

# Intelligent Code Repair iCR for Java

## Private Platform User Guide 2.2



# Table of Contents

<b>1.0</b>	<b>Introduction</b>	<b>4</b>
<b>2.0</b>	<b>Overview</b>	<b>5</b>
<b>3.0</b>	<b>Getting Started</b>	<b>7</b>
3.1	<i>Installing iCR for Java</i>	7
3.2	<i>Managing your service</i>	8
3.2.1	<code>icrforjava -a &lt;IP Address&gt;</code>	8
3.2.2	<code>icrforjava -d &lt;directory-path&gt;</code>	8
3.2.3	<code>icrforjava -l &lt;license-key&gt;</code>	9
3.2.4	<code>icrforjava -r</code>	9
3.2.5	<code>icrforjava -p &lt;default-passphrase&gt;</code>	9
3.2.6	<code>icrforjava -h / icrforjava ?</code>	9
3.2.7	<code>icrforjava -c &lt;cmd&gt;</code>	9
3.2.8	Opening Ports	10
3.3	<i>Accessing your source code</i>	10
<b>4.0</b>	<b>Using the Navigator</b>	<b>12</b>
4.1	<i>Connecting to the Navigator</i>	13
4.2	<i>Setting your private passphrase</i>	13
4.3	<i>Licensing Authorization</i>	14
4.4	<i>The Navigator top banner</i>	14
4.5	<i>The Analysis Engine status</i>	15
4.6	<i>Selecting Your Source Code</i>	15
4.6.1	Using a Cloud-based VCS	16
4.6.2	Using a local project	18
<b>5.0</b>	<b>Using the Analysis Engine</b>	<b>20</b>
5.1	<i>Initiating an analysis</i>	20
5.2	<i>Monitoring the analysis</i>	21
5.3	<i>Interrupting the analysis</i>	21
<b>6.0</b>	<b>Reviewing Your Results</b>	<b>23</b>
6.1	<i>Reviewer summary and filters</i>	24
6.2	<i>Filter by Directory pane</i>	25
6.3	<i>Filter by Category pane</i>	25
6.4	<i>Handling Results</i>	26
6.4.1	Reviewing a fix	26
6.4.2	Accepting a fix	28
6.4.3	Rejecting a fix	29
6.4.4	Undoing a fix	30
6.4.5	Providing feedback	30
6.4.6	Applying the fixes	31
6.4.7	Cases needing manual attention	31

6.4.8	Ending a Reviewer session	32
7.0	When You Are Complete	33
	Appendix A – List of Supported Fixers	34
	Appendix B – GitLab OAuth Setup	40
	Appendix C – BitBucket OAuth Setup	42
	Appendix D – Manual License Activation	45
	Appendix E – Example Summary Report	47

## 1.0 Introduction

Thank you for choosing OpenRefractory's *Intelligent Code Repair (iCR) for Java (iCR)*. iCR combines source level static analysis and machine learning for examining programs to detect security, reliability, and compliance issues and combines that with behavior-enhancing code refactoring technology to create safe and reliable corrections for those flaws. This results in code free from many serious security vulnerabilities and programming errors.

*iCR for Java* is offered as both an on-demand service, available through a cloud-provider like Amazon's AWS or Microsoft's Azure, and as a subscription service for private platform deployment. In both versions of the service, customers can choose to analyze and repair projects which are managed by well accepted cloud-based Version Control Systems such as GitHub, GitLab or BitBucket, or projects which are already copied into a project folder.

This User Guide will provide the details about the specific features of the private platform version. For details about the cloud-based service, please refer to the *iCR for Java Cloud Deployment User Guide 2.0*.

In the private deployment version, you subscribe to the service through contact with OpenRefractory. With either a paid subscription or, possibly, a limited time free trial subscription, you will be provided a package that contains everything you will need to operate *iCR for Java*.

The *iCR for Java* package contains the **iCR Navigator**, **Analysis Engine** and **Reviewer**. You use the Navigator to help you to select the projects that you want to make available for processing. The Navigator launches the Analysis Engine as needed and the Reviewer is used to browse through the fixes that were generated. The reviewer uses a "diff" window so that you can see the original code alongside the fixes that were generated. You can also use the reviewer to browse all the source in the affected file if you wish.

*iCR for Java* runs on a dedicated server that you provide as a Docker image. It is expected that this server has the Docker container infrastructure installed. From the Docker site: "Docker provides a way to run applications securely isolated in a container, packaged with all its dependencies and libraries." This allows you to install iCR for Java as part of your Development Operations infrastructure with confidence that it will not disrupt your infrastructure. The server may be dedicated hardware within your development network or could be part of private, cloud-based development environment.

This guide will show you how to connect to your Cloud-based version-control system (VCS) with support for GitHub, GitLab and BitBucket systems. Or you may choose to process projects which are already extracted from the VCS and positioned into project folders.

You select a project for analysis, initiate an analysis of that project, and then review the results. The review process presents to you all the flaws detected and allows you to review each correction whereby you can accept or reject the recommended fix. For accepted fixes, you can then incorporate them back into your project.



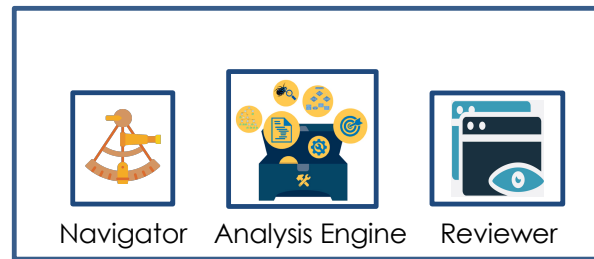
## 2.0 Overview

The following is a quick overview of how to use *iCR for Java*. It is assumed that you have installed and launched *iCR for Java* from the package that you received from OpenRefactory (see [3.0 Getting Started](#)). It is also assumed that you know the IP address of the host server where you installed iCR for Java.

Using that IP address, connect to the service using a standard browser of your choice.

iCR for Java consists of 3 major components:

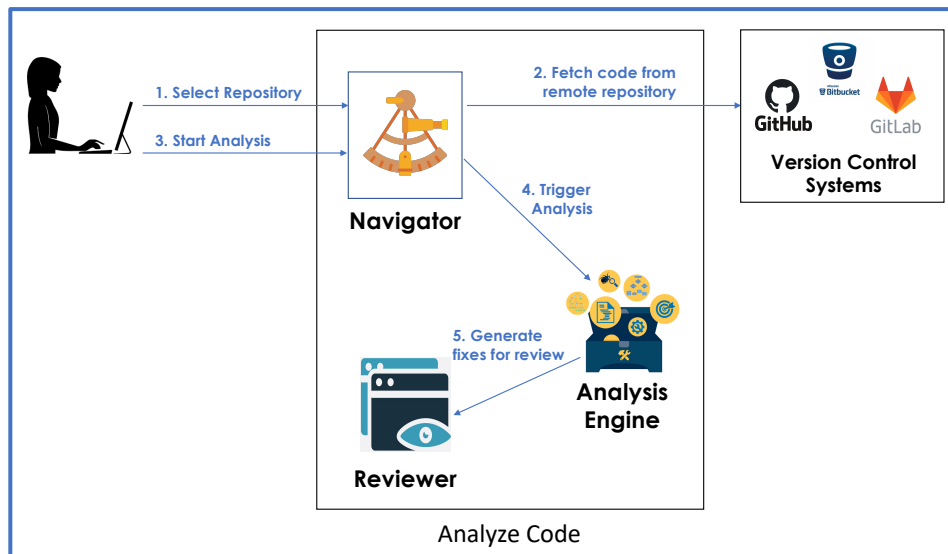
1. The **Navigator** is the main component with which you interact;
2. The **Analysis Engine** analyzes source code and generates fixes;
3. The **Reviewer** helps you to review, approve/reject and apply the fixes.



Using the Navigator, you will:

1. Direct the Analysis Engine to scan the source code of your Java project; and
2. Initiate the Reviewer(s) to examine the generated fixes and accept or reject them.

To understand how each of these steps is executed, let's first look at how to select and analyze a project. Figure 1 outlines the steps taken to select the code to be analyzed and initiating the analysis.



**Step 1.** Select the repository that you are using to manage your source code. This may be a version-control system (VCS) available on the cloud or as an in-house service. *iCR for Java* supports your choice of GitHub, GitLab and BitBucket systems.

**Step 2.** Navigator connects with the repository and fetches the source code to the server. The Navigator will use OAuth to authenticate with your VCS service. Once connected with the VCS, Navigator will present you with a view of all the available repositories associated with your User ID. You may then clone any repository you wish to examine, and you will have all of the branches available for analysis.

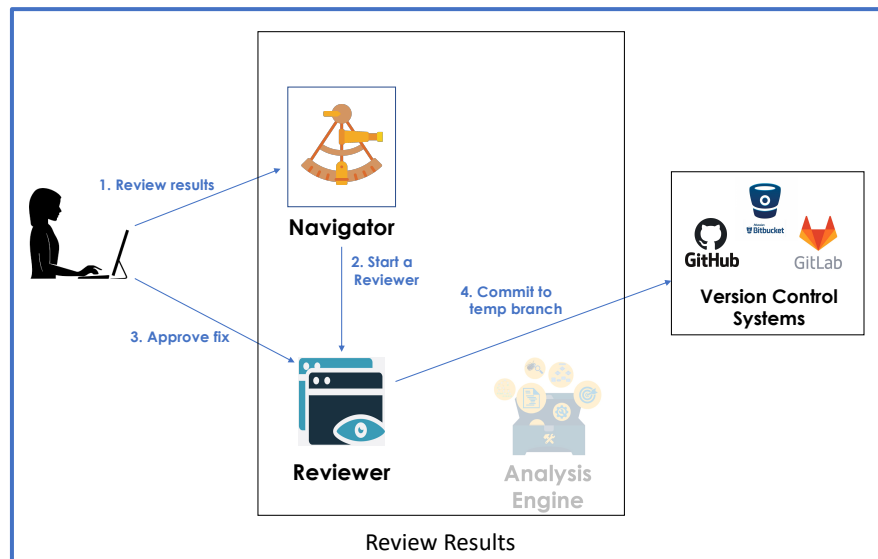
**Step 3.** Pick a branch to analyze and simply click on the Analyze button on Navigator.

**Step 4.** Navigator will start the Analysis Engine as a background process. You may monitor the progress from Navigator in a separate browser tab. For a long running analysis, you may choose to receive a notification and exit iCR. Only one analysis can be run at a time.

**Step 5.** The Analysis Engine analyzes the source code and prepares the fixes. You may choose to have the Navigator send you an email notification when the analysis completes and the fixes are ready.

*iCR for Java* employs a suite of scalable deep analysis tools to provide a comprehensive analysis of your program's flow with emphasis on tracking references across methods. From that analysis, *iCR for Java* then employs a broad family of what we call **Fixers** which are focused on common Java programming flaws and coding standards such as the SEI CERT Oracle Coding Standard for Java. See *Appendix A* for a list of supported fixers in *iCR for Java*.

Once a project has been analyzed and fixes generated, they are available for review. The diagram below outlines the steps taken to perform a reviewing session.



**Step 1.** Return to the Navigator when analysis is complete to review the fixes. Select any branch that has been analyzed and click on **Review** button on the Navigator. You may review past results even when the Analysis Engine is running on something else.

**Step 2.** The Navigator starts the Reviewer component in a separate tab.

**Step 3.** The Reviewer allows you to browse all of the fixes and gives you the opportunity to accept or reject various fixes. Any number of your developers can review and approve fixes concurrently. After approving fixes, you can **Apply** them to your project. If there are fixes that you are not clear about or you think are incorrect, you can let our developers know by filling out a quick feedback report for that particular fix.

**Step 4.** The Reviewer creates a temporary branch in your repository with the potential fixes placed there as `git` commits. This gives you a standard way of choosing when you want to roll these fixes into your project branch(es).

The remainder of this guide will provide you with all the details needed to help you to run *iCR for Java* on your projects.

## 3.0 Getting Started

### 3.1 Installing *iCR for Java*

It is quick and easy to get going on analyzing and automatically correcting programming errors in your Java projects. With your *iCR for Java* subscription (either paid or a possible free trial subscription), you will have been sent a message with 2 important items:

1. A link to a zip file containing the items listed below; and
2. A file named `license.json` which contains your unique software license.

These items will be explained in this section.

The examples in the guide will be using Linux as the reference platform. Support for platforms other than Linux will be supported in future releases. All of the installation and configuration shown in this section will be performed from a command line interface. So, your first step would be to SSH into your server and login.

Your Linux system must also be configured with two important packages:

1. You must have the `zip/unzip` package. This is frequently already installed on your Linux distribution but, if not, you will need to install it since the *iCR for Java* package is distributed in `zip` format;
2. The *iCR for Java* framework is designed to operate within a Docker environment. Docker allows you to install packages like *iCR* on your private Linux platform and know that it will be protected from other software on your system and your network.

If you need to install Docker, please refer to the Docker installation instructions which can be found here: <https://docs.docker.com/engine/install/>.

**NOTE:**

Once Docker is installed, you will want to follow the common practice of creating a User Group to allow Docker access without requiring root privileges for each user. To learn how to do that, please refer to: <https://docs.docker.com/engine/install/linux-postinstall/>. These instructions assume that you have done that so executing the `icrforjava` commands will not require typing `sudo` before each command invocation.

Once your system is configured with the required packages you can install the *iCR for Java* package.

To begin, from the command line, choose a directory to use for staging the installation. It can be any user directory. Once selected, copy the `license.json` file that was attached to your email to that directory.

Use the link also provided in that email, to download the zip file into the same directory. Use the `unzip` command to expand the contents into that directory. For the examples used here, we will use:

```
/home/userid/tools/icr
```

as the example folder/directory.

The expanded files are:

1. `iCR_for_Java-Private_Platform_User_Guide_v2.1`:  
This guide;
2. `icr-for-java.tar`:  
This is the Docker image that implements *iCR for Java*. Instructions on how to use it are detailed in this guide;
3. `install-icr`:  
This is the script used to install *iCR for Java* on your host system;

4. `icrforjava`:  
This is the command used to manage your server after installation. Use it to start up your *iCR for Java* service, update configuration values or even remove it from your host server completely;
5. `EULA_for_iCR_for_Java`:  
This is the End-User license which gives you the authority to use the *iCR for Java* on your private platform. You must have read and accepted this prior to receiving and installing this package.

With the zip file expanded, install the iCR for Java components. To do that, execute the `install-icr` command. This installation step will require `root` privileges so MUST be done using `sudo`. Run the command as follows:

```
sudo ./install-icr
```

You should see output that looks like the following:

```
034acb3a48e7: Loading layer [=====>] 253.1MB/253.1MB
dd925e02d664: Loading layer [=====>] 1.313GB/1.313GB
0537bb84bd7d: Loading layer [=====>] 720.6MB/720.6MB
Loaded image: icr-for-java:local
```

This will accomplish a number of things for you:

- The Docker image will be installed into your Docker registry under the name `icr-for-java:local`;
- The scripts will be copied to `usr/bin` where they will be accessible to all your developers who have access to your host system;
- The companion files, such as the EULA, will be saved in your `/etc` directory in a new directory named `/etc/icr-data`.

## 3.2 Managing your service

With the files copied and the Docker image installed, you can begin to run the service. One of the items installed is the `icrforjava` command script. This command is used to stop/start the service, update critical parameters used by the service and to uninstall the software if you need to do that.

The usage model for the command is as follows:

```
icrforjava [-a] [-d] [-l] [-r] [-p] [-h] [-c <cmd>] <arg>
```

The options can be combined in a single command. However, the `-c <cmd>` should always go last since it is used to invoke server operations.

We'll begin by looking at the first four options.

### 3.2.1 `icrforjava -a <IP Address>`

The `-a` option is used to provide *iCR for Java* with an IP Address for the server. Typically, the IP address for the server is determined using `localhost`. However, if you are using a cloud-based repository, such as GitHub, there needs to be a way for GitHub to reach the server from outside of the private platform's possibly protected network. In that case, you would need to provide an external public IP address for your firewall to allow GitHub to reach the *iCR for Java* server. You can configure *iCR for Java* with the public IP address using this command.

### 3.2.2 `icrforjava -d <directory-path>`

The `-d` option is used to anchor *iCR for Java* with the point in your host file system where you plan to store projects for local access. While you may access GitHub, GitLab or BitBucket to process repositories managed by those systems, you may also want to analyze and review projects resident in your local file system. You

can also include directories that may be attached via a network attached storage and mounted into your file system. The default anchor point is the `/home` directory under which user directories are normally located within Linux.

