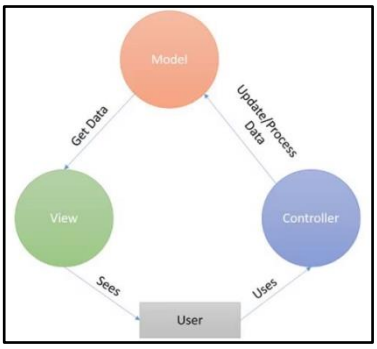
- Upload File
- Download File
- Update Bug
- Delete Bug
- Change Status
- Get All Bugs.

The REST API resources receive a base64 encoded session Id for the following requests.

- Creating a New Bug
- Update an existing Bug
- Delete an existing Bug
- Change Status of a Bug
- Get All Bugs.

These methods call the 'ValidSessionId()' method with the session Id received from the Bug Reporter Clients. If it is a valid session Id, then any requested resources will be returned back to the client.

6. Implementation Details: Bug Reporter Client



The Bug Reporter Client was designed using the Model-View-Controller (MVC) design pattern. It is commonly used for developing user interfaces to divide an application into three connected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development (Wikipedia, 2018).

Three components of the Model–View–Controller design are

1. ***The Model*** is responsible for managing the data of the application. It receives user input from the controller.
2. ***The View*** is the presentation of the model in a particular format (GUI). I created the classes: bugReporterView (main GUI), AnalyticsViewer (view data pie charts), pdfViewer (view PDF file), screenshotViewer (view screeesnhot) and ShowGeneratedHTML (view screenshot, PDF and Bug object data).
3. ***The Controller*** controls user button selection, and is the mediator between the GUI and the model. 'ActionListeners' are assigned to the GUI from this class.

6.1. Login Menu:

The GUI was generated using Google WindowBuilder Pro.

On launching this application, the user will be prompted with a login screen where they will be required to enter a valid username and password. Upon clicking the 'Submit' button. This interface will connect to the ConnectToAPIDatabase(username, password) method, which receives a username and password from the text fields in the login GUI. Within the ConnectToAPIDatabase() method, a request to the REST API getSessionId will be made, which will get users login information from the 'Credentials'

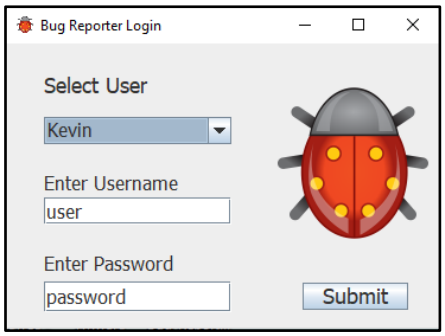


Figure 4: Bug Reporter Login

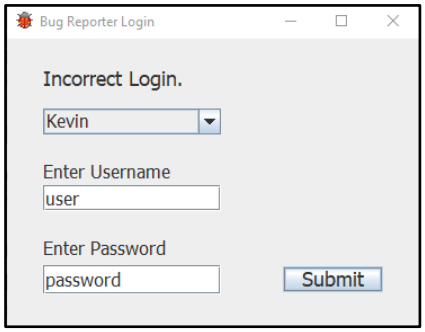


Figure 5: Incorrect Login

table in the AWS RDS relational database. If the user is authorized to use this system, a random session Id is generated and sent back to the clients. The session Id will be added to a HashMap<Long, DateTime>, where the session Id is a Long data type, and the DateTime is the current time of the system with five minutes added (This can be any arbitrary value for the session Id expiry timestamp). The session Id will enable the Bug Reporter Clients to access web service resources. The client will append the session Id encoded in base64 to the URL requests.

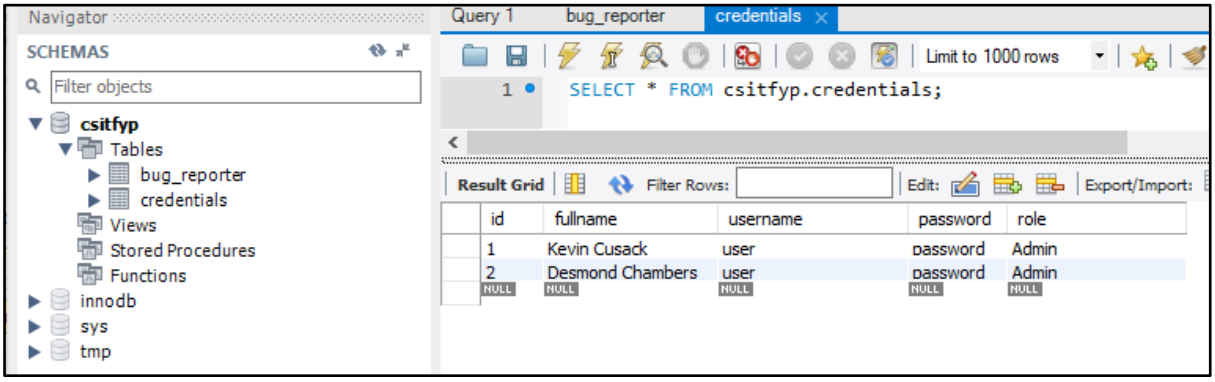


Figure 4: WorkBench - AWS Credentials Relation Database

The Bug Reporter Clients session Id's will expire after five minutes, in which case a new connection to the REST API getSessionId is called once again, and will contain the username and password to authenticate the user.

The REST API will run an ExecutiveSchedular periodically to check the HashMap that any expiry dates are removed from the HashMap.

6.2. Bug Reporter GUI:

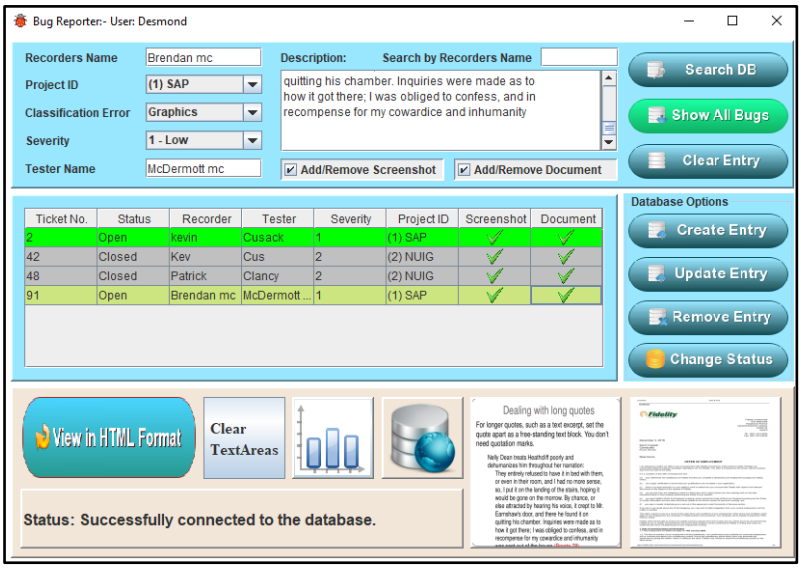


Figure 5: Bug Reporter Client Main GUI

For the Bug Reporter graphical user interface (GUI) I used Google WindowBuilder Pro to create a JFrame containing three Panels. It initially consisted of minimal buttons and a table for testing connection to the locally running API, and returned objects to the GUI table. The majority of the swing components were manually coded as I progressed through the project without Google WindowsBuilder Pro.

On initial testing whereby I was uploading and downloading files, I noticed this could take a few, to many seconds, and so I wished to add an animated ajax loader so when the application was working on network connection processes, the user would be presented with an animated loader image indicating that the application is currently working and not hanging.
I noticed when the GUI was working on connections to the web service that the animated ajax loader would not animate. I tried placing the animation in its own thread but this did not solve the issue. I investigated this further and I used 'SwingWorker', which is an abstract class to perform lengthy GUI-interaction tasks in a background thread. Several background threads can be used to execute such tasks (docs.oracle.com, 2018).

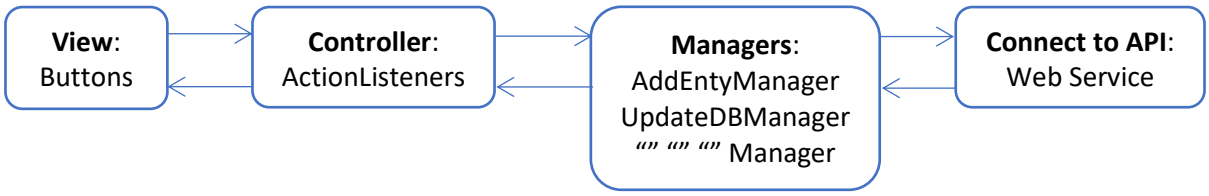
I added the 'SwingWorker' initially to each of the CRUD operations in the controller class, but the class file size was in excess of 1200 lines of code. All this code indicated to me that the controller class was doing too much work. I created manager classes to handle the CRUD operations, and from the controller I assigned these managers to the appropriate GUI class button. By moving the CRUD and file upload/download operations to their own classes, this reduced the size of the lines of code in the controller class. So any new buttons required I created a manager class for them. This also has the advantage of creating a more structured and more scalable application.