### 3.2.3 `icrforjava -l <license-key>`

The `-l` option is used when there is a need to update or replace your activation license. This might happen if you need to move the service to another host. The license is activated when you first run *iCR for Java* (not when it is installed) and connect to the Navigator for the first time. After that, your service is node-locked to the host computer. Should your license become damaged or unusable, this is a way for us to provide you with a new license.

### 3.2.4 `icrforjava -r`

The `-r` option is used to remove a manually activated license. See Section 4.3 to learn more about manual license activation.

### 3.2.5 `icrforjava -p <default-passphrase>`

The default passphrase for *iCR for Java* is set to: `icrforjava`. When you first login to the service, you use the default passphrase to gain access to the Navigator. If the passphrase is set to the default, you will be prompted each time you connect to the Navigator to change it from the default to some other practical passphrase.

Using the `-p` option allows you to reset the passphrase back to the original default or a new default phrase as specified by the string `<default-passphrase>` that you provide. Just remember that, if you change the default, the Navigator will check to see if the default passphrase is in force and will continue to prompt you to change it.

### 3.2.6 `icrforjava -h / icrforjava ?`

Displays the extended usage form for the `icrforjava` command:

```
icrforjava [-a] [-d] [-l] [-r] [-p] [-h] <arg> ? [-c <cmd>]
```

Where:

<code>-a &lt;IP_ADDRESS&gt;</code>	# define the server's public IP address
<code>-d &lt;MOUNT_DIR_PATH&gt;</code>	# define a new root directory
<code>-l &lt;LICENSE&gt;</code>	# change the license key
<code>-r</code>	# remove manually activated license
<code>-p &lt;PASSPHRASE&gt;</code>	# set a new default passphrase
<code>-h</code>	# output this help
<code>?</code>	# output this help
<code>-c &lt;cmd&gt;</code>	
Where <cmd>:	
start	# start the service
stop	# stop the service
uninstall	# remove the service and all associated files

### 3.2.7 `icrforjava -c <cmd>`

The previous four options were used to update or change some configuration values. To actually operate *iCR for Java* you use the `-c <cmd>` option. There are three specific `<cmd>` values that can be used to control your server's operation:

- `icrforjava -c start`  
This is the command that is used to start the *iCR for Java* server. It assumes that you have successfully installed the service on your host machine using `install-icr`. It starts the Docker image from scratch. The first time that `start` is invoked, the service will be node-locked to this host using the license provided in the package that you received. The default passphrase will be used to

access the Navigator for the first time. The default string is: “icrforjava”. You will be prompted to change it once you access the Navigator.

- `icrforjava -c stop`

The `stop` command is used to stop the running instance of *iCR for Java*. The Docker container is stopped and any activity in progress is interrupted. For example, if the service is stopped during an analysis, that analysis will be abandoned. All results from previous analyses, however, are still available and can be viewed again once the service is restarted using the `start` command.

- `icrforjava -c uninstall`

There may be cases where you wish to remove *iCR for Java* entirely from your host system. If so, run the `uninstall` command. This will remove all of the Docker information as well as the directories that were created as part of the `install-icr` process. All results from analyses will be removed. Please be certain that you want to remove everything if you decide to run this command. If you need to restore *iCR for Java*, and you have kept your original package, you can re-install the service using the `install-icr` command again.

### 3.2.8 Opening Ports

*iCR for Java* uses a browser interface to interact with the Navigator and the Reviewer. It may be that the host system for the service is behind a firewall or has security restrictions applied to it to limit network access. For the service to run correctly, a number of TCP ports must be open to the server. Some of these ports may already be open as they allow, for example, remote logins over SSH or other browser access. A couple of the ports are unique to *iCR for Java*. The ports that you need to have open are the following TCP ports:

1. **22** – This is the SSH port to allow you to access your server. You use this to securely login into your host system to be able to run the *iCR for Java* commands.
2. **80** – This is the regular HTTP port to allow browsers to access your server.
3. **443** – This is the secure HTTPS port to allow browsers to access your server.
4. **3002** – This port is used by your Browser to work with the Navigator.
5. **3003** – This port is used by your Browser to work with the Reviewer.

## 3.3 Accessing your source code

*iCR for Java* is designed to work with source code managed by industry leading version-control systems (VCS). In this release, *iCR* supports GitHub, GitLab and BitBucket. *iCR* also allows you to copy or upload a project source tree to your server and analyze it that way if your source code is managed off of the cloud.

Assuming that you are using a cloud-based VCS, you need to authorize *iCR for Java* to access your projects. Once you are logged into your source code control system, *iCR* will connect to your specific repositories and analyze the specific project branches that you identify. In order to do this securely, and to ensure that OpenRefactory NEVER has access to your Users’ login credentials, we employ the industry standard protocol: OAuth<sup>1</sup>.

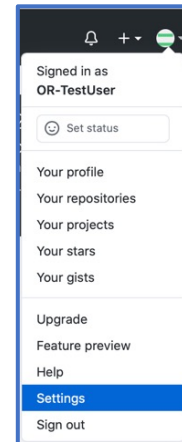
From Wikipedia: “**OAuth** is an open standard for access delegation, commonly used as a way for Internet users to grant websites or applications access to their information on other websites but without giving them the passwords.”

---

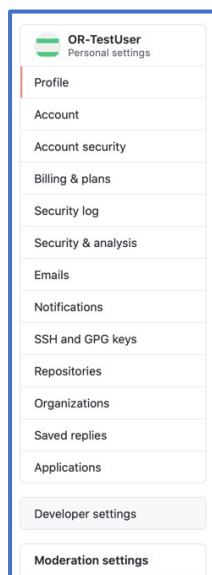
<sup>1</sup> OAuth reference: <https://medium.com/security-operations/what-is-oauth-and-why-should-i-use-it-5aa2f27ce387>

To allow iCR to use OAuth, you must authorize it with your VCS. For these examples, we will be using GitHub.

Similar steps are available for GitLab (see Appendix B for details) and BitBucket (see Appendix C for details).

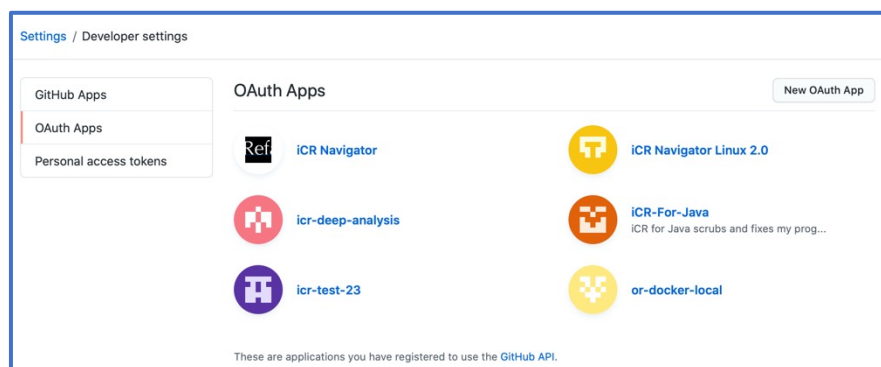


To register a new OAuth app in GitHub, login into GitHub and traverse to “Settings”->



<- Then, select “Developer settings”

From here, click on “OAuth Apps”. This will open the page allowing you to add *iCR for Java* to the set of approved third parties from which you will accept login redirect requests.



Clicking will open the window shown to the right.

You can enter a helpful string, such as “*iCR-for-Java*” for the Application name. The Homepage URL will need to use the IP address of your host system. For the purposes of this guide, we will use an example IP address, `http://3.237.77.219`.

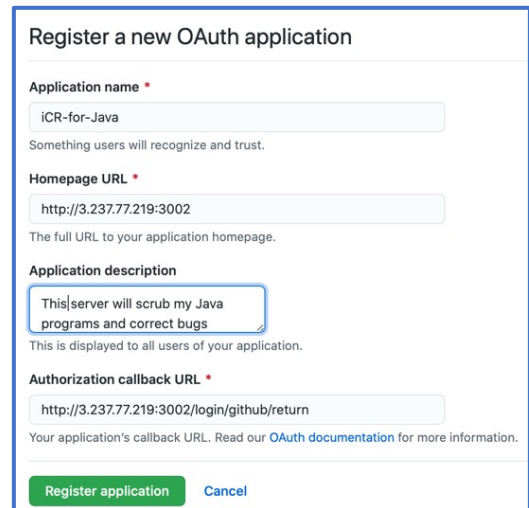
iCR uses port 3002 to communicate with the browser, so that needs to be added to the IP address to create the Homepage URL.

Using the sample IP address, you would enter:  
`http://3.237.77.219:3002`

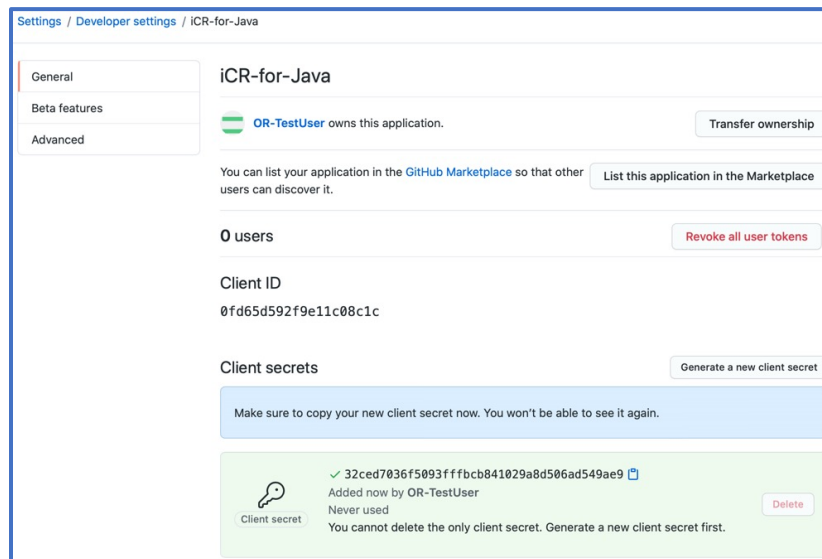
The application description is optional so you can leave it blank. Note that this information will not necessarily be seen by anyone logging into GitHub. Once the OAuth app is created, Users will log in to GitHub using their private credentials and will not see this information.

The Authorization callback needs to provide the server’s URL of the callback, so, enter:  
`http://3.237.77.219:3002/login/github/return`

Clicking on “Register Application” opens a window that asks you to create the secret keys that you will use on your server to authenticate it with GitHub.



The screenshot shows the 'Register a new OAuth application' form. It includes fields for 'Application name' (iCR-for-Java), 'Homepage URL' (http://3.237.77.219:3002), 'Application description' (This server will scrub my Java programs and correct bugs), and 'Authorization callback URL' (http://3.237.77.219:3002/login/github/return). There are 'Register application' and 'Cancel' buttons at the bottom.



The screenshot shows the 'Developer settings' page for the 'iCR-for-Java' application. It displays the 'General' tab with fields for 'Client ID' (0fd65d592f9e11c08c1c) and 'Client secrets'. A 'Client secret' is shown as '32ced7036f5093fffbcb841029a8d506ad54ad549ae9'. There are buttons for 'Transfer ownership', 'List this application in the Marketplace', 'Revoke all user tokens', and 'Generate a new client secret'.

You will need both the client ID (0fd65d592f9e11c08c1c) and the client secret (32ced7036f5093fffbcb841029a8d506ad54ad549ae9). Copy and paste these values in a convenient place as you will need to present them to the Navigator when you first select GitHub as your preferred repository, as described in Section 4.6 [Selecting your source code](#).

A similar process is used to allow access for GitLab and BitBucket. Details of that are given in Appendix B and Appendix C respectively.

With this information setup, you are ready to connect to *iCR for Java* for the first time.

## 4.0 Using the Navigator

This section will introduce you to the iCR Navigator, which is used to help you manage your project analyses. It assumes that you are familiar with the [Getting Started](#) procedures outlined in Section 3 and have already



installed *iCR for Java* on your host server. You know your server's IP address, and have created the OAuth credentials to allow your developers to securely log into your cloud-based version-control service (VCS).

In the examples to follow, we will work with GitHub as the example cloud-based VCS.

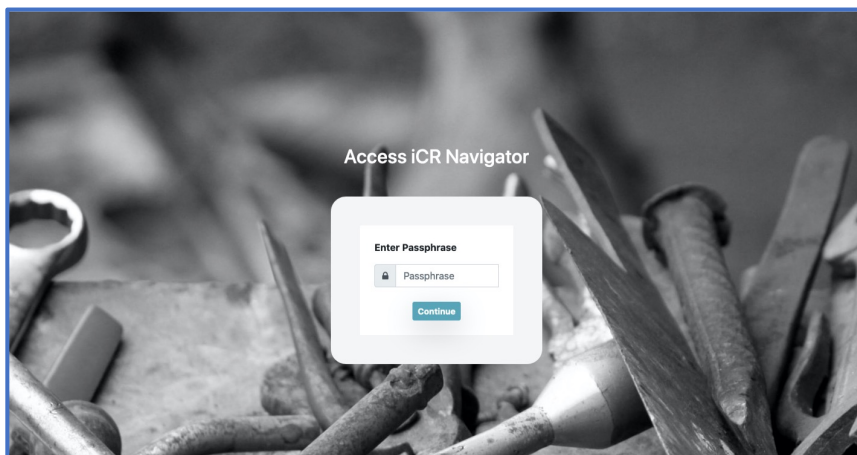
## 4.1 Connecting to the Navigator

The *iCR for Java* service is accessible using any industry standard browser such as Chrome, Firefox, Safari or Edge. To begin working with iCR, you need to access your server via the browser. It is reached using your server's IP address. In our examples we are using `3.237.77.219` as the public IP address. ICR uses port 3002 to reach the Navigator which is the application that will help you to manage your interactions with *iCR for Java*.

Access the Navigator by entering your Server IP address followed by port 3002 into your browser. Using our IP address as an example, this is the URL to enter:

```
http://3.237.77.219:3002
```

Entering this URL will take you to the welcome screen for *iCR for Java*:



You are presented with a window that prompts the user to enter a passphrase. Since the IP address to your Server is public, it is possible to have uninvited “guests” attempt to access your service. To protect the service from unwanted access, you must enter the secret passphrase before you can access the service. The initial, default passphrase is set to `icrforjava`.

**NOTE:** You can reset the default passphrase to a different string if you wish using the command `icrforjava -p <default-passphrase>`.

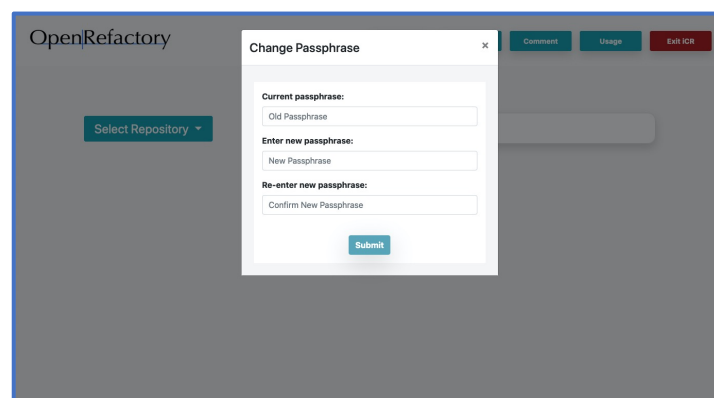
So, type in the default passphrase:

```
icrforjava
```

to enter *iCR for Java*.

## 4.2 Setting your private passphrase

Entering the passphrase will bring you to the Navigator where you will be prompted immediately to alter the passphrase to something other than the default. The phrase should be at least 8 characters long and you may use any alphanumeric values as well as special characters.



### 4.3 Licensing Authorization

Beyond the passphrase authorization step to ensure that only you and your users gain access to the service, OpenRefactory also uses a License Management service to aid in the secure distribution of its software across the various private platforms. While the use of the license service should be invisible to most users, there are cases where a manual activation of the license may be required or there may be other reasons why you may want to be able to validate your license while offline.

Section [3.1 Installing iCR for Java](#) introduced the license file as part of your Private Platform distribution. The license file is unique for each instance of *iCR for Java* that is installed on a private platform. Once activated, the license binds the software to the specific platform for which it was activated. After activation, the software cannot be re-installed on a different platform without working with OpenRefactory to provide a new license.

In many installations, the private platform, while isolated within a closed development network, still has Internet access to reach out from the development network. OpenRefactory uses this to take a “signature” of the platform and performs a handshake with the License Manager service in the cloud to validate the license and activate it. Once activated, the license manager completes the handshake by returning a secure authorization that enables the software to run.

This happens transparently to the user and no additional steps are required. In operation, the license is periodically checked as well so access to the License Manager service must be maintained.

However, OpenRefactory recognizes that, since your source code is valuable and keeping it secure is essential, some installations may demand that no internet access is allowed for any purpose. In such cases, the license cannot be authorized transparently. In such cases, a manual authorization via a proxy is required, and completely offline operation must be allowed.

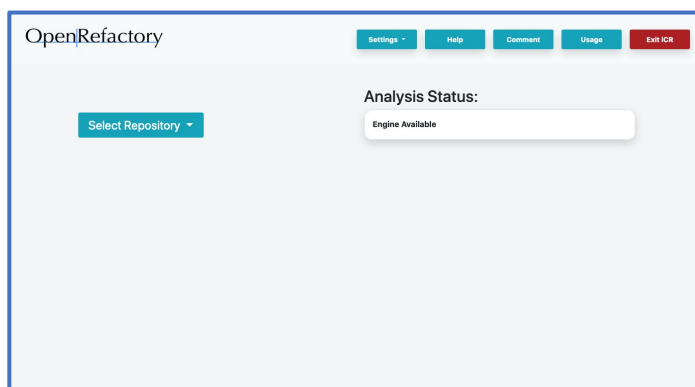
If you need to use this manual process, refer to [Appendix D: Manual License Activation](#) for the procedure to authenticate and authorize a license manually.

**NOTE:**

In the event that you may need to reset a manually activated license, you can use the `icrforjava -l` command to set a new license key. Then you need to remove the manually activated license using the command `icrforjava -r`.

### 4.4 The Navigator top banner

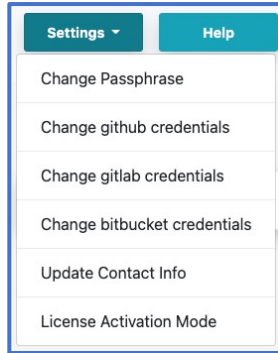
Once the passphrase is updated, you are presented with the Navigator Home screen. From here, you can select and open project repositories with your projects, analyze one or more branches of any of these projects and then, following analysis, you can review and apply corrections to flaws detected in those branches.



At the top of screen, on the right side, you see 5 buttons:

- Settings
- Help
- Comments
- Usage
- Exit iCR

The **Settings** button is used to change or configure a number of optional properties for *iCR for Java*.



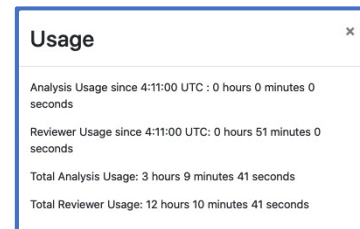
The default passphrase can be updated along with reconfiguring the OAuth credentials for your VCS as outlined in Section [Error! Reference source not found.](#)

You can add contact information if you choose to send OpenRefactory either crash information or other feedback about the operation of iCR.

For some users of the Private Platform deployment,

is used to change the main passphrase and to update OAuth credentials if you have chosen to modify those. The **Help** button will take you to the OpenRefactory Website where you can download help documents, such as this guide, and view the Video Tutorials to help you learn how to use *iCR for Java*. The **Comment** button allows you to send your feedback to OpenRefactory. Your feedback helps us to improve the interface and also helps us to improve the quality of the service by getting feedback concerning potential false positives or improvements on the Fixers.

The **Usage** button helps you to learn how much time you have spent doing analysis and reviewing. This gives you the knowledge of how much time you have accumulated doing both Analysis and Reviewing Fixes.

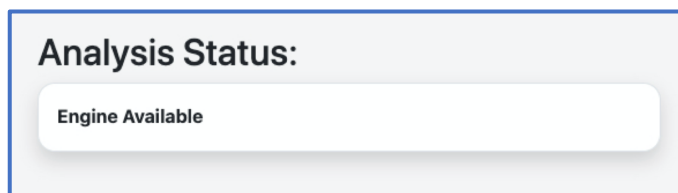


Finally, the **Exit iCR** button takes you out of iCR for Java and back to the *iCR for Java* welcome screen. To re-enter, you would, of course, have to enter your new passphrase.

These buttons will be presented on all other screens in the application so that you can always **Exit iCR** at any time or provide feedback and get help. You may only change the settings from this Home Screen, however.

## 4.5 The Analysis Engine status

Below the top banner, the status of the Analysis Engine is displayed on the right side. Since the analysis process is very RAM and CPU-intensive, iCR currently only support one analysis at a time. The status window lets you know that the engine is available for a new analysis. Or, if an analysis is in progress, it will display a brief summary of the ongoing analysis.



The status shown on the left indicates that the Analysis Engine is available for use.

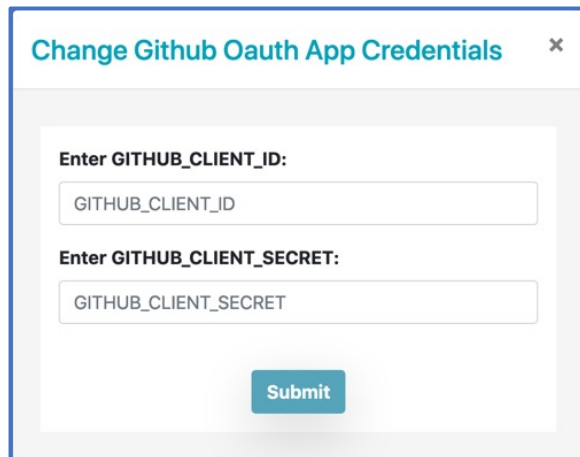
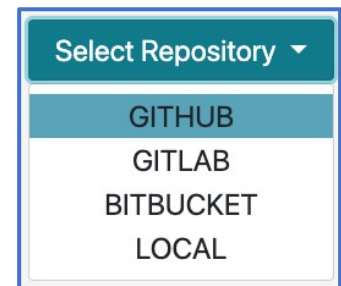
## 4.6 Selecting Your Source Code

Below the banner on the left side opposite the Engine Status is where you start the process of selecting your source code to be analyzed or reviewed.

Your first step is to select the repository where your project resides. *ICR for Java* is best used when working with a commercial Version-Control System (VCS) like GitHub, GitLab or BitBucket. The button is a drop-down menu from which you can select your VCS. Or you can set up a path to a local directory on your server where you have uploaded your project.

### 4.6.1 Using a Cloud-based VCS

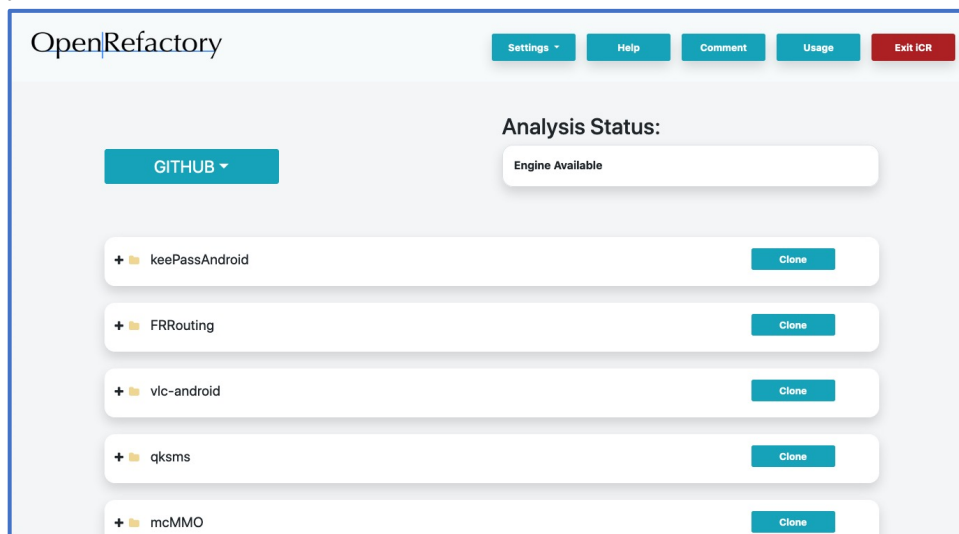
From [Section 3.3](#) Accessing your source code, you will have already set up the OAuth credentials to allow logins to your preferred VCS. Assuming that you have done that, select your VCS from the pull-down menu. For our examples, we will be using GitHub.



The very first time a user attempts to reach GitHub following the OAuth configuration, the Navigator will pop up a window requesting you to enter the Client ID and Secret keys from the OAuth configuration. As explained in [Section 3.3 Accessing your source code](#), hopefully you copied the Client ID and Secret somewhere so that you can enter them here. Once done, users may login into their GitHub accounts without needed to repeat this process.

If there is some reason to change the OAuth Client ID and Secret, you can get back to this window using the [Settings](#) button on the main menu.

If you are already logged into GitHub from earlier activity on your browser, then your repository will become available right away. Otherwise, you will be redirected to the GitHub Website for Authentication. Once logged in, you will now see all of your available GitHub projects.

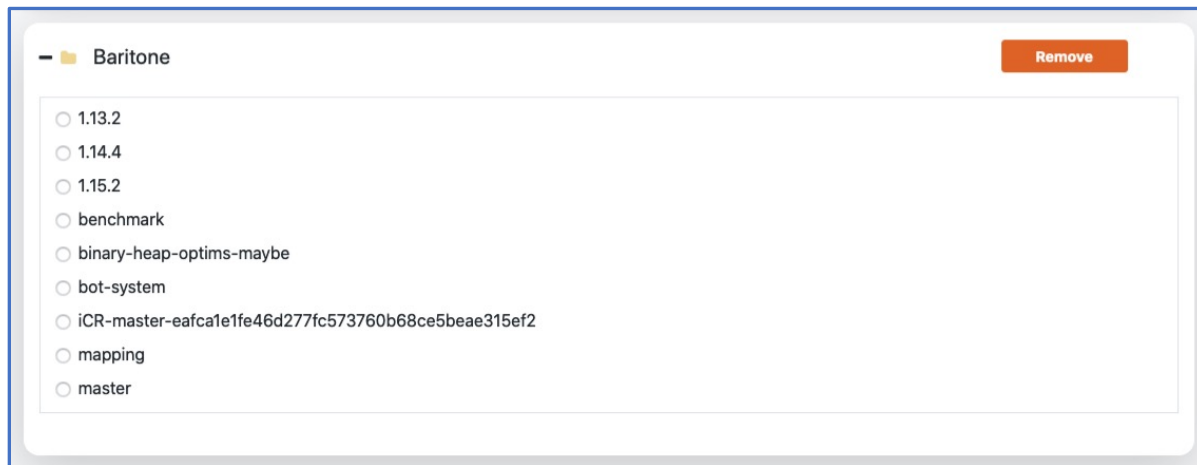


Each project is presented with a "+" sign so that you can open it up to view its branches. Before you can browse the project branches, however, you need to "clone" a copy of the project from GitHub. The [Clone](#) button is to the right of the project name box.

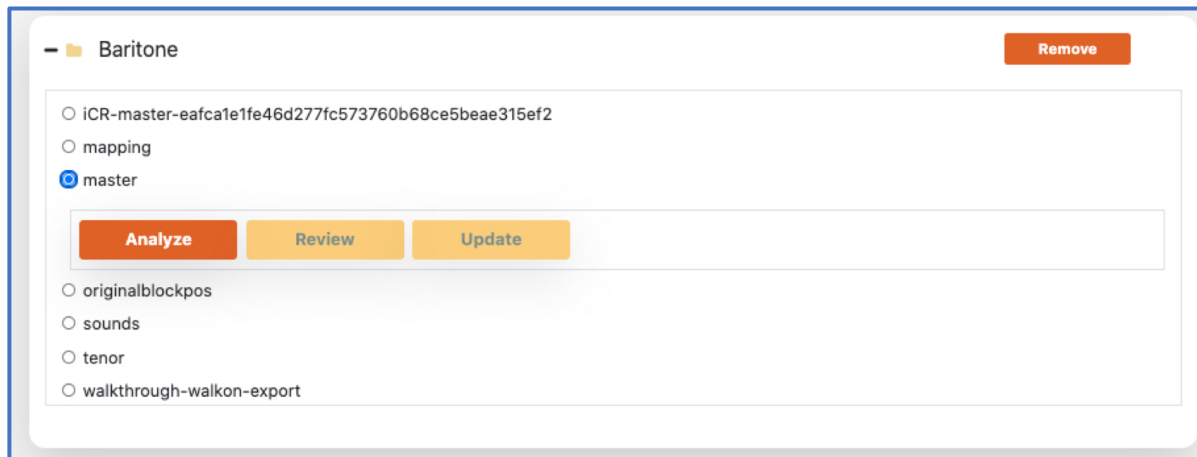


For our example, we will use a project called Baritone, which we show below as cloned and ready for analysis. Note that, once cloned, the [Clone](#) button is replaced by [Remove](#). This provides you with a way of removing a project if you desire. When you remove a project, however, note that ALL RESULTS WILL BE

REMOVED. That is, any analysis that you have performed and not applied to your project will be lost. Clicking on the “+” will enumerate all of the available branches:



In our example, the Baritone project shows many branches. Only one branch at a time can be selected. That is reflected using the radio buttons to choose which branch to examine. Let's look at the master branch.



Selecting its radio button causes three new options to appear:

- **Analyze**
- **Review**
- **Update**

The first button, **Analyze**, is always available and allows you to perform an analysis on the branch. Clicking on it will take you to the Analysis screen which will display status on the ongoing analysis. Section 5.0 will describe the Analysis Engine further.

The second button, **Review**, is not available unless one of more analyses of this branch have been executed. Once an analysis is complete, you would want to click **Review** so that you can begin the process of looking at the detected problems and the corrections that iCR has provided. Section 6.4 Handling Results covers the details of the Reviewer within *iCR for Java*.

The third button, **Update**, is made available when the current status of the branch is out of date with the currently checked-in status. That is, it may be “behind” the current master copy of the branch on the VCS repository. This is not unusual in that your developers may be working with a branch while others are also working on it. If you have made updates to the branch using the Reviewer those changes may already have been incorporated into the “master” branch.

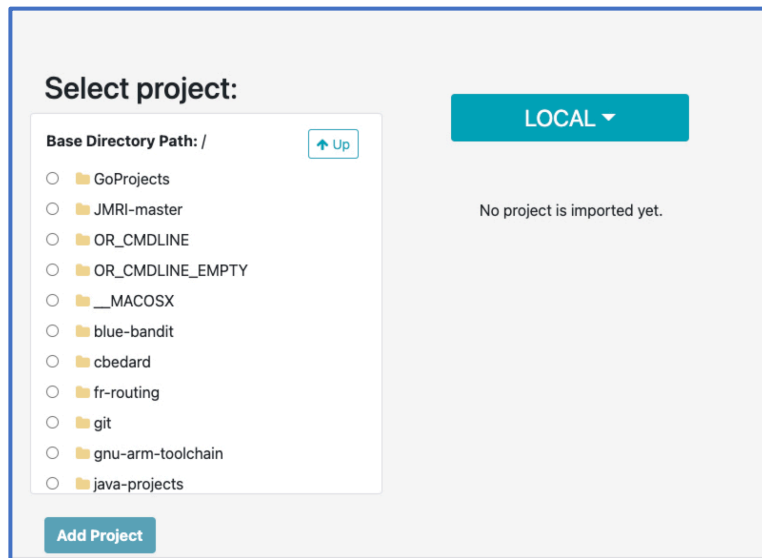
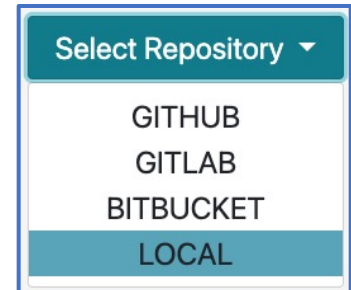
Even though you may not have completed reviewing all of the corrections offered in the last analysis, you may decide to interrupt that process and perform a more up-to-date analysis using the latest “master” version. If so, you can select the **Update** button. This will cause the Navigator to pull-down the most recent version of source code to the *ICR for Java* server.

Naturally, this will make further review of the old source invalid and so should be followed by a click of the **Analyze** button, to perform a new analysis of the updated source code. In such a case, you may also decide to simply **Remove** the project entirely and **Clone** it again. Doing so removes all past history of earlier analyses.