Action-Listeners were created in the main controller and are assigned to the buttons in the View class.
ActionListeners were created for the following buttons in the View class

- ViewAnalytics
- ViewScreenshot
- ViewPDF
- AddConnectToDB
- AddEntryToDb
- AddUpdateDB
- AddDeleteFromDB
- AddSearchDB

- UploadScreenshot
- UploadDocument
- ViewHTMLFormat
- AddEmptryFields
- AddJTableListener

Communication between the Controller, View, Managers and Connection to API classes are as follows



An ImageManager class was created to load all images for this application. The images reside in a source folder called resources. They will be compiled within the runnable Jar file, and loaded for the GUI using the getClassloader.getResource method. This ImageManager class contains three methods for loading specific images

1. Button Images: All images are resized to the same size of the buttons
2. Table Images: Contains a green check mark and the red X. Resized to fit the table cells
3. Label Icons: Contains the animated Ajax loader, animated trash bin and analytics images: All images are resized to the dimension of appropriate labels.

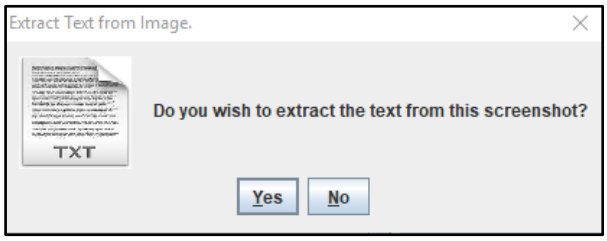
Three panels were created for this GUI. The following contains the functionality of each component with their panels.

6.3. Panel 1:

- Drop down menu containing a list of companies that the software localization error occurred with. This consists of five companies SAP, NUIG, Ericsson, Medtronic and Hewlett Packard
- Drop down menu with classification of Bugs i.e. text area truncation, screen graphics, on click error, software code error and non-comprehensive localized text

- Drop down menu with a severity of the Bug Low, Low/Medium, Medium, Medium/High and High. ActionListeners use the index of the drop down component to save/update the database with is integer
- Text area where the recorders name is entered
- Text area where the testers name is entered
- Text Field which the recorder can document an issue
- Checkbox where the recorder can upload a screenshot of the localization error. A swing filesController component is used to get a file from the user to upload to the database. A swing 'filechooser' component launches, whereby the user selects the file they wish to upload. This component only accepts .png and .jpg files. If a file other than .png or .jpg file is selected, a warning message appears indicating the user has not selected an appropriate file.

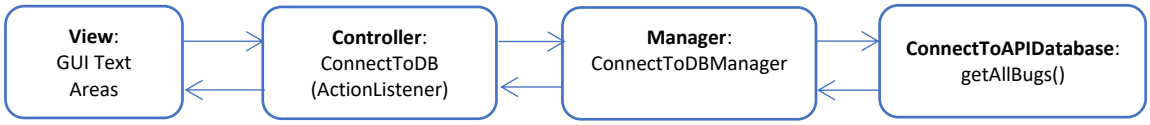
  A prompt will ask the user if they wish the text to be extracted from the screenshot. If yes is selected, the application runs a pattern recognition on the screenshot file for text which is added to the description text area of the GUI, otherwise the description is not updated. A manager class called 'ImageToTextManager()' is used for this function.

  The method ConnectToApiDatabase.POSTRequest(), makes a copy of the selected file and placed into the absolute path of the application under the directory 'Downloaded_FIles'. This has the advantage that if the user wishes to view this file, the file does not have to be downloaded from the database, but the viewer can retrieve and render the locally stored file.
- Checkbox where the recorder can upload a PDF file. A swing 'filechooser' component is used to get the file from the user to upload to the database. The swing 'filechooser' component launches where, the user selects the file they wish to upload. This class only accepts PDF files. If a non PDF file is selected, a warning message appears indicated an inappropriate file was selected.

  The method ConnectToApiDatabase.POSTRequest(), makes a copy of the selected file and placed into the absolute path of the application under the directory 'Downloaded_FIles'. This has the advantage that if the user wishes to view this file, the file does not have to download from the database, but the viewer can retrieve and render the locally stored file.

- Get All Bugs button that loads all recorded Bugs from the database in to the table. The Bug Reporter Clients receive a Json object containing a List of Bugs it receives from the web service. The Input-stream from the API connection is converted to a Json element. The Json element is converted in to a Json array. From this array, each individual Bug object is created and added to an ArrayList of Bugs.



  The ArrayList of Bug objects is passed to the View.setTable(Bugs), where a 'JTable' model is created with each individual Bug object is added. Depending on Bug data, different graphical components will be included within the table indicating whether screenshots or PDF files exist, and whether tickets are open or closed.

- Button that clear all entries in a table.
- Search button that will search all Bug entries by recorders name.

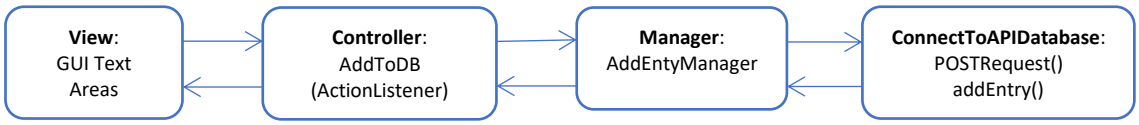
6.4. Panel 2:

This Panel contains a table that will display recorded Bugs data

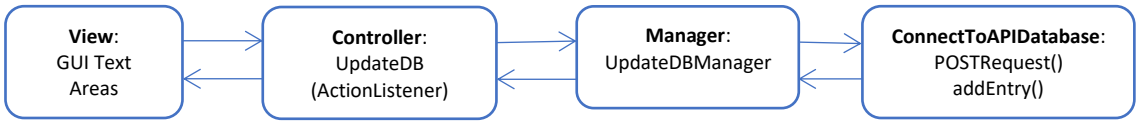
- Ticket number, Ticket Status (Open / Closed), Recorders name, Testers name, Severity, Project ID (Company name), Screenshot and PDF
- If a PDF or/and Screenshot file exist(s), then a green check mark will be visible in the table
- If either the PDF or/and screenshot does not exist(s), then a red X will be visible in the table
- If the user clicks anywhere in the table then the information in the table populates the text areas and text fields in panel one. This allows the user to update/modify any Bug entry without retyping text
- The table contains a scroll pane to allow for numerous Bugs to be loaded in to this application
- An embedded panel is placed within this panel, and contains four buttons
  - *Create Bug*: Create a new entry in the database. Information entered in the text areas and text fields in panel one will be used to create a new Bug object. If files have also been selected then they will also be added to a database using the following class.

AddEntryManager Class: Through the ActionListener in the controller, the GUI text areas and files are sent to the AddEntryManager.addEntrywithAJax() method. Screenshot and PDF Files to be added to the database are forwarded to the connectToAPIDatabase.POSTRequest() method where they are forwarded to the REST API web service and are added to the Amazon S3 file database.

Bug objects are forwarded to connectToAPIDatabase.addEntry() method and are forwarded to the REST API web service where they are added to the Amazon RDS.



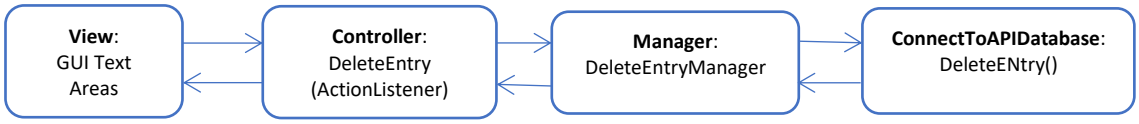
- *Update Bug*: Information modified in the text areas and text fields in panel one will be used to update that specific Bug. If a screenshot/PDF file(s) have also been updated then they will also be updated to the database.



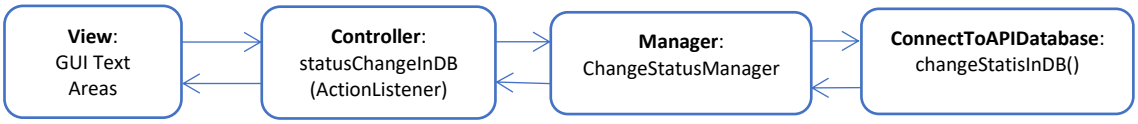
Should the user uncheck either of the PDF and screenshot checkbox's, and the update Bug button pressed, a confirmation message will be displayed to the user indicating to click yes to delete the file. If yes is clicked, then the file(s) will be removed from the database.

Should the Bug have existing files already, and the user updates with newer ones and the update Bug button pressed, the application will delete the previous files in the database and update it with the newer files.

- Delete Bug: By selecting a row in the table and clicking this button, this will delete the Bug object from the AWS RDS and any files in the AWS S3 database associated with that ticket.