#### 4.6.2 Using a local project

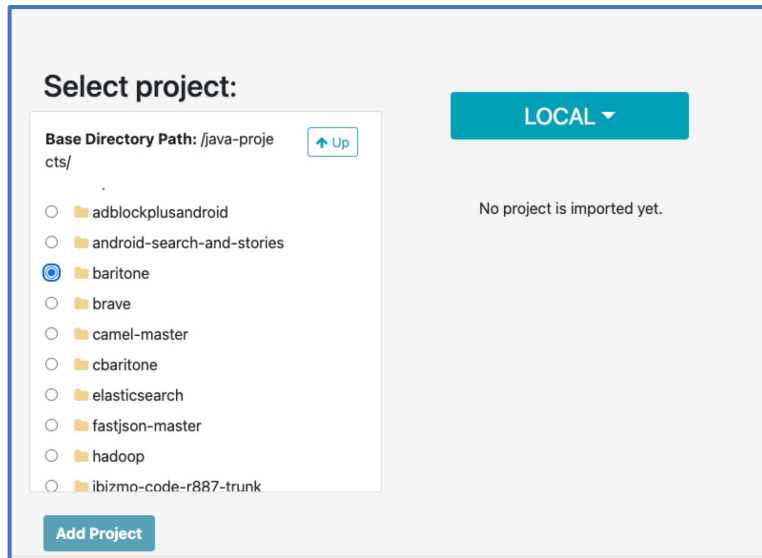
You may choose to not access source code from a cloud-based repository. *ICR for Java* also supports accessing projects that are accessible directly on your server. Any of your developers may be given login access to your server. From a shell console, you can upload projects to it or, you may mount a network attached file system. In this case, it would need to be mounted as a subdirectory of your configured anchor point in the file system.

To select one of these local projects instead of a Git repository, choose the “LOCAL” option on the “Select Repository” drop-down menu. →



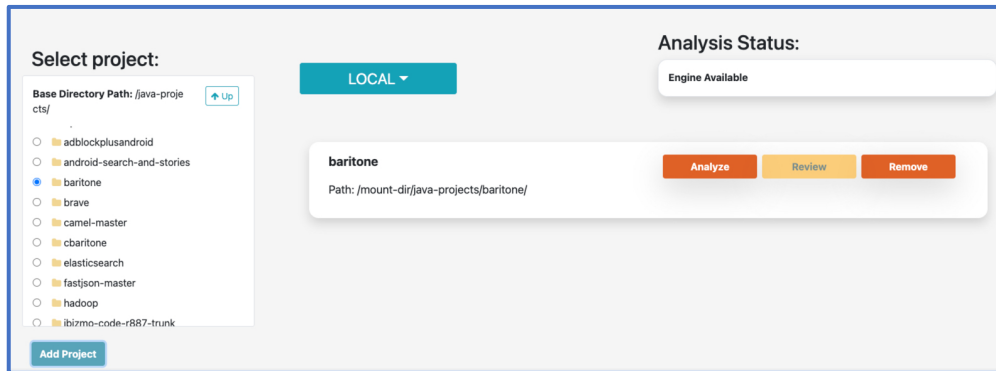
Selecting this option brings up the **Select Project** window on the left side of the screen. By default, it is anchored at `/home`. However, you can reposition this anchor point using the `icrforjava -d <directory-path>` as noted earlier in Section 3.2.2.

From the **Select Project** frame, you can scroll down through directories and subdirectories looking for the desired project for analysis.



In this example, we will select a project called `baritone` which is located within the directory `java-projects` (which is displayed as the **Base Directory Path**).

Clicking on [Add Project](#) brings the project into our list of available projects.



Now the project has been added to the list of locally accessible projects available for analysis. Note that the **Analyze**, **Review** and **Remove** buttons are now available. The **Review** button is grayed out until an analysis is complete and results are available for review.

The **Remove** button allows you to drop this project from the list of available projects.

**Note:**

If you choose to remove a project ALL OF ITS RESULTS will also be deleted.

You can add as many projects as you wish.



## 5.0 Using the Analysis Engine

*iCR for Java* supports the analysis of projects being managed by Cloud-Based services such as GitHub, GitLab or BitBucket as well as locally accessible projects. For the purposes of demonstrating the Analysis Engine and the Reviewer, we will use examples using the cloud-based repositories. The behavior when using locally accessible projects is nearly identical and should be easy to infer from the following descriptions.

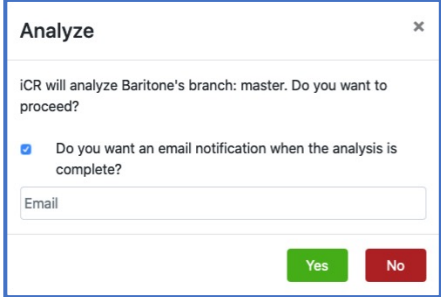
### 5.1 Initiating an analysis

To begin the analysis of a project, you will have logged into your Version-Control System (VCS) such as GitHub, which is being used in our examples. You connected to the Navigator using your server's Public IP address and port 3002 (See 3.0 Getting Started). Once connected to the Navigator, you selected the project you want to analyze, Cloned it and then selected the branch that you wish to analyze (See 4.6 Selection Your Source Code).

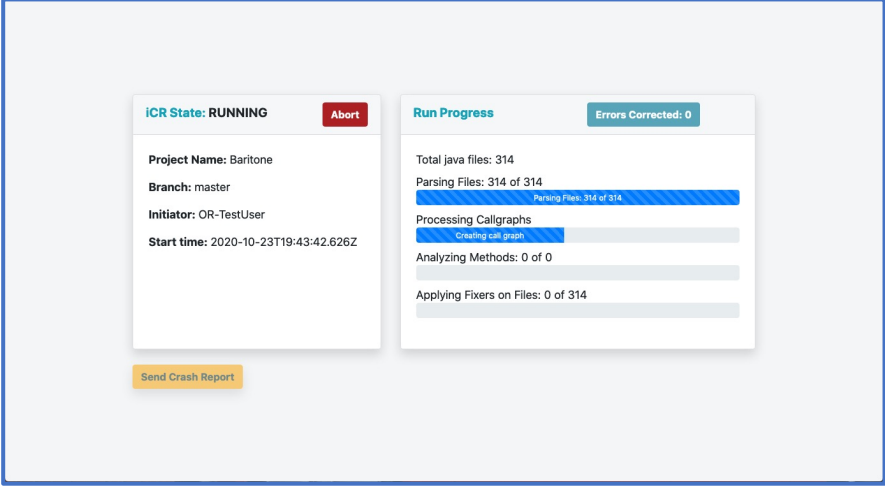
To begin the analysis of the branch, click on the Analyze button.

Smaller projects (< 100,000 Lines of Code) tend to be less complex in terms of number of files and methods. These may be analyzed within minutes. However, larger projects (> 1M LoC) may take much longer to analyze. That's OK. You don't have to sit and watch as it could take many hours for a large, complex project to be thoroughly analyzed.

Clicking the **Analyze** button gives you the option of requesting an email notification when the analysis completes. If you select the box requesting a notification, an email prompt is displayed. Enter the email to which iCR will address your notification.

A dialog box titled "Analyze" with a close button (X) in the top right corner. The text inside says "iCR will analyze Baritone's branch: master. Do you want to proceed?". Below this is a checkbox labeled "Do you want an email notification when the analysis is complete?" which is checked. Under the checkbox is an input field labeled "Email". At the bottom right are two buttons: "Yes" (green) and "No" (red).

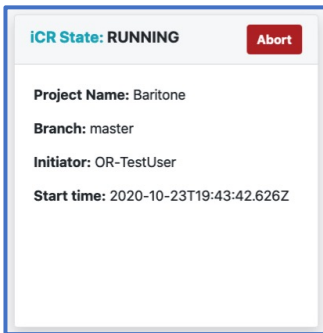
To begin the analysis, click **Yes**. A new tab opens which takes you to the Monitor Analysis screen.

The Monitor Analysis screen displays two main panels. The left panel, titled "iCR State: RUNNING" with a red "Abort" button, shows project details: "Project Name: Baritone", "Branch: master", "Initiator: OR-TestUser", and "Start time: 2020-10-23T19:43:42.626Z". At the bottom of this panel is a yellow "Send Crash Report" button. The right panel, titled "Run Progress" with a green "Errors Corrected: 0" button, shows progress bars for: "Total java files: 314", "Parsing Files: 314 of 314" (100% complete), "Processing Callgraphs" (100% complete, with sub-progress "Creating call graph"), "Analyzing Methods: 0 of 0", and "Applying Fixers on Files: 0 of 314".



## 5.2 Monitoring the analysis

This Monitor Analysis screen displays the progress of the analysis of a project.



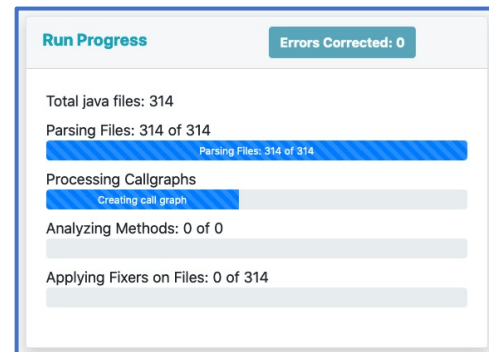
The window on the left displays information about the project including:

- The current state of analysis;
- The name of the project;
- The branch within that project being analyzed;
- The User ID of the User who initiated this analysis;
- Time when the analysis began;
- There is also an **Abort** button to stop the analysis.

The window on the right displays the phases of the analysis and their progress. A total count of the number of errors that have been corrected so far is at the top of the window.

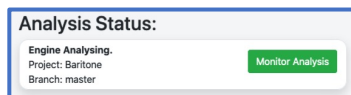
Various phases of the analysis are shown with progress bars to give you a sense of how far the analysis has progressed.

Once analysis completes, the end time is added to the state display.



While you may choose to watch the Monitor Analysis display, as noted earlier, analysis may take a long time for complex projects. In some cases, it may take many hours. So, you can go back to the Navigator tab and **Exit iCR** and return when you are notified that the analysis is done. Or from the Navigator, you may choose to review the results from an earlier analysis in a different project or branch.

If you return to the Navigator home screen, you will see that the Analysis Status has changed.



The status now shows that there is a project running. The name and branch of what is being analyzed is displayed. You will also note that a new button has appeared. It is the **Monitor Analysis** button. Clicking on this button will open a new Monitor tab so that you can check up on progress.

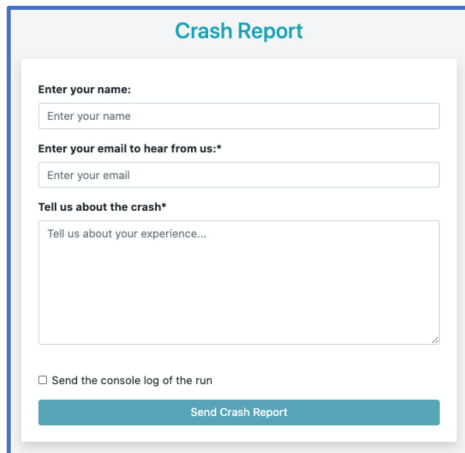
## 5.3 Interrupting the analysis

It could happen that you started an analysis on a project with the wrong branch and want to start over. Or, after watching the progress for some time (remember, many large and complex projects could take many hours to analyze) you may decide to abandon the analysis.

In either case, you may decide to **Abort** the analysis. If so, click on the **Abort** button at the top of the left window. This will terminate the analysis. If you terminate the analysis you will lose any information you produced to that point.

You can help out OpenRefactory determine if there was an issue with your analysis by clicking on **Send Crash Report** which is the button at the bottom left of the left window. Selecting this is at your discretion but it will help us to help you complete your analysis.

When clicked, a crash report window appears. You can enter the experience that you encountered as to why you aborted the analysis. For example, it could be as simple as "I was analyzing the wrong branch". Or it may be that you thought the analysis was not progressing.



**Crash Report**

Enter your name:

Enter your email to hear from us:\*

Tell us about the crash\*

☐ Send the console log of the run

**Send Crash Report**

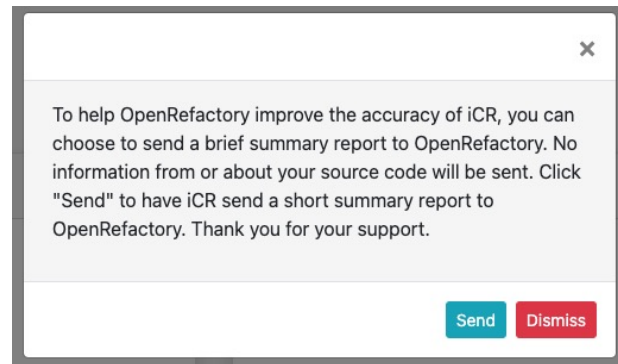
In the latter case, we request that you consider clicking on the option labeled “*Send the console log of the run*”. From the log, we may be able to determine whether the analysis was in the wrong or if it was progressing but just taking longer than you expected.

We want to be clear that the log may contain various snippets of your source code such as Method and Class names. We made this optional so that if you have a concern about OpenRefactory seeing even a tiny fragment of your source code, you can refuse to forward the log. Of course, this means that we will not likely be able to determine the cause of a failure if one occurred. But we believe that having you retain complete control of your source code is necessary for you to be able to trust that we treat your code with the utmost privacy.

When an analysis completes normally, you will receive an email notification if you requested that at the time you initiated the analysis. Once complete, the Navigator will present the prompt on the right.

It is helpful for OpenRefactory to collect some summary data on as many analyses as possible. This helps us to better understand what bugs are most common and where our performance could be improved. Sending a summary is completely up to you and the report contains none of your source code information of any kind. To reassure you that is the case, check out Appendix E which shows a typical summary report from the Baritone project which we have been using as our example in this guide.

Once your analysis for each project is complete and fixes have been applied, you can stop your server or keep it running for others to use.

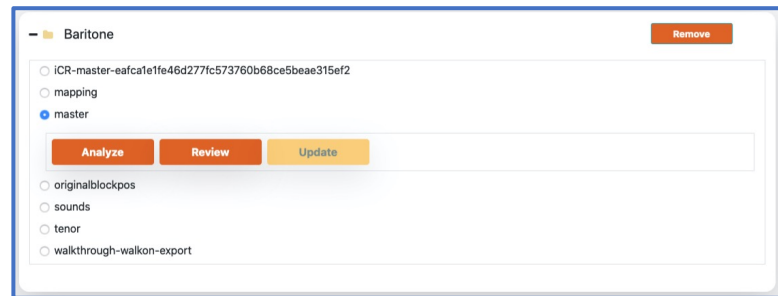


To help OpenRefactory improve the accuracy of iCR, you can choose to send a brief summary report to OpenRefactory. No information from or about your source code will be sent. Click "Send" to have iCR send a short summary report to OpenRefactory. Thank you for your support.

**Send** **Dismiss**

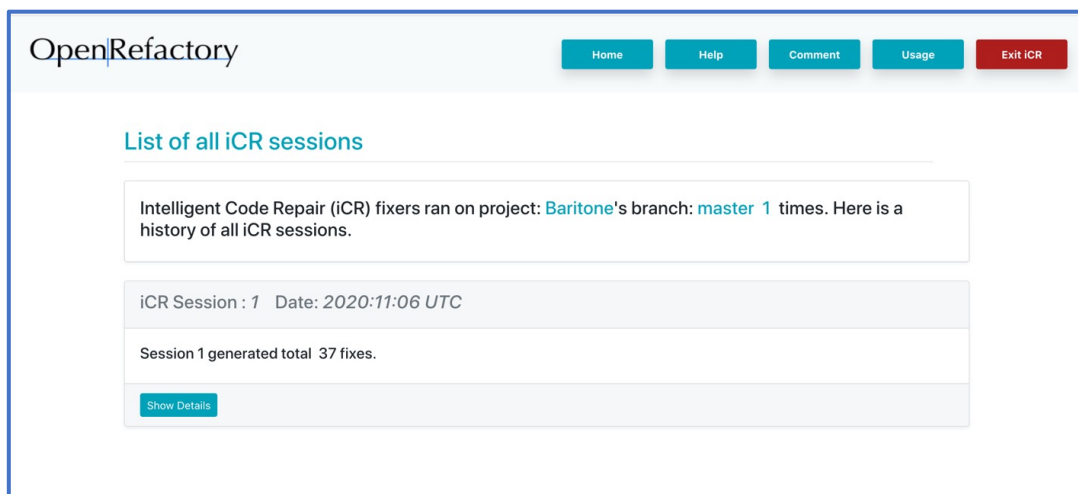
## 6.0 Reviewing Your Results

Once you have completed an analysis of one of your project branches, you can use the Navigator to begin reviewing the results. Using our earlier example project, following completion of the analysis on the master branch, the Navigator now shows the **Review** button as being available.