- Change Status: Selecting a row in the table and this button, allows the user to close this ticket. The ticket will be still visible in the Table but grayed out. The screenshot and PDF labels on the bottom right of the application will also be greyed out. The user can delete this entry at a time of choosing.



6.5. Panel 3:

This panel is used to display information regarding Network Issues, Ajax loader, Open Analytics Window, View bug in a Web Browser and View Screenshots/PDF's.

- Label displays an animated ajax loader image indicating to the end user that the application is currently working. The ajax loader animates when it is doing the following
  - Loading the table with Bug entries from the database
  - Creating a Bug object in the database
  - Updating a Bug object in the database
  - Deleting a Bug object in the database
  - Changing the change status of a Bug
  - Clicking 'View in HTML Form' button
  - Extracting text from an Image.
- Status Label that will display the current status of operation
  - If the application can/cannot connect to the database
  - Successfully/Failed created an entry in the database
  - Successfully/Failed updated an entry in the database
  - Successfully/Failed deleted an entry in the database
  - Successfully/Failed changing the status of an entry in the database.

  All status update strings are propagated from the network connection ConnectToAPIDatabase() class, back through the manager classes, back to the main controller and finally displayed in this status label in the main View.

  If a valid row within the table is not selected, and the Update Entry, Remove Entry, Change Status or View in HTML Format buttons are pressed, a message pane launches prompting the user they have not selected a valid row.

- PDF Button will be in enabled should the selected row in the table contain a PDF file
    - If a PDF file exists on the selected row of the table, and this button pressed, then the file will be downloaded from the database and stored on the local machine
    - If the file originated from the local machine when creating the tuple, then when the user selected the file for upload, a copy would be placed in the absolute path of this application under a folder called “Downloaded_Files”. This is to ensure that the file does not have to be downloaded from the database and is not a get request to Amazon which charge for each request
    - If the PDF file is stored on the local machine then this button will render and display the first page of the PDF file on the PDF button. The rendering occurs in the MouseListenerClass.java file, and uses the pdfBox.jar file for this purpose
    - The PDFManager class is used to view the PDF file by checking if the PDF file is stored locally. If yes, then it runs the following command "rundll32 url.dll, FileProtocolHandler” to open that specific PDF file. If the local machine is a Linux then it uses the command "xdg-open". This application is for operation on a Windows machine.

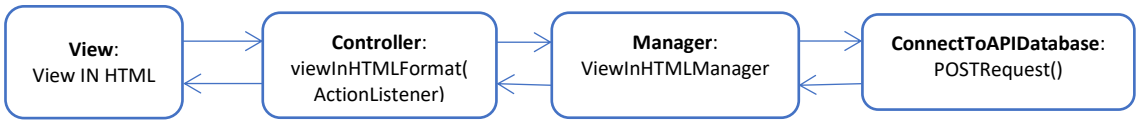
- Screenshot Button will be in enabled should the selected row contain a screenshot
    - If a screenshot file exists on the selected row of the table, and this button pressed, then the file will be downloaded from the database and stored on the local machine
    - If the file originated from this machine then when the user selected the file for upload, a copy would be placed in the absolute path of this application under a folder called “Downloaded_Files”. This is to ensure that the file does not have to be downloaded from the database
    - The ScreenshotManager class is used to resize and open/view the screenshot file in a separate JFrame window. The directory of the saved file is visible at the bottom of this window should the user require direct access to it.

- Clear Text Area Button: when clicked, all the text areas and check boxes are reset from Panel one. All dropdown menus will reset to index 0.

- Button to launch and view the Bug property values, screenshot and PDF file (should they exist) in a web browser. The ViewInHTMLManager class accomplishes this by generating a

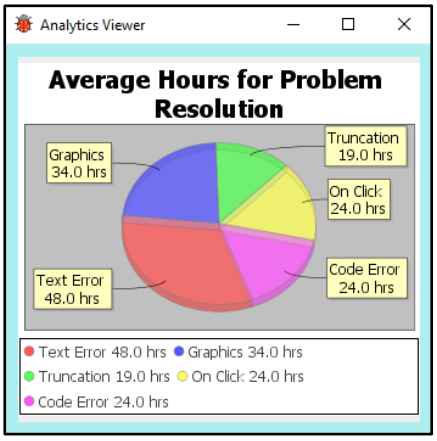
HTML file to view the Bug object values and Image/PDF file, by embedding in to the HTML file. The HTML file is generated every time the 'View in HTML Format' is pressed.

If the file(s) do not exist on the local machine, then the class 'ViewInHTMLManager' brings down all files associated with that specific Bug, and generates the HTML to view all content associated with that Bug.



- Analytics button to launch a separate analytics window containing the time it took to fix a Bug. Only Bug objects that are closed are displayed on the pie chart due to opened Bugs only have a start date timestamp for when the Bug was first created and no timestamp for when it was finished.

  The pie chart displays the average time it took to fix each classification of Bug, divided by the number of occurrences of the classification.



Logging (Log4j) will be enabled so should issues arise the administrator/developer will be able to track where they occurred. The file will be saved in the Absolute Path/BugReporter_Logs folder.

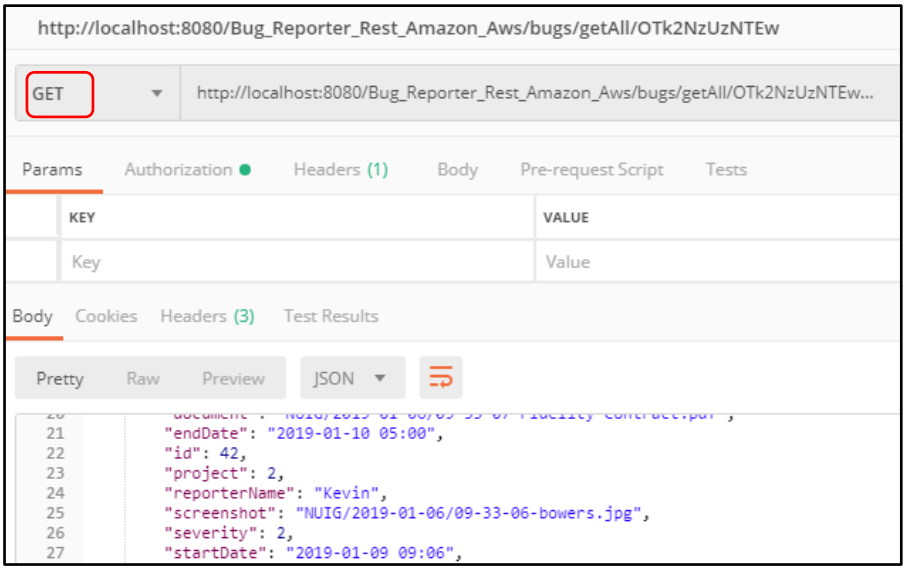
The following data will be recorded for the exception caught

1. Date and Time
2. Exception Type (Error, Info)
3. Class file name
4. Line of code the exception arose at
5. Method name.

7. Testing

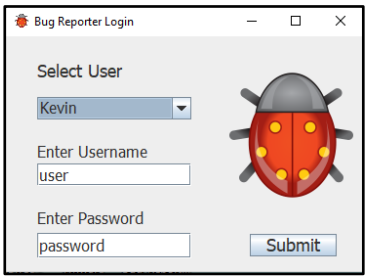
Initial testing on the REST API running locally was conducted using POSTMAN, whereby I was hitting the local AIP with a GET request to return an array list of objects.

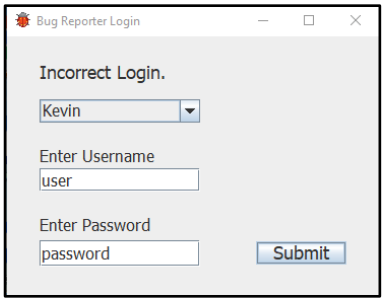
Testing was successful, as this did return an array list of Bug objects.



See section 9 for list of POSTMAN commands used for testing the REST API with CRUD requests.

User Login: When you launch the application you will have to select a user from the drop down menu. There are currently two users (Kevin, Desmond). Each has a username 'user' and password 'password'. This is set in the relational database. When the submit button is pressed, the user, username and password is sent to the API where it is checked with users in the "credentials" table of the relational database.





If an incorrect user or credentials are entered, then they will receive an Incorrect Login message. This message is also given should there be no network available to connect to the REST API for user verification and a session Id.

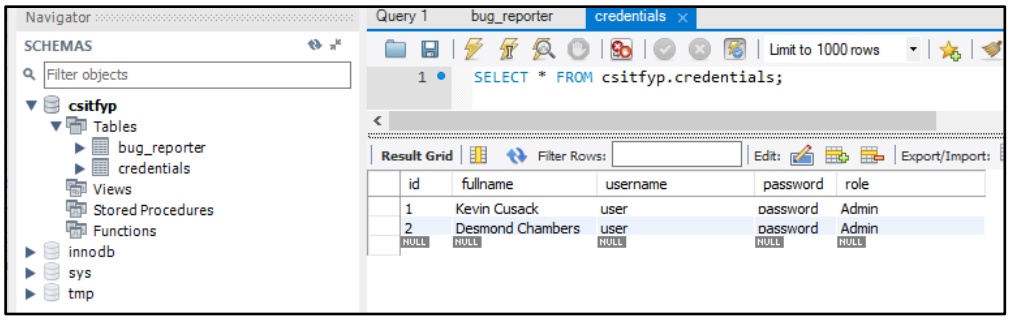
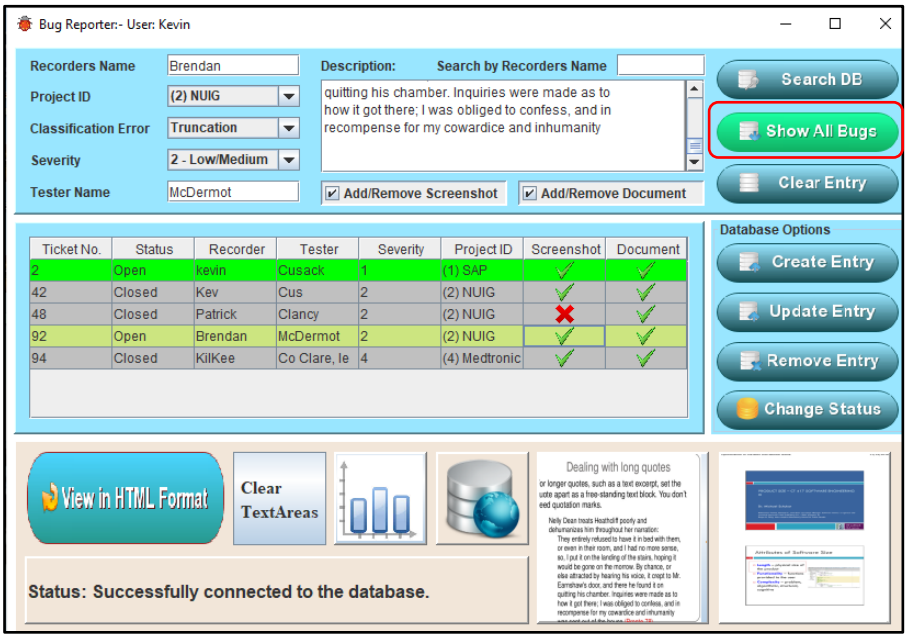


Figure 9: WorkBench - AWS RDS of approved users

Create Object: When the user successfully logs in, clicking the green 'Show all Bugs' will return all objects from the RDS.



To clear the interface, click the 'Clear TextAreas' and start entering new Bug details. Check the 'Add/Remove Screenshot' and you will be prompted to browse for a screenshot file. When selected you will then be prompted with a message stating do you wish to extract text from the image file. For testing purposes I used an image containing text from a 'James Joyce' image. From the 'Add/Remove Document' checkbox I selected a PDF file. I then pressed the 'Create Entry' button.

The Amazon RDS auto increments the primary key (pid) so no ID is passed in to the database with a prepared statement.

Before a prepared statement is executed, the "SetAutoCommit" is set to false, so if the database can execute a successful create statement, then it will be applied by calling "SetAutoCommit" to true.

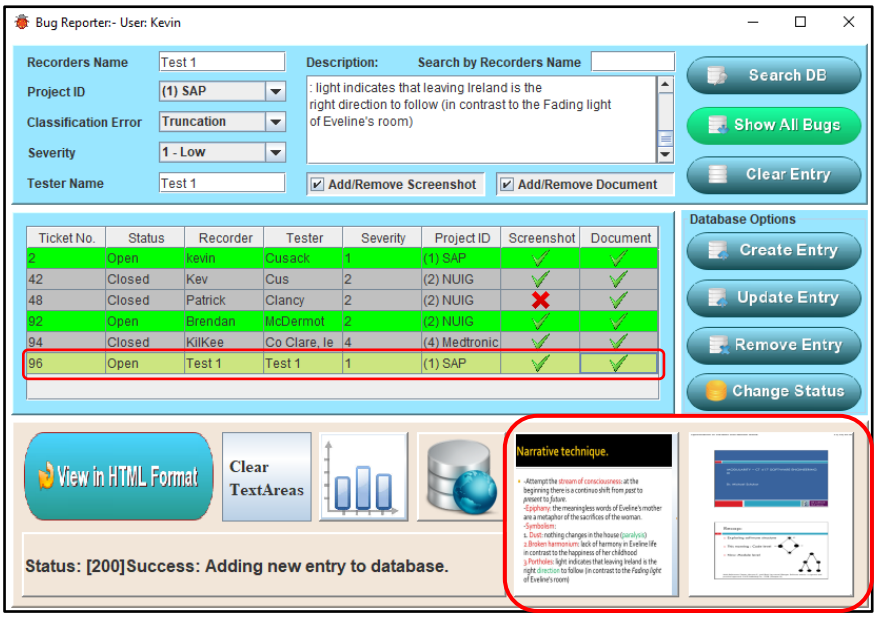


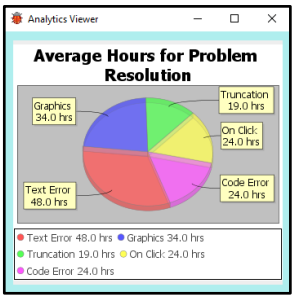
Figure 6: New Bug Entry with rendered images

Screenshot now shows a new 'Ticket No. 96', with the screenshot and PDF rendered in its own label in the bottom right. In the description text area is the text extracted from the image.

Update Object: I tested the update by unchecking and then rechecking the 'Add/Remove Screenshot' check box. This will remove the previous file and update the database with the new file which it allows you to browse for. This was tested by verifying the screenshot image was rendered in the bottom right hand side of the GUI.

Entering 'kevin' in to the search area on the top of the GUI and clicking search populated the table with just one ticket no (2) for Bug objects recorded by.

Change Status: Selecting an 'Open' row in the table and clicking 'Change Status' button will change the status to closed. The status will now have a finish date in this object, whereby clicking the graph icon will create a pie chart with the average time to complete a ticket.



Remove Object: Selecting a row and then clicking the 'Remove Entry' button will delete the object in the RDS and any files associated it with it in the non-relational database. This was checked with workbench to ensure the object was deleted in the RDS and in the AWS S3 to ensure the files were removed.

severity	project	screenshot	document	startDate	endDate
1	1	SAP/2019-02-28/08-08-58-ST Localisation1.png	SAP/2019-03-19/09-20-17-11. Modularity.pdf	2019-01-09 12:06	2019-02-25 08:18
2	2	NUIG/2019-02-28/08-09-24-ST Localisation2.png	NUIG/2019-02-24/12-12-06-Cusack Kevin PEPReport.pdf	2019-01-09 09:06	2019-01-10 05:00
2	2	No	NUIG/2019-02-24/12-12-46-Fidelity Contract.pdf	2019-01-15 01:18	2019-01-16 11:47
2	2	NUIG/2019-03-19/11-12-58-James Joyce2..jpg	NUIG/2019-03-19/09-20-55-12. Product Size.pdf	2019-03-04 09:59	2019-03-08 10:54
4	4	SAP/2019-03-16/07-17-01-Fredson Bowers.jpg	Medtronic/2019-03-16/07-18-22-Fidelity Contract.pdf	2019-03-16 07:17	2019-03-16 19:18
1	1	SAP/2019-03-19/02-03-25-James Joyce..jpg	SAP/2019-03-19/02-03-27-11. Modularity.pdf	2019-03-19 02:03	2019-03-19 14:39

If a selected row contains an image file, PDF or both, and the user clicks the 'View in HTML Format', then this application will download those files and display them in a web browser.