Clicking on the **Review** button will open a browser tab with a new Reviewer screen. Notice that the top banner from the Navigator screen is also available in the Reviewer, with one exception. The **Settings** button is gone and replaced by the **Home** screen. This allows you to return to a Navigator from this same tab. This is convenient if you have closed the Navigator tab following the initiation of a Reviewer session.

The initial screen displays a summary of all previous analysis sessions (if any). You may have run the ICR Engine more than once. It is helpful to repeat the analysis as you make changes to your code base. Subsequent runs may reveal new issues that were introduced with the changes in the code base. The sessions will be listed with the most recent at the top of the list and will have the highest Session number.



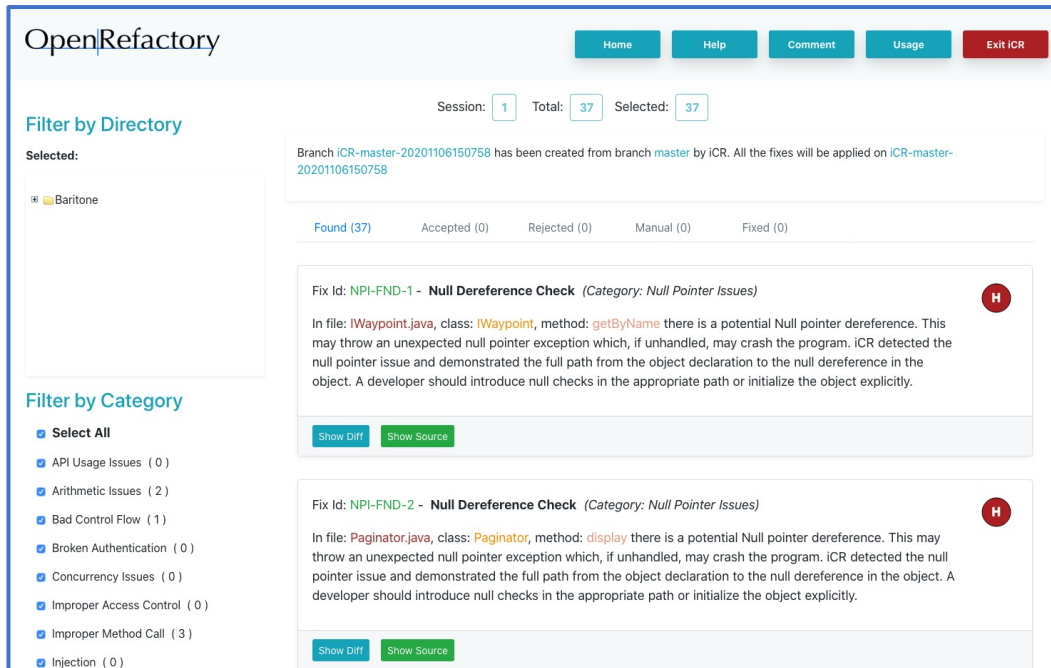
To view the results of any previous analysis, click on its **Show Details** button.

While you can select the results of any past session, only the most recent will permit the user to make changes. Results from older sessions may only be viewed.

In the example above, we will be reviewing the initial set of results that we just produced so will click on the **Show Details** button at the bottom of the “iCR Session: 1” box.

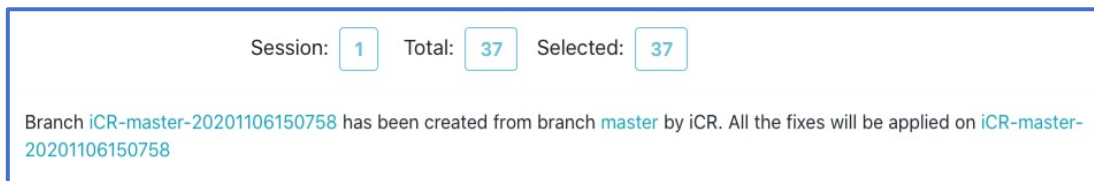
## 6.1 Reviewer summary and filters

The Reviewer results are displayed using a combination viewing panes with filters.



The window is divided into 3 panes. At the top left is the *Filter by Directory* pane. Below that is the *Filter by Category* pane. Finally, to the right of both of the filter panes is the *Fixes* pane.

The top portion of the *Fixes* pane displays a quick summary of the results. The Session number, the total number of fixes produced by the analysis is shown along with information about which fixes have been selected for display. At launch, all fixes are displayed by default.



The branch name that was the subject of this analysis is also displayed. More importantly, there is an additional branch name displayed. When you accept and then apply fixes, the Reviewer will create Git commits and apply them to a new, temporary branch in your repository. This allows *iCR for Java* to automatically update your source code in a fashion that allows you to prepare and review pull-requests before merging them into your actual project branch. In this example, the temporary branch is named: `iCR-master-20201106150758`.

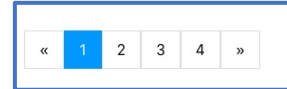
The tabs summarize the states of the various fixes. When the Reviewer is first launched following a fresh analysis, all the fixes generated are accounted for in the *Found* tab. It shows the total number of fixes (37, in this example). The other states of a fix are:



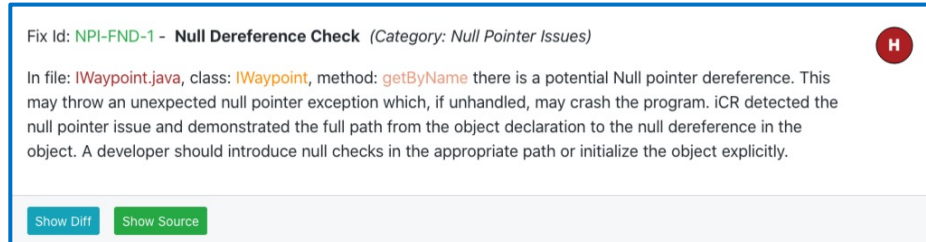
- *Accepted* – This fix has been approved for future application to the code base
- *Rejected* – This fix has been rejected
- *Manual* – There were conflicts in accepting all of the fixes so some manual intervention will be required
- *Fixed* – These are fixes that have been accepted and applied to the code base. Their state can no longer be changed

Note that each of the above tabs will show the total number of fixes in that state. If there are no fixes in that state, the tab will be inactive. How the state of a fix is modified will be described in a later section.

Up to 10 fixes are presented at any one time. The bottom of the *Fixes* pane shows the number of pages of fixes available for review and allows navigation across the pages. Also, the summary bar at the top of the page will not scroll off the top of the pane keeping the summary and the various state tabs available all of the time.



Each fix is identified with a unique Fix ID to help to distinguish each fix as it moves through the system. In this example below, the fix ID is **NPI-FND-I**. There is a title for the fix: **Null Dereference Check**, and the category within which this fix belongs: *Null Pointer Issues*.



There is also a description of the fix which includes the file name where the fix was produced: *IWaypoint.java*, the particular class: *IWaypoint* and the method: *getByName*. This information makes it easier for you to find the specific place in the code where the fix is being applied.

In the top right corner of the box is an icon that presents OpenRefactory's view of the risk associated with the bug being corrected. There are three levels of risk being assessed:

- High
- Medium and
- Low

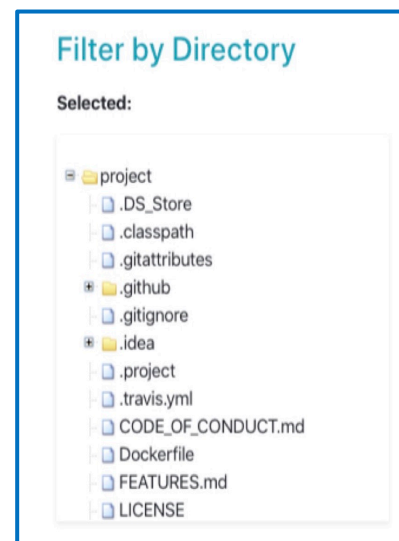
Higher risk bugs represent flaws that represent greater potential vulnerabilities if not corrected.

## 6.2 Filter by Directory pane

While this example “only” shows 37 fixes, larger projects may uncover many more fixes to be reviewed and eventually applied. As such it is helpful to be able to narrow the set of fixes to be reviewed.

One way to limit the fixes to be displayed is by selecting a subset of the files to be viewed. The *Filter by Directory* enables that. To navigate the directory structure and locate subdirectories of files, simply click on the “+” next to each folder to display the next level down. In our example, here is the view of the expanded “project” directory. Clicking on a folder will only display fixes from within that folder and its sub-folders. Clicking on a single file will limit the display to fixes that only apply to that file.

The pane is scrollable so that you can see to any depth of directory that you wish.



## 6.3 Filter by Category pane

Another way to filter the set of reviewable fixes is by constraining the various classes of fixers that are to be reviewed using *Filter by Category*. When the Reviewer is first launched, all of the categories are selected by default. This is indicated by showing that the “*Select All*” box is checked, and each individual category box is checked.

## Filter by Category

- ☒ **Select All**
- ☒ API Usage Issues ( 0 )
- ☒ Arithmetic Issues ( 2 )
- ☒ Bad Control Flow ( 1 )
- ☒ Broken Authentication ( 0 )
- ☒ Concurrency Issues ( 0 )
- ☒ Improper Access Control ( 0 )
- ☒ Improper Method Call ( 3 )
- ☒ Injection ( 0 )
- ☒ Null Pointer Issues ( 11 )
- ☒ Object Visibility ( 17 )

When all categories are selected and the entire project directory is selected, the summary will show all fixes that are available for review. In this example, that is 37.

Category filters are combined with the directory filter to limit the fixes summaries to only those fixes within that directory subtree AND the selected categories.

You may want to ONLY review fixes in a single category. In this case, you may click on the *Select All* option. Doing that deselects all of the categories. Then, you can click on only the one (or multiple) categories of particular interest. Clicking *Select All* will reset the category filters and all fixes will be displayed again.

If there is a directory subtree selected, only fixes in that category within the selected subdirectory or file will be shown.

In the example provided here, we have selected only those fixes in the *Object Visibility* category. Because of that, the summary at the top of the *Fixes* pane is updated to reflect that now, only 17 fixes are selected for review. Note that the *Found* tab also reflects this.

The screenshot shows the OpenRefactory web interface. On the left, the 'Filter by Category' sidebar is visible, with 'Object Visibility (17)' selected. The main pane shows the 'Found' tab with a summary of 17 fixes. Below the summary, two fix details are shown: 'Fix Id: OV-LFA-1 - Encapsulation Problem' and 'Fix Id: OV-LFA-2 - Encapsulation Problem', both categorized under 'Object Visibility'. Each fix detail includes a description of the problem and buttons for 'Show Diff' and 'Show Source'.

## 6.4 Handling Results

### 6.4.1 Reviewing a fix

Once you have filtered for the set of fixes for review, you may begin processing them. That typically begins with clicking on the *Found* tab to see what fixes need to be reviewed. In our example, we will be looking at a set of fixes within the *Object Visibility* category. There were 17 fixes identified.

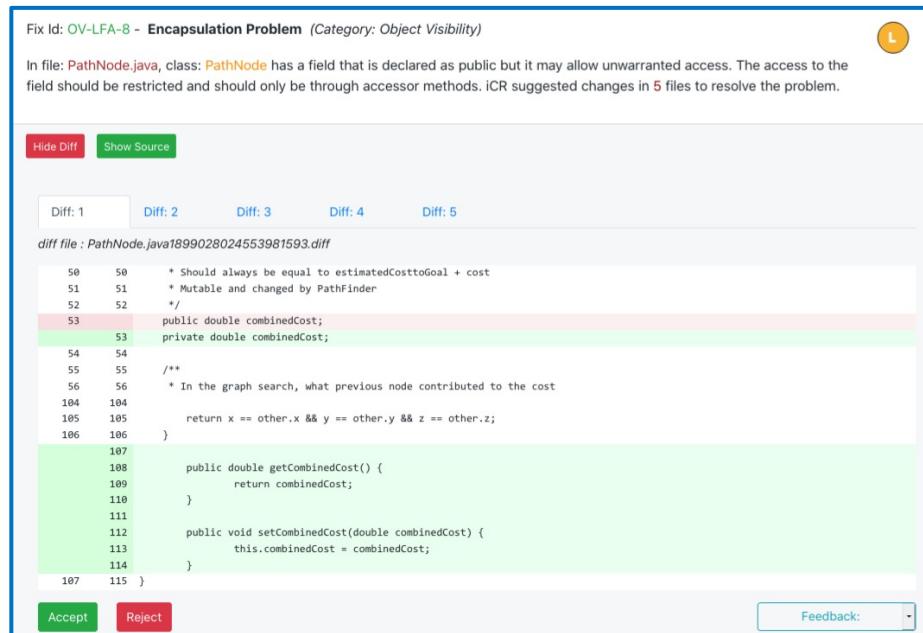
To show how to process a fix, we will look at Fix OV-LFA-8. In this example, it has detected an encapsulation problem where a variable that should be declared private to the class was declared as public.

The screenshot shows the details of a specific fix: 'Fix Id: OV-LFA-8 - Encapsulation Problem' (Category: Object Visibility). The description states: 'In file: PathNode.java, class: PathNode has a field that is declared as public but it may allow unwarranted access. The access to the field should be restricted and should only be through accessor methods. ICR suggested changes in 5 files to resolve the problem.' Below the description are buttons for 'Show Diff' and 'Show Source'.



To correct this Encapsulation Problem, the variable is made private and a pair of accessor methods to set and get the value is created. Any other files that reference the variable are updated to use the accessor methods instead of modifying the variable directly. As a result, the summary of the fix shows that there are offered changes to a total of 5 files.

To see the diffs for all of the 5 files, click on the [Show Diff](#) button. Doing that reveals an expanded display.



Since there were 5 files affected, there are 5 [Diff](#) tabs shown where each tab corresponds to the changes suggested for each affected file. In this example, [Diff: 1](#) is selected and displayed. This is the diff for the file containing the improperly declared public variable.

The lines that were changed are identified by the red highlighted statements. In this example, that is Line 53. The text below that shows the corrected code with green highlights. The class variable `double combinedCost` was declared `public` but should be `private`. The iCR generated code corrects the issue by making the variable `private` shown as the replacement for line 53. In addition, the accessor methods `getCombinedCost` and `setCombinedCost` are added to allow controlled access to the now private variable as shown in added lines 107 through 114.

If you want to browse the original source file associated with this fix, you can click on the [Show Source](#) tab. A scrollable window will appear below the diff window with tabs for each of the files that have a diff for this fix. You can click on any tab to browse the source for any of the affected files. In this case [Source of Diff: 1](#).

You can scroll through the original source file independently of the diff window.

Fix ID: OV-LFA-8 - Encapsulation Problem (Category: Object Visibility)

In file: PathNode.java, class: PathNode has a field that is declared as public but it may allow unwarranted access. The access to the field should be restricted and should only be through accessor methods. ICR suggested changes in 5 files to resolve the problem.

Hide Diff Hide Source

Diff: 1 Diff: 2 Diff: 3 Diff: 4 Diff: 5

diff file : PathNode.java1899028024553981593.diff

```

50 50  * Should always be equal to estimatedCostToGoal + cost
51 51  * Mutable and changed by Pathfinder
52 52  */
53 53  public double combinedCost;
54 54  private double combinedCost;
55 55  /**
56 56  * In the graph search, what previous node contributed to the cost
104 104
105 105  return x == other.x && y == other.y && z == other.z;
106 106  }
107 107
108 108  public double getCombinedCost() {
109 109  return combinedCost;
110 110  }
111 111  public void setCombinedCost(double combinedCost) {
112 112  this.combinedCost = combinedCost;
113 113  }
114 114  }
107 115  }

```

Accept Reject Feedback: ▾

Source of Diff: 1 Source of Diff: 2 Source of Diff: 3 Source of Diff: 4 Source of Diff: 5

Source file : Baritone/src/main/java/baritone/pathing/calc/PathNode.java

```

47 public double cost;
48
49 /**
50  * Should always be equal to estimatedCostToGoal + cost
51  * Mutable and changed by Pathfinder
52  */
53 public double combinedCost;
54
55 /**
56  * In the graph search, what previous node contributed to the cost

```

Once you are satisfied with reviewing a particular correction, you can select other [Diff](#): tabs to review all the suggested changes for this fix.

To view other fixes, scroll through the list of fixes or select new filters.

### 6.4.2 Accepting a fix

Continuing with the example of Fix OV-LFA-8, there are 2 buttons at the bottom left of the diff window. They are labeled **Accept** and **Reject**. These options allow you to make a decision on whether or not the changes are desired.