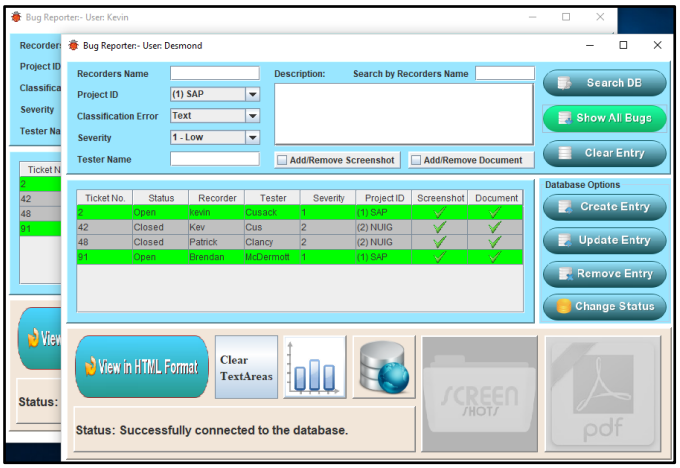
Testing Session Id:

The following is on the client side, where if the user try's to do any request and the session Id has expired, then the client will request a new session Id with the current user login information. Print statements were added to the client code to see that the client was requesting a new updated session Id. The print statements were removed when finished testing.

```
login [Java Application] C:\Development\Eclipse\eclipse\jre\bin\javaw.exe (4 Mar 2019, 09:45:43)
CLIENT: Current DateTime Is: 2019-03-04T09:45:55.866Z, Session Expiry DateTime Is: 2019-03-04T09:46:48.102Z
CLIENT: OK: 582742452, CurrentDate is LESS than SessionExpiryDate...
CLIENT: Current DateTime Is: 2019-03-04T09:47:01.459Z, Session Expiry DateTime Is: 2019-03-04T09:46:48.102Z
CLIENT: The CurrentDate is GREATER than the Expiry Date, So i need a NEW SessionId.
NEW SessionId returned is: 514407277
CLIENT: Current DateTime Is: 2019-03-04T09:47:06.025Z, Session Expiry DateTime Is: 2019-03-04T09:48:01.468Z
CLIENT: OK: 514407277, CurrentDate is LESS than SessionExpiryDate...
```

The session Id is created on the REST API and a Hashmap is used to store the SessionID (key) and a Datetime (Value).

The following are two Bug Reporter Clients running, and you can see the size of the Hashmap.




```
Size of HashMap is 0
Size of HashMap is 2
Size of HashMap is 2
```

8. Conclusion

This project gave me insight into using development tools used in industry, and how when structured you can use them at certain times of the software development cycle. Case in point; before any GUI was built I created the REST API and ran it locally on my PC. Apache Tomcat was used to host it and I used POSTMAN to test that the API was working as expected. Alternatively I could have built it and have it hosted on AWS, but this would be vert time consuming uploading the war file after every code change. I found POSTMAN to be user friendly and it is definitely a tool that I would use again.

I developed the Bug Reporter Client using the Model-View-Controller design pattern. This allowed me to split the application in to sections

1) The View been the GUI
2) The Controller mediating between the GUI and the business model
3) The Model was the business model, with such data as Bug objects.

I found that as I progressed through this project, that the controller class was getting too big i.e. 1200+ lines of code. So I split it up in to manager classes. This allowed me to structure even further. I created a new class for each of the CRUD operations. Another class I created was called 'ImagesManager' whose purpose is to import images in to this application. Another manager class was created that updates the GUI when the user clicks anywhere in the table.

I taught that the REST API would be the problem area when it came to development as setting up the Amazon cloud environment I figured would be problematic, but it was actually the GUI that took most of my time as there is a lot of moving parts there. The GUI was accomplished by adding small amounts of java code and testing. When testing was successful, I added more lines of code and tested once again. There was times that writing code would brake previous code and I would have to back track on that code. The use of using break points to see where variables were changing and to what values was very helpful.

The main take away from this is design the code in such a way that is structured such as manager classes, and when you have to add another feature you could create another manager class that

handles that. Looking at my code now, I could navigate it quite easy and would never have taught it turned out the way it did.

9. Appendix A

The following are the URL commands used to test the REST API running on the local machine.

Get the Session Id: The "user:password" is encoded in base64.

http://localhost:8080/Bug_Reporter_Rest_Amazon_Aws/bugs/getSessionId/dXNlcjpwYXNzd29yZA==

Get All Bug Objects:

http://localhost:8080/Bug_Reporter_Rest_Amazon_Aws/bugs/getAll/OTk2NzUzNTEw

Add a new Bug Object:

http://localhost:8080/Bug_Reporter_Rest_Amazon_Aws/bugs/addBug/MTIzNDU2Nzg5

The following was added in to POSTMAN and the new Bug Object was created in the database.


```
{
   "id": 0,
   "severity": 2,
   "project": 2,
   "active": 0,
   "bugClassification": 2,
   "reporterName": "Pat",
   "testerName": "Clancy",
   "description": "Hi There",
   "screenshot": "NUIG/2019-01-11/10-18-39-Untitled.png",
   "document": "NUIG/2019-01-11/10-18-43-Fidelity Contract.pdf",
   "startDate": "",
   "endDate": ""
}
```


Delete Bug:

http://localhost:8080/Bug_Reporter_Rest_Amazon_Aws/bugs/deletebug/NDQ=/MTIzNDU2Nzg5

Change Status of a Bug Object:

http://bugreportersunday-env.jbmbcxixcs.eu-west-1.elasticbeanstalk.com/bugs/changeBugStatus/OA==/MTIzNDU2Nzg5

Update Bug:

This was tested using the GUI with little updating using POSTMAN.

10. Appendix B

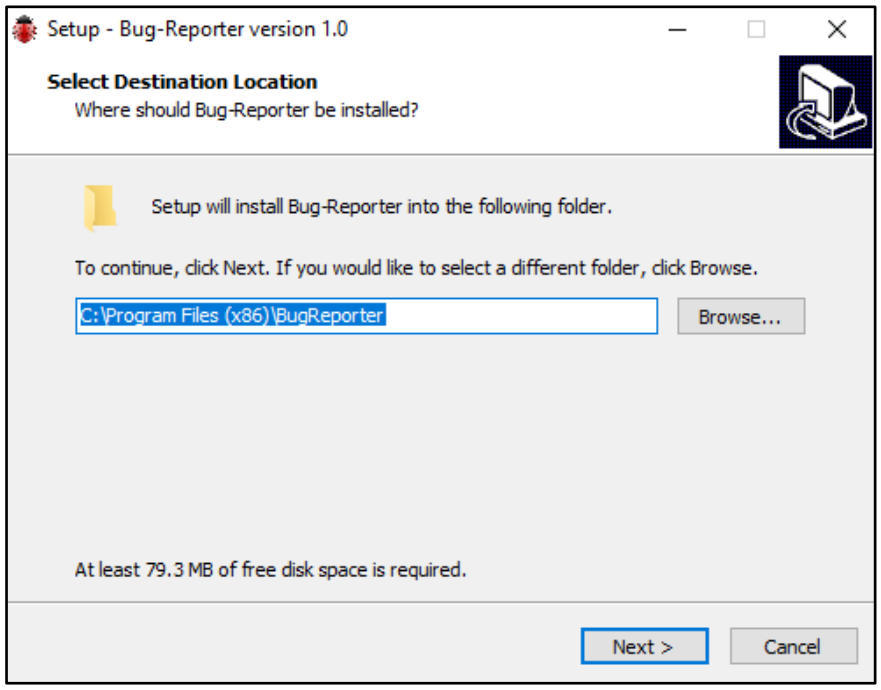
The Windows Setup Installer can be downloaded from the following repository.

https://www.dropbox.com/home/BugReporter_Setup_Installer

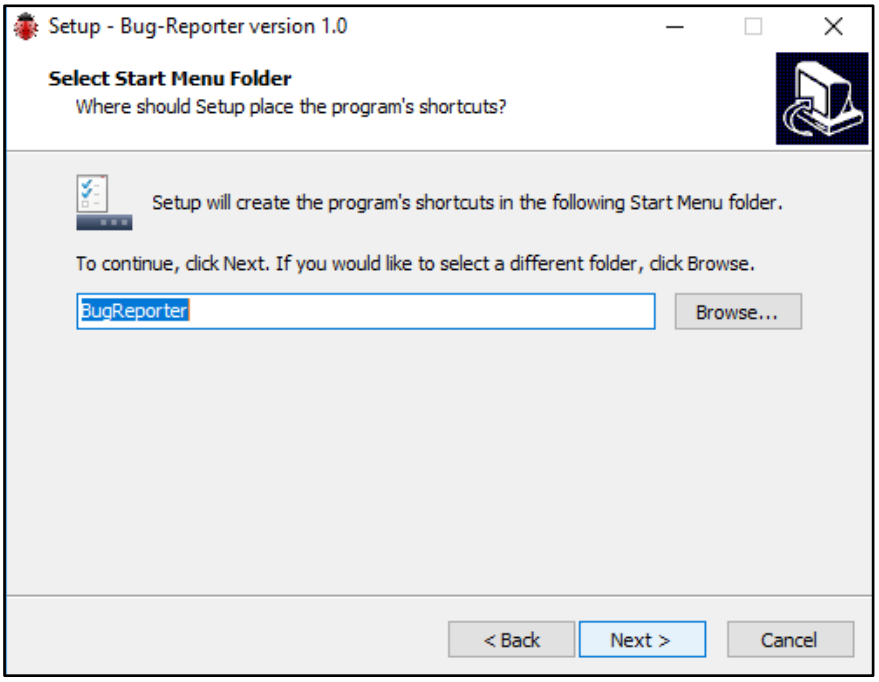
Double click the setup.exe and select the default installation directory for Bug Reporter or browse for one of choosing.

The following contains step by step installation instructions for the Bug Reporter Client to be installed on a Microsoft Windows computer.