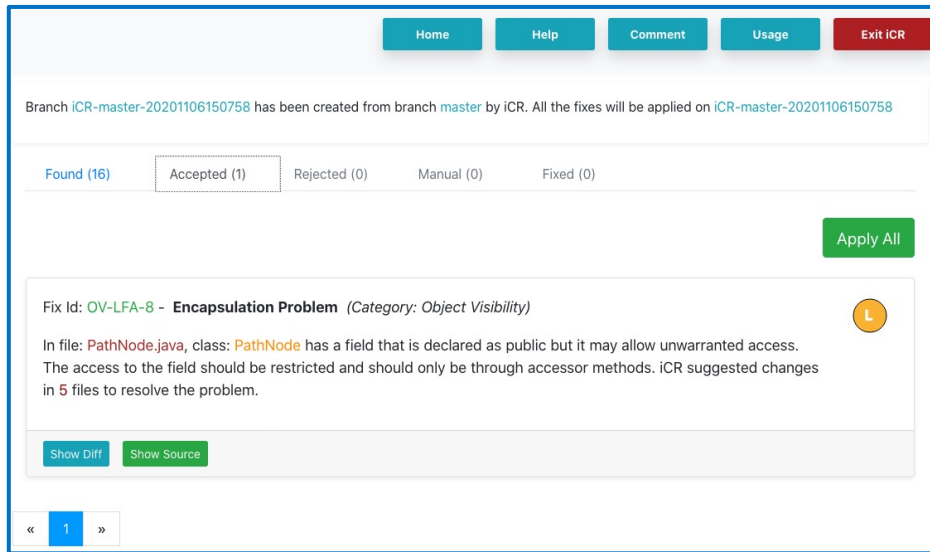
By clicking the **Accept** button, the fix (OV-LFA-8) is placed in the *Accepted* state. All of the changes are connected and changing some without the others would result in invalid code. It does not really matter which particular diff is used to accept or reject the changes. In the above example, **Accept** is chosen for [Diff: 1](#). Once chosen, the Diff window closes, and the fix disappears from the list of *Found* fixes.

Note that the summary tab now shows one fix in the *Accepted* state. Its tab is now highlighted in Blue because it is no longer empty.

Found (16)	Accepted (1)	Rejected (0)	Manual (0)	Fixed (0)
------------	--------------	--------------	------------	-----------



Clicking on the *Accepted* tab brings up the list of accepted fixes. In this example, there is the fix we just accepted, fix OV-LFA-8, with changes to 5 files.

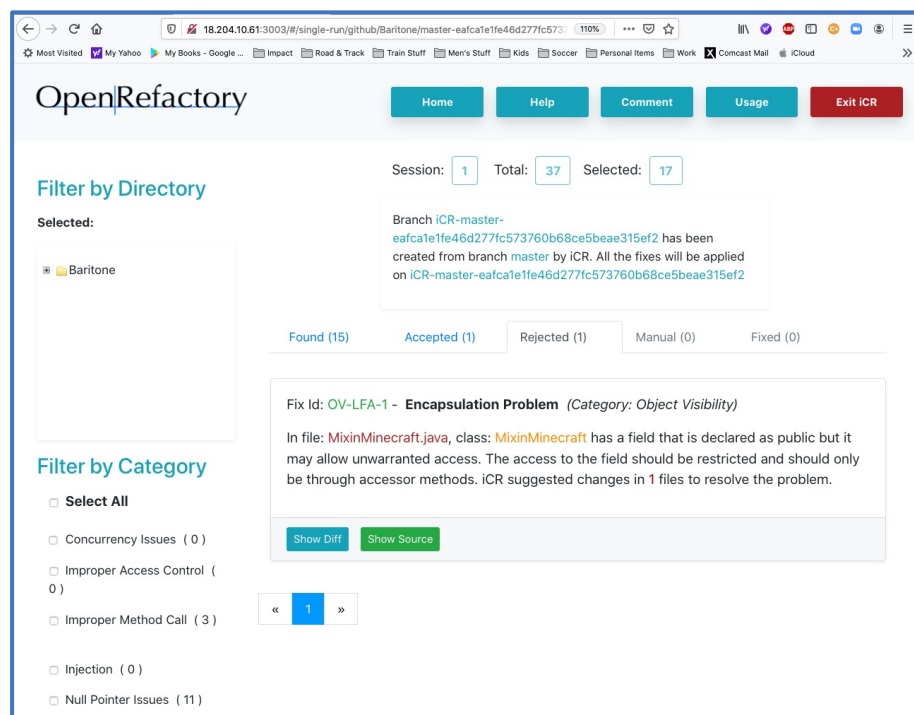


### 6.4.3 Rejecting a fix

It may be that there is a reason for not accepting a fix. If so, you may choose to click on the **Reject** button. This behaves in exactly the same way as accepting a fix. All of the diffs associated with this fix are kept together and the fix moves to the *Rejected* state. For an example of this, we will reject fix OV-LFA-1. After clicking the **Reject** button, the fix is moved to the *Rejected* state and that is reflected in the summary tab. As before, the *Rejected* tab becomes highlighted as it is now active with one rejected fix in that state.



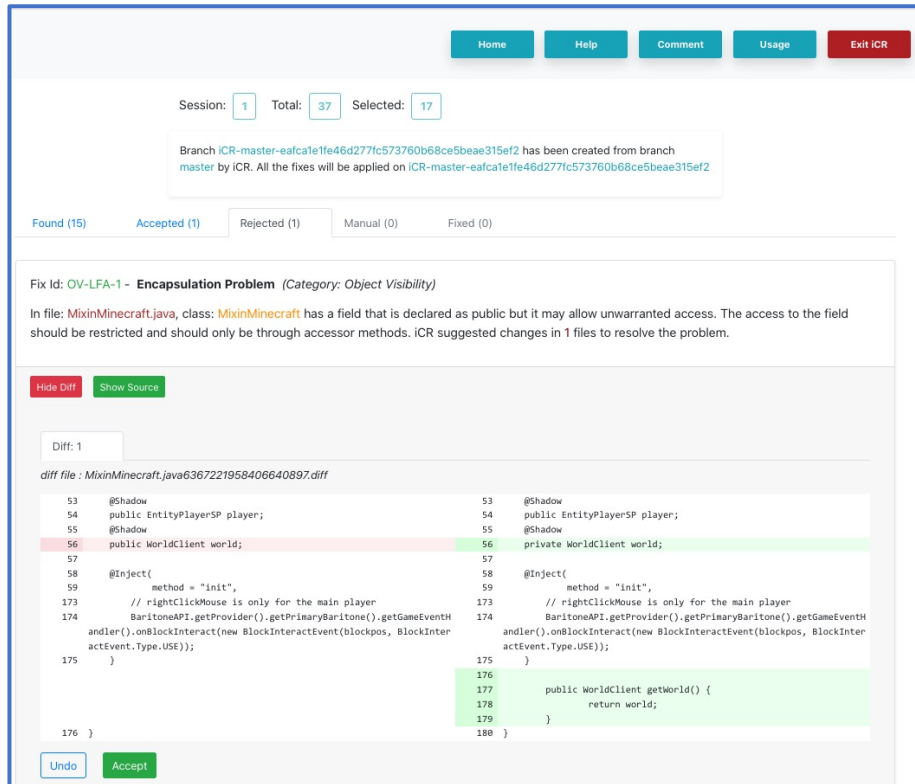
Clicking on it reveals the rejected fix:



### 6.4.4 Undoing a fix

Using the above example of OV-LFA-1, it may be realized that the fix is, indeed, needed, and that you want to change its status. This is easy to do by clicking on the one of the diffs to review the changes.

Clicking on the **Show Diff** button, as before, will display the original code and the rejected changes. But you will notice that the buttons at the bottom of the window are different from the *Found* fixes with a new button at the bottom.



A new **Undo** button is now available. If it is chosen, then the fix moves back to the *Found* state where it can be left for further review later.

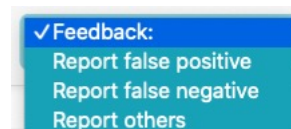
Since this example is one of a rejected fix, then the other option, to accept it instead, is also offered. So, you can click on the **Accept** button, and the fix will be moved from *Rejected* to *Accepted*.

A similar process works for *Accepted* fixes. Should the user decide to reject it instead, the **Reject** button is available. Also, as in the example above, the **Undo** button is also there as so the fix may be moved back to the *Found* state for later review.

A fix can be moved from any one of *Accepted*, *Rejected* and *Found* states by clicking the appropriate button while displaying a diff.

### 6.4.5 Providing feedback

When looking at diffs that are in the *Found* state, you will note that there is another option shown on the bottom right of the diff window opposite the **Accept** and **Reject** buttons. This is a pull-down menu that offers your developers the opportunity to provide feedback to OpenRefactory engineers.



While iCR for Java has a comprehensive analysis engine, there are always ways to improve it. Should your engineers determine that there may be an error in the analysis, or some other issue that they would like to see improved, they can select one of the feedback options and write a brief email for our development team.

The feedback window gives your developers the options to include the text of the fix and source code snippets so that we can evaluate our analysis and our correction.

We are constantly finding ways to improve both our analysis and the quality of our fixes, so your feedback would be welcome.

Feedback ×

Report false positive

☒ Attach fixes

☐ Attach source code snippets

Submit Report

### 6.4.6 Applying the fixes

The Reviewer provides the ability for you to select, browse and identify fixes to be accepted or rejected. The main purpose of this process is be able to apply these fixes to the source code itself.

When reviewing fixes in the *Accepted* state, you may click on the [Show Diff](#) button to review the offered changes. The display is a bit different from the one shown earlier.

Since this is an *Accepted* fix, the options at the bottom of the window are different. The [Undo](#) button is there as before, but now the user has the option of changing their mind and rejecting the change. That will move it over to the *Rejected* state.

Found (16)
Accepted (1)
Rejected (0)
Manual (0)
Fixed (0)

Apply All

Fix Id: OV-LFA-8 - Encapsulation Problem (Category: Object Visibility)

In file: PathNode.java, class: PathNode has a field that is declared as public but it may allow unwarranted access. The access to the field should be restricted and should only be through accessor methods. iCR suggested changes in 5 files to resolve the problem.

Hide Diff
Show Source

Diff: 1
Diff: 2
Diff: 3
Diff: 4
Diff: 5

diff file: PathNode.java1899028024553981593.diff

```

50 50  * Should always be equal to estimatedCosttoGoal + cost
51 51  * Mutable and changed by Pathfinder
52 52  */
53 53  public double combinedCost;
54 54  private double combinedCost;
55 55  /**
56 56  * In the graph search, what previous node contributed to the cost
104 104
105 105  return x == other.x && y == other.y && z == other.z;
106 106  }
107
108  public double getCombinedCost() {
109      return combinedCost;
110  }
111
112  public void setCombinedCost(double combinedCost) {
113      this.combinedCost = combinedCost;
114  }
115  }

```

Undo
Reject
Apply Fix

And there is an additional option on the right side of the window that is only available for fixes in the *Accepted* state. The [Apply Fix](#) button offers you the ability to insert the corrected code into the project itself. Clicking on [Apply Fix](#) instructs the Reviewer to create `git` specific `commits` to the temporary branch.

Also, at the top of the page shown above, there is a new button that appears at the top right of that window. That is the [Apply All](#) button which becomes active when any fixes are moved to the *Accepted* state. Clicking on this will tell the Reviewer to apply all of the fixes which are in the *Accepted* state. This is a quick way of applying all the currently accepted fixes in one step.

Once fixes have been applied, they are moved into the *Fixed* state. Once in the *Fixed* state, the fixes cannot be undone other than having a developer manually edit the code. It is exactly the same as if the developer had modified the code directly and committed them manually.

### 6.4.7 Cases needing manual attention

The *iCR for Java* engine creates fixes independently of other fixes. As such, it is sometimes the case when the same area of code may be affected by overlapping fixes. Since some fixes may be accepted and others

rejected, there are cases where the Reviewer cannot make an unambiguous set of edits to the code to result in the correct output when **Apply Fix** or **Apply All** is clicked. In those cases where the changes could not be safely applied automatically, the Reviewer will move the fix into the *Manual* state.

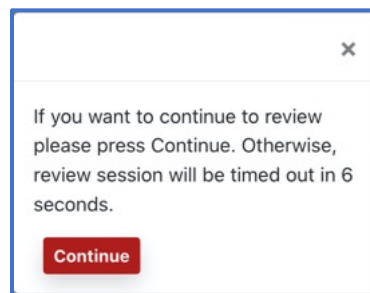
Once a fix is in the *Manual* state, it is treated the same as those in the *Fixed* state in that its state can no longer be changed. It would need to be edited manually to incorporate any desired fixes and the `commits` or other edits to the source code in the temporary branch would need to also be performed manually.

#### 6.4.8 Ending a Reviewer session

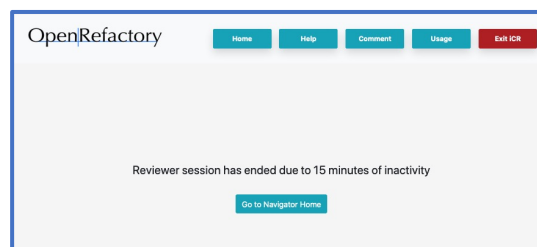
It may be the case, especially if you are executing *iCR for Java* for the first time, that there will be many offered fixes to be reviewed. You may want to distribute the task of reviewing the fixes to multiple members of your team. Or, you may want to review fixes in batches over time.

You can end a Reviewing session at any time by clicking on the **Home** button. This is recommended so that you can avoid unnecessary usage charges. This will redirect your tab back to the Navigator. This is handy if you had closed the Navigator tab from before. Or, you can simply close the Reviewer tab and return to your previous Navigator tab. In either case, your Reviewer session ends, and your usage accrual is stopped.

Because *iCR for Java* is a pay-as-you-go service, the Reviewer also monitors your activity. If you are idle for a period of 15 minutes, the reviewer pops up an alert asking if you want to continue the session. If so, simply click continue to proceed.



If you allow the timeout to expire, your Reviewer session will end, and a simple display will be shown to let you know that iCR for Java ended your session and your usage accrual has stopped.



Of course, you can always end your session by simply clicking the **Exit iCR** button.

You may return at any time to the Navigator by clicking on its tab. However, it is strongly recommended that you explicitly end each Reviewer session to limit your usage time accrual.

## 7.0 When You Are Complete

Once you have reviewed all of your results, you can exit the Navigator. To close a Reviewer session, simply close the tab or click the **Home** button. To leave the Navigator, you may close the Navigator tab or click **Exit iCR**.

If you have completed all the analyses and have reviewed all of your results, you can check them on your VCS and verify the commits are there. Any fixes that were applied will be committed to the temporary branch as identified in the Reviewer header banner.

Once satisfied that you are complete, you can go to your terminal console and stop the server using `icrforjava -c stop`. Note that stopping the server will leave all of your previous results available for later. You can restart the server using `icrforjava -c start`. Of course, it may be more convenient to always keep the server running so that other developers can analyze their projects

OpenRefactory appreciates receiving all feedback on its products that users are willing to provide. Please contact us at [info@openrefactory.com](mailto:info@openrefactory.com) if there are any questions or suggestions for improvements on the operation of *iCR for Java*.

## Appendix A – List of Supported Fixers

This appendix enumerates the currently supported set of Fixers for the *ICR for Java* Analysis Engine. OpenRefactory is constantly updating this list as new algorithms are developed for additional Fixers. Please contact OpenRefactory at [info@openrefactory.com](mailto:info@openrefactory.com) to stay current on available Fixers.

### API Usage Issues (7):

#### Add Controller Class Restrictions –

A Spring `@Controller` class that uses `@SessionAttributes` have to call `setComplete()` on the `SessionStatus` object from an `@RequestMapping` method. This is specific to the Spring framework.

#### Add Component Package Location –

A class with annotation `@ComponentScan` should include all component (Service, Repository, Controller, RestController) packages. Otherwise, the classes will not be available in the Spring application context. This is specific to the Spring framework.

#### Add Default Package Restrictions –

The default package should not contain a class with `@ComponentScan`, `@SpringBootApplication`, or `@ServletComponentScan` annotation. This is specific to the Spring framework.

#### Block Serialization with Append –

An object output stream that is opened in append mode should not be serialized because the data will be stored in the wrong format and a deserialization attempt will result in an exception being thrown.

#### Replace Confusing Scope Combination –

Classes with annotation `@Controller`, `@Service`, or `@Repository` are singleton classes and if they have `@Scope` annotation, that should be explicitly specified. This is specific to the Spring framework.