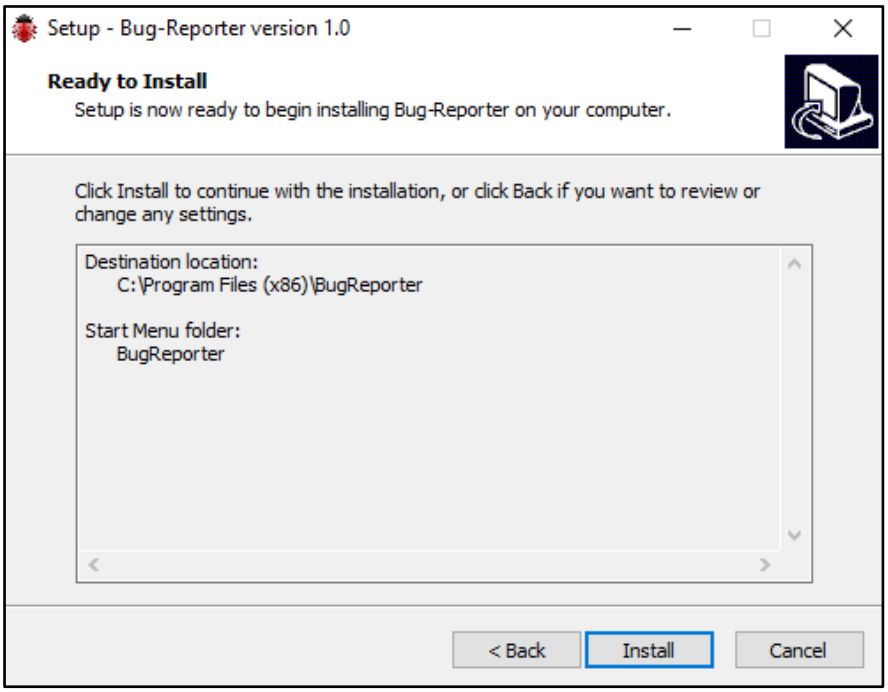
Click 'Next'



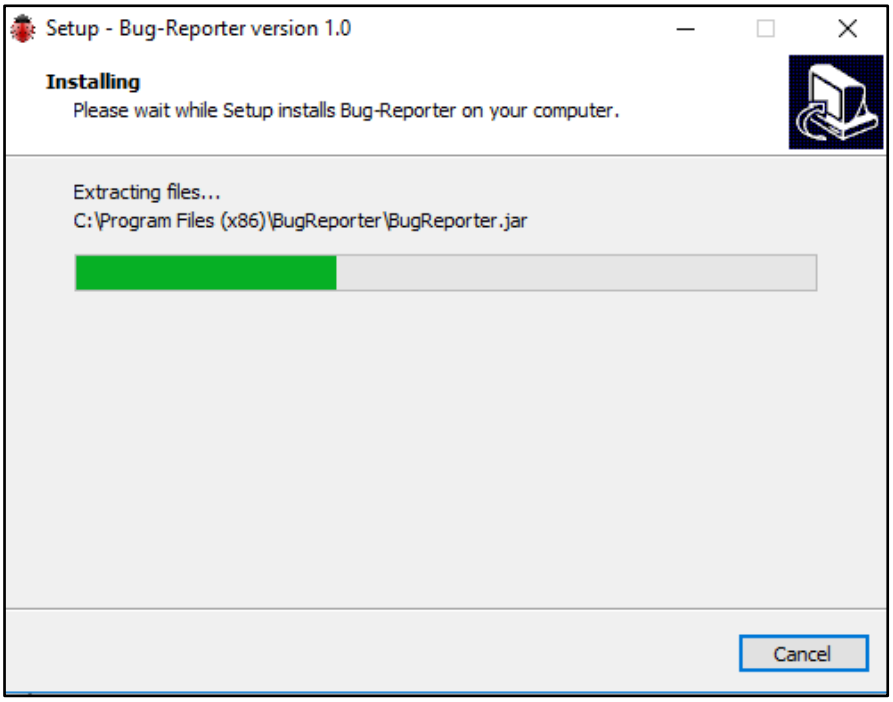
Select default Start Menu Folder or select one of your choosing and click ‘Next’.



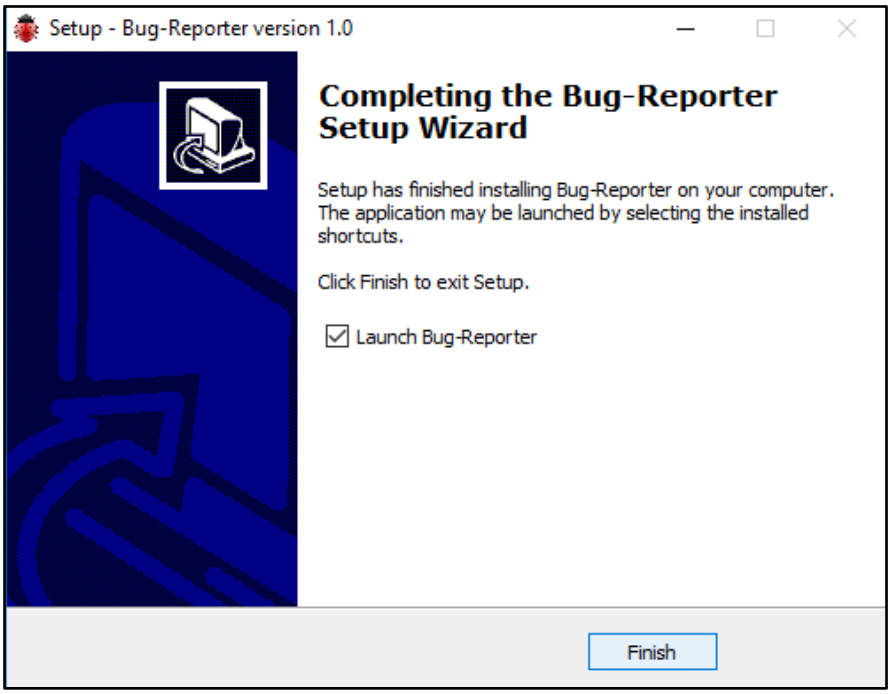
Click ‘Install’ to commence installation of ‘Back’ to change installation directory.



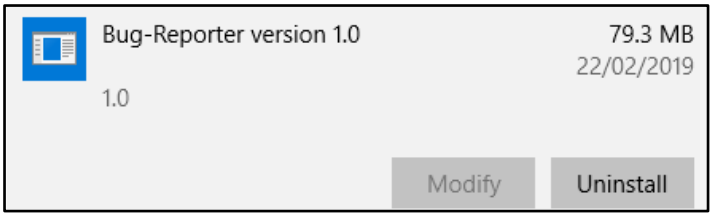
Installation commences.



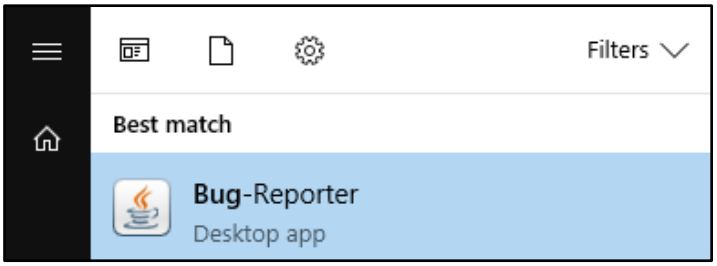
Click ‘Finish’ to finalize the installation.



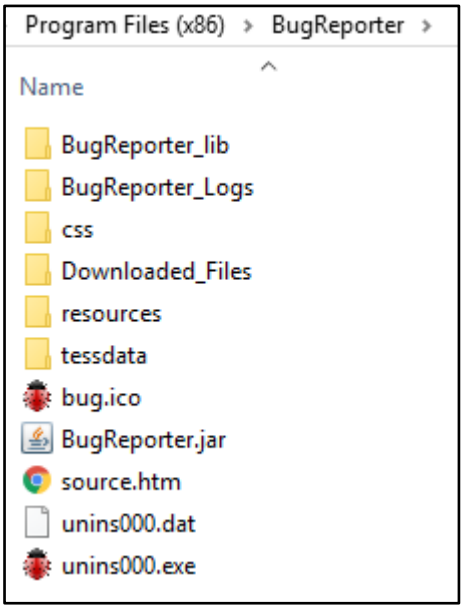
Application is now visible in the ‘Add Remove Programs’. This application can also be uninstalled from here.



The application creates a desktop shortcut and can also be launched from the start menu by typing bug...



When installed successfully, you will see the following files in the default Bug Reporter Client directory.



11. Appendix C

The following is the Login information to access Amazon AWS resources used for deploying the REST API and the following web-services.

- Amazon Elastic Beanstalk
- Amazon E2 (Running Linux Machine to host the REST API)
- Amazon Relational Database (RDS)
- Amazon S3 (Store files on this non-relation database).