#### Replace EnableAutoConfiguration with Import –

A class with annotation `@EnableAutoConfiguration`, should be replaced by `@Import`, as `@EnableAutoConfiguration` may include unnecessary beans which slows down an application. This is specific to the Spring framework.

#### Remove Method Call –

Remove unnecessary method calls that have been deprecated.

### Arithmetic Issues (1):

#### Fix Zero Division –

Removes zero division opportunities in code. Fixes CWE 369, CERT Secure coding standard NUM02-J.

### Bad Control Flow (4):

**Add Missing Breaks –**

Unexpected control flow because of missing `break` statements in `switch`. Fixes CWE 484.

**Fix Equality Check –**

Confusing object equality (`equals` method) with reference equality (`==` operator) and vice versa lead to inappropriate control flow, thus leading to hard-to-debug root causes. Perform appropriate equality checks according to the context. Fixes CWE 595, CWE 597, CERT Secure coding standard EXP03-J, CERT Secure coding standard EXP50-J.

**Move Default Statement –**

Switch statements should handle default case after everything else is handled.

**Remove Unused Semicolon –**

Remove unexpected control flow scenarios because of bad use of delimiters.

### Broken Authentication (2):

**Fix Hard-coded Key –**

Cryptographic keys or other credentials should not be kept hard-coded in the source code. An attacker can extract the strings or byte arrays from an application source code or binary. Fixes CWE 798, OWASP A2-Broken Authentication, CERT Secure coding standard MSC03-J.

**Fix Hard-coded Password –**

User passwords should not be kept hard-coded in the source code. An attacker can extract the strings or byte arrays from an application source code or binary. Fixes CWE 259, OWASP A2-Broken Authentication, CERT Secure coding standard MSC03-J.

### Concurrency Issues (4):

**Avoid Value-Based Class Locks –**

Synchronization should avoid value-based classes as locks. Fixes CERT Secure coding standard LCK01-J.

**Avoid String and Boxed Primitive Locks –**

Synchronization should avoid `Strings` and boxed primitives that can be reused. Fixes CERT Secure coding standard LCK01-J.

**Remove Servlet Mutable Fields –**

Make the instance fields of the servlet classes to be `static` or `final`, or remove them. The servlet container creates one instance of each servlet for each HTTP request and the threads will share the instance fields, leading to concurrency issues. Fixes CERT Secure coding standard MSC11-J.

**Synchronize with Proper Class –**

Synchronization should avoid using `getClass()` methods. Fixes CERT Secure coding standard LCK02-J.

### Improper Access Control (2):

**Get Proper Permission –**

Get Proper Permission from the super `ClassLoader` if any class extends the `URLClassLoader`. Fixes CERT Secure coding standard SEC07-J.



**Prevent Persistent Entity Short-circuiting –**

Persistent objects annotated with `@Entity` or `@Document` should not be used as arguments in methods annotated with `@RequestMapping` and similar other annotations. This is specific to the Spring framework. Fixes CWE 915, OWASP A5-Broken Access Control Issue.

**Improper Method Call (4):****Check Return Result –**

Method `return` values that return error codes should be checked against error codes before being used. Fixes CWE 252.

**Fix Finalize Method Implementation –**

`finalize` method should be avoided or if used, called properly with reference to `Object.finalize`. Fixes CWE 568, CERT Secure coding standard MET12-J.

**Call Super Method –**

Overriding methods should reference the method in the parent class.

**Prevent Incompatible Transactional Calls –**

Methods should not call same-class methods with incompatible `"@Transactional"` values.

**Injection (8):****Prevent SQL Injection –**

Constructing SQL queries with untrusted user provided data, e.g., URL parameters, enables attackers to inject code in place of data that changes the meaning of the SQL query. Identify potential SQL injection opportunities. Fixes CWE 20, CWE 85, CWE 943, OWASP A1-Injection Issue, CERT Secure coding standard IDS00-J.

**Prevent Cross-site Scripting –**

When endpoints reflect back tainted, user-provided data such as `POST` content, URL parameters, etc., it may allow attackers to inject code that will eventually be executed on the browser of a user. Identify potential SQL injection opportunities. Fixes CWE 79, CWE 80, CWE 81, CWE 82, CWE 83, CWE 84, CWE 85, CWE 86, CWE 87, OWASP A7-XSS.

**Prevent Path Manipulation –**

Constructing file system paths from untrusted user-provided data such as `POST` content, URL parameters, etc., enables attackers to inject specific path browsing symbols, such as `".."`, to manipulate the file path and to access files that they are not allowed to access otherwise. Identify potential path manipulation opportunities. Fixes CWE 22, CWE 23, CWE 36, CWE 99, CWE 641, OWASP A1-Injection, OWASP A5-Improper Access Control.

**Prevent OS Command Injection –**

Applications that execute operating system calls should not use untrusted user-provided data to create the command or command parameters. Identify potential OS command injection opportunities. Fixes CWE 77, CWE 78, CWE 88, OWASP A1-Injection.

**Prevent XPath Injection –**

Constructing `XPath` expressions using untrusted user-provided data such as `POST` content, URL parameters, etc., enables attackers to inject specially crafted values that change the way the expression is supposed to be interpreted under normal circumstances. Identify potential `XPath` injection opportunities. Fixes CWE 643, OWASP A1-Injection, CERT secure coding standard IDS53-J.

**Prevent LDAP Injection –**

Constructing `LDAP` names or search filters using untrusted user-provided data enables attackers to inject values that change the way the name or the filter is supposed to be interpreted under normal circumstances. Identify potential `LDAP` injection opportunities. Fixes CWE 90, OWASP A1-Injection, CERT secure coding standard IDS54-J.

**Prevent Regular Expression Denial of Service –**

Using external strings as regular expressions leads to potential denial of service attack since evaluating the regular expressions is CPU intensive. Identify potential regular expression injection opportunities. Fixes CWE 400, OWASP A1-Injection.

**Prevent SQL Injection in Prepared Statement –**

`PreparedStatement` is used to prevent SQL injection attacks. But, constructing SQL queries with string concatenation using untrusted user provided data, e.g., URL parameters, undermines the benefits of a `PreparedStatement`. Identify potential SQL injection opportunities. Fixes CWE 20, CWE 85, CWE 943, OWASP A1-Injection Issue, CERT Secure coding standard IDS00-J.

**Null Pointer Issues (2):****Bad Return Value –**

Methods with boxed type `return` values should not return `null`. Fixes CWE 476, CERT Secure coding standard EXP01-J.

**Fix Null Dereference –**

A pointer which has not been initialized is used as if it pointed to a valid memory area in the heap. A null pointer issue happens because the developer mistakenly did not allocate an object or has mistakenly assumed that the object is allocated when in fact it is null. Fixes CWE 476, CERT Secure coding standard EXP01-J.

**Object Visibility (2):****Add Qualifier for Static –**

Access inherited static fields using the parent class as the qualifier.

**Limit Field Access –**

Accessibility of fields in Java classes should be limited. Fixes CWE 582, CWE 607, CERT Secure Coding Standard OBJ01-J and OBJ13-J.

**Security Misconfiguration Issues (11):****Declare EJB Connectors Properly –**

Following EJB 3.0 conventions, application security interceptors must be listed in the `ejb-jar.xml` file, or they will not be treated as default interceptors. Fixes OWASP A6-Security Misconfiguration Issue.

**Remove Duplicate Validation Forms –**

The names of Struts validation forms should be unique. When there are duplicate validation form names, the Struts Validator arbitrarily chooses one of the forms to use for input validation and discards the other. This is specific to the Struts framework. Fixes CWE 102, OWASP A6-Security Misconfiguration Issue.

**Use Declared Filter –**

Every filter declared in web.xml file should be used in an element. Otherwise such filters are not invoked. Fixes OWASP A6-Security Misconfiguration Issue.

**Limit Scope of Maven Dependencies –**

System dependencies in Maven are sought at a specific path that matches a configuration and cannot be ported. If an artifact is deployed in an environment that is different from the original configuration, the build would fail.

**Track Messages During Restart –**

In Spring, `DefaultMessageListenerContainer` is implemented as a Java Message Service (JMS) polling component. While the Spring container is going through a restart/shut down, the message listener may discard messages and will therefore lose them. The message listener container should be declared such that the messages are not discarded. This is specific to the Spring framework.

**Allow Automatic Connection Recovery –**

Spring framework uses a factory object (`SingleConnectionFactory`) that returns the same connection for all connection requests. This should be declared with a setting to allow automatic recovery when the connection goes bad. This is specific to the Spring framework.

**Stop Debugging Web Remoting –**

Direct Web Remoting (DWR) is a Java and JavaScript library that enables RPC calls in an Ajaxian application. If the debug mode of DWR is turned on, it will allow users to access information exposed under the debugging servlet. This is specific to the DWR library.

**Remove Duplicate Servlet Definition –**

When a deployment descriptor contains the same name for multiple servlets, only the first one is deployed, and the others are ignored.

**Avoid GET Mix –**

Spring framework annotations should use `HTTP GET` method and not mix it with other methods. Fixes CWE 352, OWASP A6-Security Misconfiguration Issue.

**Use Proper Request Mapping –**

Ensures that Proper Request Mappings are used. Fixes CWE 352, OWASP A6-Security Misconfiguration Issue.

**Annotate Public Method –**

Spring framework annotations should be accompanying a public method.

**Sensitive Data Exposure (4):****Replace Random Generator –**

Weak random number generator should be replaced with a strong random number generator. Fixes CWE 330, CWE 332, CWE 336, CWE 337, OWASP A6-Security Misconfiguration Issue, Secure coding standard MSC63-J.

**Remove Weak Seed –**

A strong random number generator does not need a seed value to be set. Setting the seed with a constant or a predictable value will weaken the random generator itself. If a seed value is set explicitly, it should be removed. Fixes CWE 337, OWASP A6-Security Misconfiguration Issue, CERT secure coding standard MSC63-J.

**Use Proper Dependency Injection –**

Non-static members and constructors in classes with annotation `@Controller`, `@Service`, or `@Repository` should have proper annotations (any one of `@Autowired`, `@Value`, `@Resource`, or `@Inject`). This is specific to the Spring framework. Fixes OWASP A3-Sensitive Data Exposure Issue.

**Prevent XML eXternal Entity –**

XML Document Type Definition (DTD) should be disabled to prevent information disclosure via XXE attacks. Fixes CWE 611, CWE 827, OWASP A4-XML eXternal Entities.

**Weak Cryptography Issues (6):****Use Strong Password Encoder –**

Authentication manager should use a strong password encoder. This is specific to the Spring framework. Fixes CWE 327, CWE 328, OWASP A2-Broken Authentication Issue, OWASP A6-Security Misconfiguration Issue.

**Use Strong Hash Function –**

Hashing should be done using strong hashing algorithm such as SHA-256 or SHA-3. Fixes CWE 327, CWE 328, OWASP A6-Security Misconfiguration Issue.

**Use Secure Socket Protocol –**

SSL context objects should use a secure socket protocol such as TLS or DTLS. Fixes CWE 326, CWE 327, OWASP A3-Sensitive Data Exposure, OWASP A6-Security Misconfiguration Issue.

**Restrict Access to Broadcast Receiver –**

While registering broadcast receivers in Android, broadcast permission should be specified so that a receiver only receives broadcasts sent by components having proper permission. This is specific to Android applications. Fixes CWE 925, OWASP A1-Injection.

**Restrict Access to Broadcast Sender –**

While sending broadcast messages in Android, broadcast permission should be specified so that only receivers with proper permission can receive it. This is specific to Android applications. Fixes CWE 927, OWASP A3-Sensitive Data Exposure.

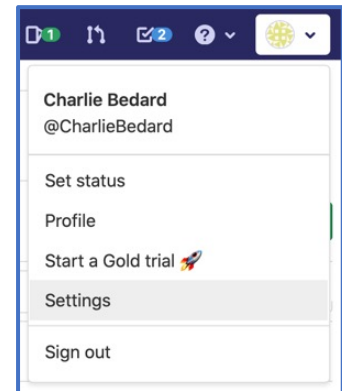
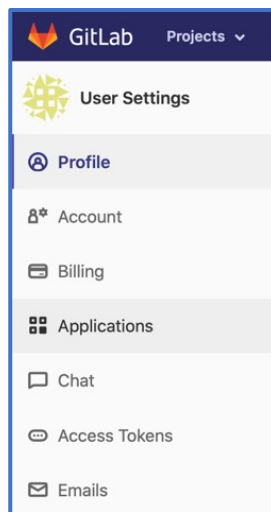
**Prevent Sharing of User Preferences –**

Android `getPreferences` and `getSharedPreferences` should use private mode when invoked, so that the preferences are not exposed globally. Fixes OWASP A3-Sensitive Data Exposure.

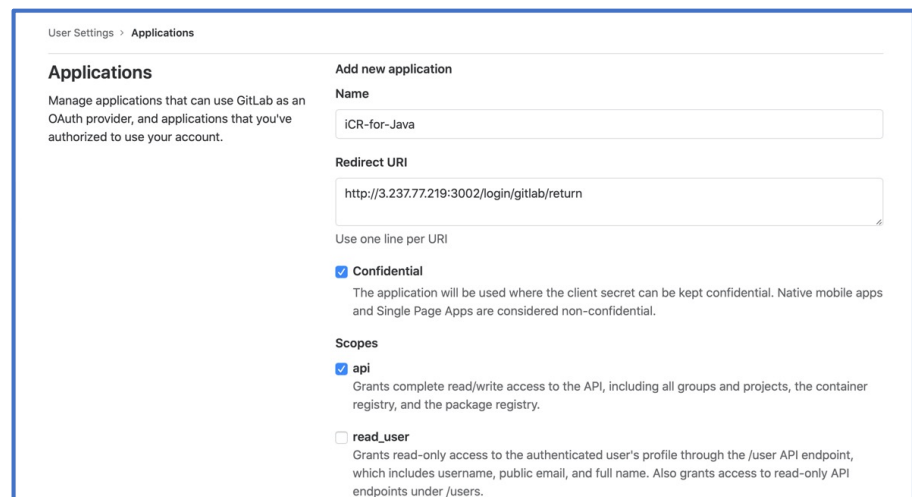
## Appendix B – GitLab OAuth Setup

Section 3.3 [Accessing Your Code](#), describes how to create the OAuth credentials needed to access GitHub. This appendix adds the additional details if you are planning on using GitLab to access your source code.

GitLab also uses the OAuth standard to allow you to tell GitLab that your *iCR for Java* server is allowed to redirect login credentials for GitLab to authenticate. To set this up, login into GitLab and go to your user menu at the top right of the GitLab menu bar. From there, select “Settings” from the pull-down menu.



The “Settings” menu offers a number of configurable options. Click on “Applications” to go to the Applications authorization page.



The “Applications” page is where you tell GitLab to allow your Server to allow logins redirected from the Server. For the application “Name” use whatever you like. “iCR-for-Java” has been used in this example. You also must enter the redirect URL to the server. GitLab’s OAuth uses that to verify the authorization handshake. Enter the URL as your server’s IP address with port 3002 and the callback text. Using the example IP address from Section 3.3, enter:

<http://3.237.77.219:3002/login/gitlab/return>

You need to select both the “Confidential” and the “api” options.

User Settings > Applications > iCR-for-Java

🔔 The application was created successfully. ✕

**Application: iCR-for-Java**

Application ID	88f866d210b32c556da6b7	📋
Secret	e23e6c83796af88a30b2bd	📋
Callback URL	http://3.237.77.219:3002/login/gitlab/return	
Confidential	Yes	
Scopes	• api (Access the authenticated user's API)	

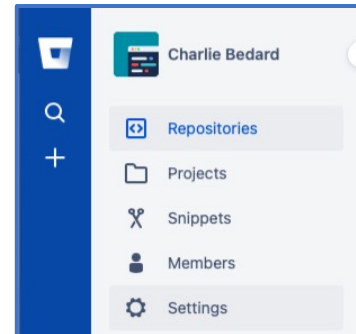
[Edit](#) [Destroy](#)

As was noted for GitHub, once you have completed this step, you will need to copy the Application ID and the Secret. From here, the process is the same as outlined for GitHub.