Browse to the following link and enter the login information below

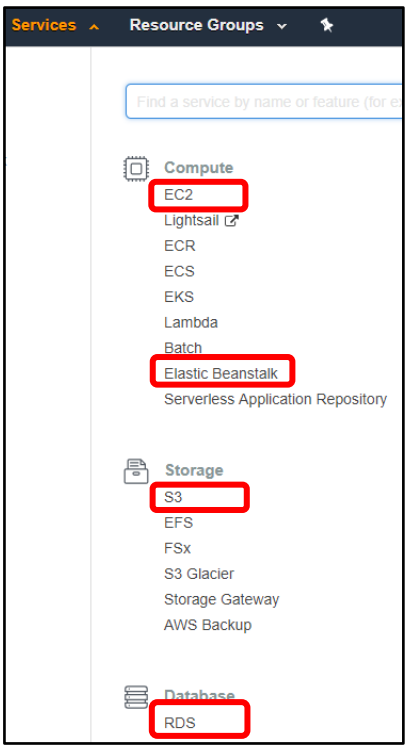
https://signin.aws.amazon.com/signin?redirect_uri=https%3A%2F%2Fconsole.aws.amazon.com%2Fbilling%2Fhome%3Fregion%3Dus-east-2%26state%3DhashArgs%2523%26isauthcode%3Dtrue&client_id=arn%3Aaws%3Aiam%3A%3A934814114565%3Auser%2Fportal-aws-auth&forceMobileApp=0

UserName: kev17404@yahoo.co.uk

Password: Cusask174

Click the 'Services' tab on the upper left hand side and you will be presented with the following web page.

For the FYP I used the following services.



The following gives a description of the services used and some the options that are available for administration purposes.

Select ‘Elastic Beanstalk’, and from the upper left hand side. Select EU (Ireland) from the drop down menu next to the logged in user. On creating an instance of the Elastic Beanstalk, I choose Ireland as the location of where the Bug Reporter API should be running from. The closer the hosting server is to the clients, the faster the connection.

Select the ‘BugeporterSunday_env’ from the environments drop down menu.

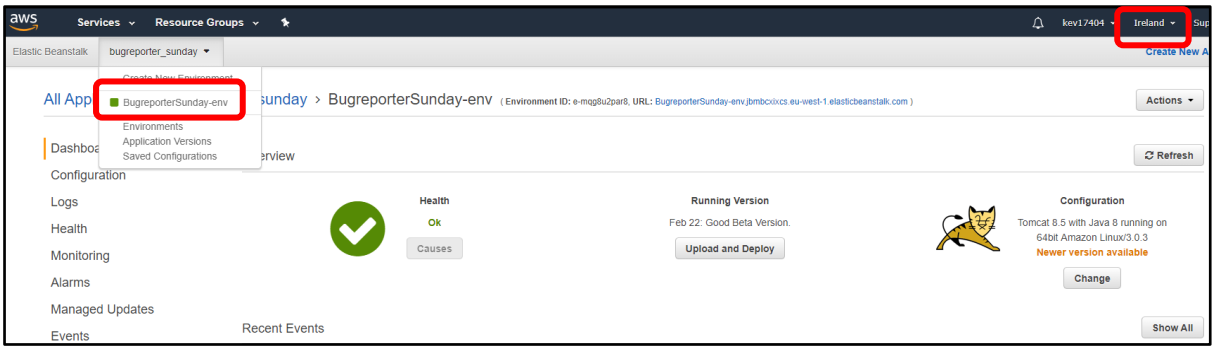


Figure 7: Elastic Beanstalk - Linux Server

This shows the status of the Linux web server hosting the REST API. I can also upload a war file for deployment from here.

Select ‘EC2’, from the Services web page and you will be brought to the EC2 dashboard. From the Resources section, click the running Instances link.

This section shows the status of the Linux server hosting the REST API web-service.

The description section gives such information as the IP address of the Instance running, what zone it is running in and so forth.

Clicking the Monitoring tab, displays the Linux server status such as CPU usage, Disks Reads/Writes, Network In/Out (bytes)…

Figure 8: Monitoring the Linux Server

Select 'RDS' from the Services web page and you will be brought to the Amazon RDS dashboard. Select the DB Instances link in the Resources section. You can now see the csitfyp RDS is available and running. You can create security groups and assign IP addresses from here.

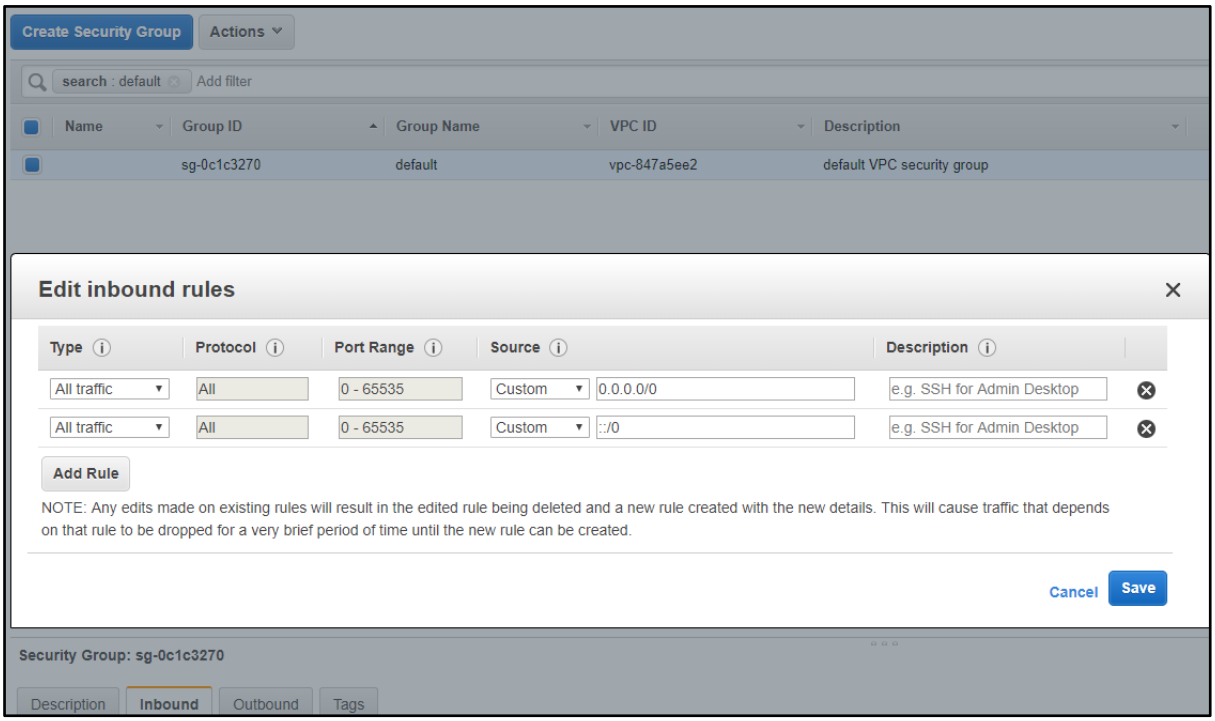


Figure 9: RDS Security

Clicking the 'Monitoring' tab displays RDS utilization such as CPU, DB Connections, Storage and Read/Write IOPS per second.

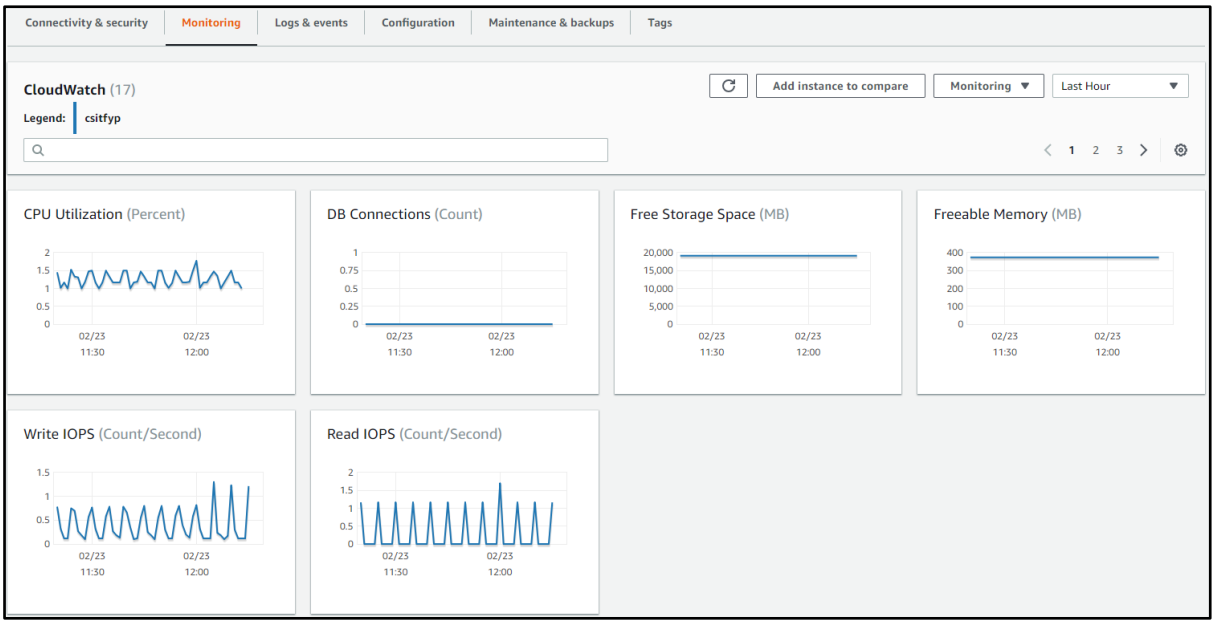


Figure 10: Monitoring the RDS

You can create backups of a RDS database should want to shut down the RDS for maintenance and later bring that database back online.

Select 'S3' from the Services web page and you will be brought to the Amazon S3 dashboard. You can create folders to hold files, along with assign permissions to files and folders. Files and folders can be deleted manually here.

When you add and/or delete files from a Bug object, the files will also be added and/or deleted in here.

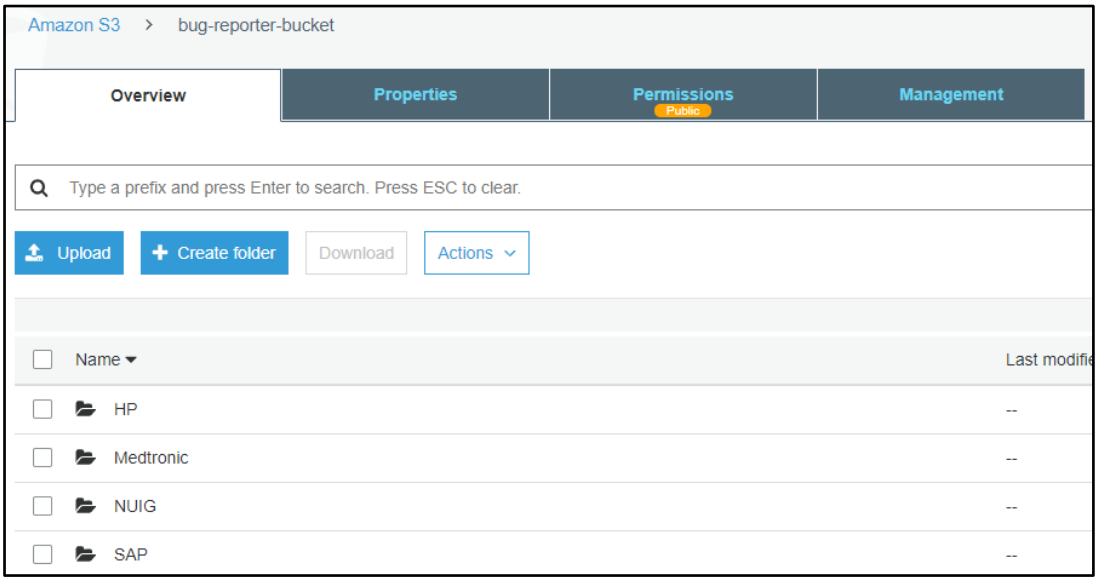


Figure 11: Non-Relational Database S3

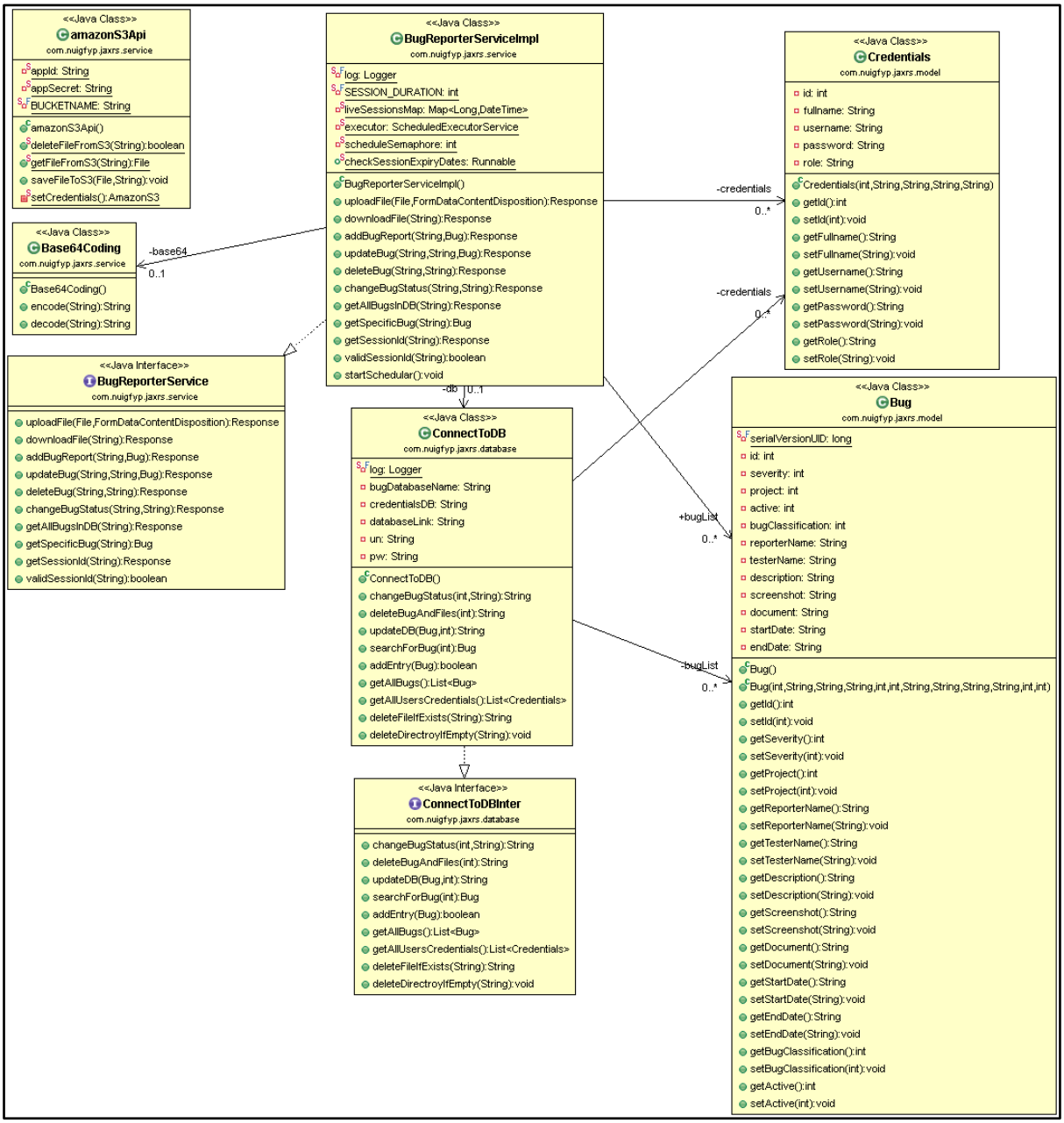
12. Appendix D

Bug Reporter Client UML Diagram:

This is the main controller for the Bug Reporter Client, and all the classes associated with this controller.

REST API Class UML Diagram:

This is the Bug Reporter REST API, and all the classes associated with this class.



References

Buttonoptimizer (2018) 'Call-to-Action Button Generator' [online], Available at http://buttonoptimizer.com/ (accessed 10th January 2019).

Wikipedia (2018) 'Model–view–controller' [online], Available at: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller (accessed 10th October 2018).

Oracle (2018) 'Class SwingWorker' [online], Available at: https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html (accessed 22nd December 2018).

Info-Graphics:

Iconfinder (2018) 'Database web icon' [online], Available at: https://www.iconfinder.com/icons/66065/database_web_icon (accessed 5th December 2018).

Iconfinder (2018) 'Bug icon' [online], Available at: https://www.iconfinder.com/search/?q=bug (accessed 13th February 2019).

Google (2019) 'txtFile Icons' [online], Available at: https://www.google.com/search?rlz=1C1CHBD_enIE831IE831&q=image+to+text+icon&tbm=isch&source=univ&sa=X&ved=2ahUKEwjQibCt-c7gAhVPKuwKHcqKDHsQ7Al6BAgAEA8&biw=2075&bih=986#imgrc=3FepC4ELcnKHJM: (accessed 3rd March 2019).

Google (2019) 'Change icon' [online], Available at: https://www.google.com/search?rlz=1C1CHBD_enIE831IE831&q=change+status+png+icon&tbm=isch&source=univ&sa=X&ved=2ahUKEwiZw7Onsc_gAhWrUxUIHTh0B3oQ7Al6BAgEEA8&biw=2075&bih=1041#imgrc=Y0Cf7GRGciXrzM: (accessed 3rd March 2019).