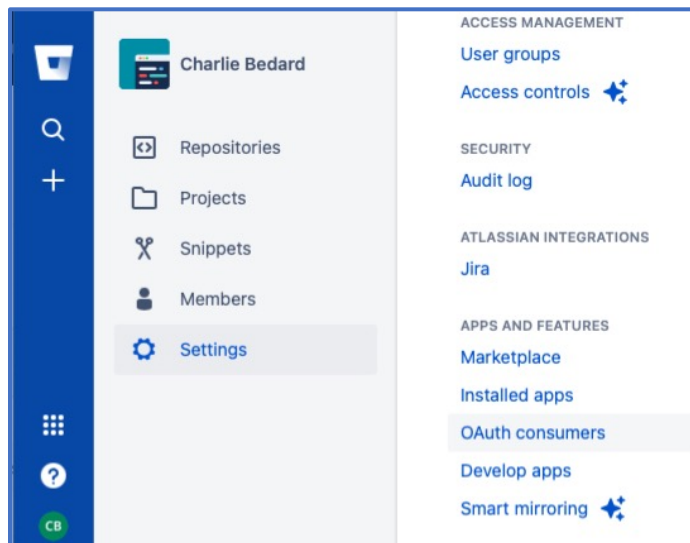
## Appendix C – BitBucket OAuth Setup

Section 3.3 [Accessing Your Code](#), describes how to create the OAuth credentials needed to access GitHub. This appendix adds the additional details if you are planning on using BitBucket to access your source code.

BitBucket also uses the OAuth standard to allow you to tell BitBucket that your *iCR for Java* server is allowed to redirect login credentials for BitBucket to authenticate. To set this up, login into BitBucket and go to your Repositories page.

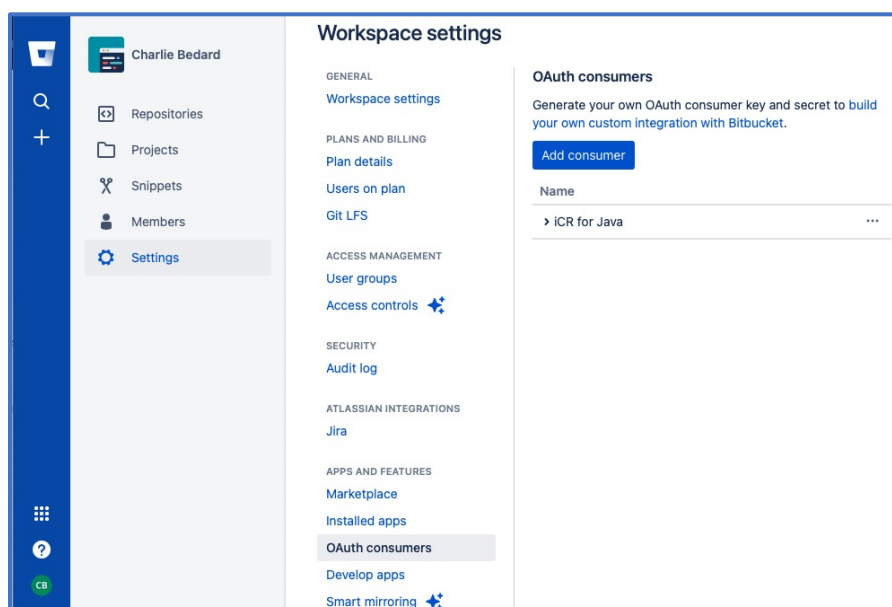


From there, select “Settings” from the pull-down menu →



The “Settings” menu offers a number of configurable options.

← Click on “OAuth consumers” under “APPS AND FEATURES” to go to the OAuth authorization page.



The “OAuth consumers” page is where you tell BitBucket to allow your Server to allow logins redirected from the Server.

Click on “**Add Consumer**” to add a new OAuth consumer for *iCR for Java*.



For the consumer “Name” use whatever you like. “iCR-for-Java” has been used in this example. You also must enter the redirect URL to the server. BitBucket’s OAuth uses that to verify the authorization handshake. Enter the URL as your server’s IP address with port 3002 and the callback text. Using the example IP address from Section 3.3, enter:

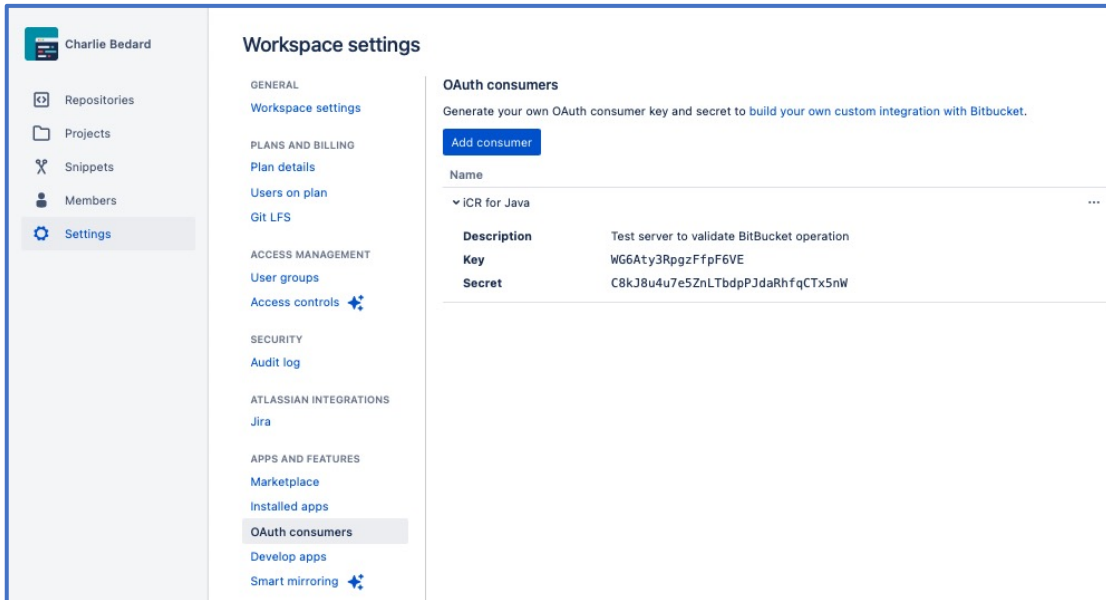
<http://3.237.77.219:3002/login/bitbucket/return>

The screenshot shows the BitBucket 'Workspace settings' page for a user named Charlie Bedard. The left sidebar contains navigation links: Repositories, Projects, Snippets, Members, and Settings (highlighted). The main content area is titled 'Workspace settings' and has a sub-header 'Add OAuth consumer'. The 'Details' section includes fields for Name (iCR for Java), Description (Test server to validate BitBucket operation), Callback URL (http://3.237.77.219:3002/login/bitbu), and optional fields for URL, Privacy policy URL, and End user license agreement URL. A checkbox 'This is a private consumer' is selected. The 'Permissions' section has two columns of checkboxes. In the first column, 'Account' has 'Read' checked, and 'Repositories' has 'Read', 'Write', 'Admin', and 'Delete' checked. In the second column, 'Pull requests' has 'Write' checked, and 'Issues' has 'Read' and 'Write' checked. At the bottom are 'Save' and 'Cancel' buttons.

Category	Item	Read	Write	Admin	Delete
Account	Email	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Read	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Workspace membership	Read	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Write	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Projects	Read	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Write	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Repositories	Read	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	Write	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	Admin	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	Delete	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Pull requests	Read	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Write	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Issues	Read	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Write	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Wikis	Read and write	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Snippets	Read	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Write	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Webhooks	Read and write	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Pipelines	Read	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Write	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Edit variables	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

You also need to select both the “Account: Read” and the “Pull requests: Write” options. The select “Save” to complete the consumer authorization.

As was noted for GitHub, once you have completed this step, you will need to copy the Key and the Secret. From here, the process is the same as outlined for GitHub.



The screenshot shows the BitBucket 'Workspace settings' page for user Charlie Bedard. The left sidebar contains navigation links: Repositories, Projects, Snippets, Members, and Settings (highlighted). The main content area is titled 'Workspace settings' and lists various configuration categories: GENERAL (Workspace settings), PLANS AND BILLING (Plan details, Users on plan, Git LFS), ACCESS MANAGEMENT (User groups, Access controls), SECURITY (Audit log), ATlassian INTEGRATIONS (Jira), and APPS AND FEATURES (Marketplace, Installed apps, OAuth consumers, Develop apps, Smart mirroring). The 'OAuth consumers' section is active, displaying instructions to generate an OAuth consumer key and secret for custom integration with BitBucket. An 'Add consumer' button is present. Below, a table lists existing consumers, with one entry expanded for 'ICR for Java'.

Name	Description	Key	Secret
ICR for Java	Test server to validate BitBucket operation	WG6Aty3RpgzFfpF6VE	C8kJ8u4u7e5ZnLTbdpPJdaRhqCTx5nW

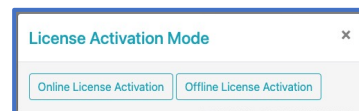
## Appendix D – Manual License Activation

Section 4.3 [Licensing Authorization](#), introduces the procedure of enabling the manual creation and activation of an *ICR for Java* license. This appendix details the individual steps needed to execute this procedure.

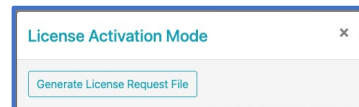
Manual license processing is required when the private platform is isolated from the Internet. This isolation prevents the automatic license activation handshake. These steps allow you to execute the handshake manually. These steps refer to the private platform as the “isolated” platform. To perform the procedure, a computer needs to access the Internet and the License Management system. That computer is referred to as the “proxy” system.

These are the required steps:

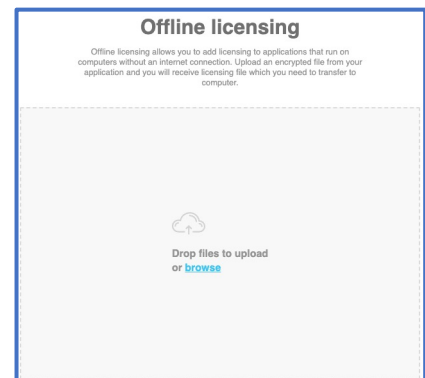
- It is assumed that the software has been installed on the target private platform using the steps outlined in Section 3.1;
- Using the `icrforjava` commands, configure the platform and start the server for the first time;
- Once started, use a browser to access the passphrase page and enter the default passphrase to enter the Navigator;
- From the “[Settings](#)” menu, select the “[License Activation Mode](#)” setting as shown in Section 4.4;
- Choose the “[Offline License Activation](#)” option to enable manual license activation.



- Activating the offline license activation option enables the mechanism for creating the unique hardware signature for this platform. You need this signature in order to use a proxy to activate the license with the License Manager service.
- Clicking on “[Generate License Request File](#)” produces a file with the hardware signature and related information needed to validate and activate your license. The name of this file is `ls_activation.req`. This file is saved in the directory named: `/etc/icr-data/offline/` where it can be retrieved.
- The signature must be moved/copied from the isolated private platform to a proxy system which has access to the Internet.



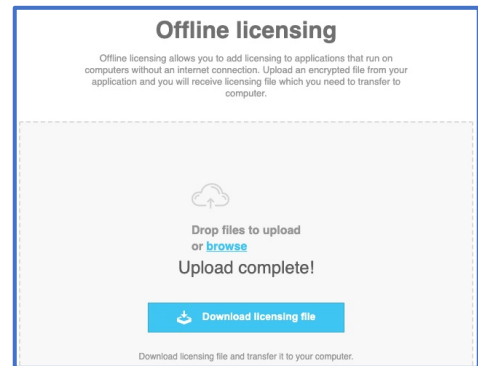
- Once on the proxy, use a browser on the proxy to access the License Management System (LMS) at this URL: <https://saas.licensespring.com/offline/>. Once there you will see the prompt to upload your signature file to the License Manager.
- Drag and drop or use the “[browse](#)” button to upload the file to the LMS.



- Once uploaded, you will see an acknowledgement that the file was uploaded. Once uploaded, the LMS will validate the request and create an authorization file enable the license for the isolated platform that created the request.
- You may now retrieve the authorization file from the LMS by clicking on the button labeled “[Download licensing file](#)”. This will download it to the proxy system wherever you choose to place downloaded files. The file is named:  
`ls_activation.lic`.
- Now you can transfer this license activation file back to the isolated private platform by moving/copying it to the system. It **MUST** placed in the directory named:  
`/etc/icr-data/activate/`.
- With the license activation file in the correct place, your isolated *ICR for Java* Private Platform software will now execute properly.

### NOTE:

In the event that your manual license activation failed for some reason. Or if you need to renew or reset your license, execute the `icrforjava -r` command. This will remove the current license activation file and allow you to repeat these steps with a fresh license key (if required).



## Appendix E – Example Summary Report

When an analysis completes, a prompt is displayed requesting that you send OpenRefactory a brief summary of your analysis. This is completely optional on your part and the data we send does not include any information about your source code. To show you what we send, here is an example summary of an analysis performed on the Open Source example used in the Guide, the Baritone project.

Summary: Project Name: Baritone

\*\*\*\*\*Summary of the Console\*\*\*\*\*

Start Time: 2021-02-16 13:34:52 PST

End Time: 2021-02-16 13:41:54 PST

Total Runtime: 00:07:02

Ended?: true

Total Java Files: 314

Total Exceptions/Errors generated: 1

Total Unique Exceptions/Errors generated: 1

Parsing Phase:

Parsing phase runtime: 00:00:13

Number of exceptions/errors thrown during file parsing: 0

Number of unique exceptions/errors thrown during file parsing: 0

CG Phase:

CG phase stage-1 runtime: 00:00:04

CG phase stage-2 runtime: 00:01:09

Number of exceptions/errors thrown during creating callgraph stage-1: 0

Number of unique exceptions/errors thrown during creating callgraph stage-1: 0

Number of exceptions/errors thrown during creating callgraph stage-2: 0

Number of unique exceptions/errors thrown during creating callgraph stage-2: 0

Methods Hash size: 440 KB

Classes Hash size: 78 KB

Sub Classes size: 21 KB

Root Methods size: 9 KB

CG Method Index size: 80 KB

Class to Fields size: 83 KB

Annotation to Class size: 0 KB

Class To Bit Index Size: 29 KB

Inner Classes size: 2 KB

Interface size: 10 KB

Field Access Modifier & Init Method Call size: 184 KB

root methods size: 240

call graph size: 8019

execution time: 73.0s

Points To Phase:

PointsTo phase runtime: 00:03:36

Number of points to clusters: 1565

Total Sat calls: 0 / 0  
Average graph edges: 110420 / 1601 = 68.96939412866958  
Points-to-analysis time: 215.0s  
Number of exceptions/errors thrown during Points To: 1  
Number of unique exceptions/errors thrown during Points To: 1

Fixing Phase:

Fixing phase runtime: 00:02:00  
Number of exceptions/errors thrown during Fixing: 0  
Number of unique exceptions/errors thrown during Fixing: 0

-----All Exceptions-----  
-----

Parsing Phase Exceptions:

-----  
-----

CallGraph Phase Stage-1 Exceptions:

-----  
-----

CallGraph Phase Stage-2 Exceptions:

-----  
-----

PointsTo Phase Exceptions:

-----  
-----

EXC-ID: POINTSTO-PHASE-1

-----

Total no. of occurrences: 1  
Exception:

```
java.lang.NullPointerException
  at com.10.01.01.6.CoM2.const(fnb:1170)
  at com.10.01.01.6.COM5.visit(ktb:395)
  at org.eclipse.jdt.core.dom.SimpleName.accept0(SimpleName.java:198)
  at org.eclipse.jdt.core.dom.ASTNode.accept(ASTNode.java:2971)
  at com.10.01.01.6.coM5.const(oib:90)
  at com.10.01.01.01.01.NUL.const(pob:156)
  at com.10.01.01.01.01.nuL.const(jkb:100)
```

```

    at com.10.01.01.6.Con.const(nlb:1271)
    at com.10.01.01.01.01.nUl.const(vjb:22)
    at com.10.01.01.6.coM1.const(qpb:197)
    at com.10.01.01.6.coM1.const(qpb:199)
    at com.01.10.10.10.con.const(gfc:176)
    at com.01.10.10.nul.const(lec:50)
    at com.01.10.10.nul.call(lec:204)
    at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
    at
java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:515)
    at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
    at
java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1
128)
    at
java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:
628)
    at java.base/java.lang.Thread.run(Thread.java:834)

```

Context Lines (These lines come after the exception):

2021-02-16 13:36:35 PST [SERVER]: Processing method cluster with call chain:  
 Single CG Node:  
 PathingControlManager.java::revalidateGoal::6507::547::1241::7366::PathingControlMana  
 ger::class::47

Fixing Phase Exceptions:

```

-----
-----

```

#### \*\*\*\*\*Summary of the Results\*\*\*\*\*

```

Total Fixes: 28
LFA: 17
  LFA-1: 17
FEC: 1
  FEC-6: 1
FND: 7
  FND-1: 7
RRG: 1
  RRG-1: 1
USA: 2
  USA-3: 2

```