Harry Duong
Thuc Nguyen
Deon Zhao

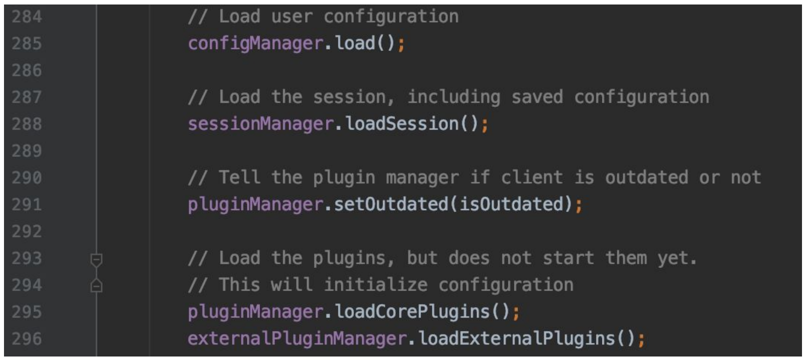
Changes:

1) The first feature on the PluginManager was changed to dive deeper into the code and to elaborate what made it essential.
2) We changed the second feature from IdleNotifierPlugin to SessionManager.
3) Code snippets, call graphs, and sequence diagrams now have explanations

Feature 1: PluginManager

The first feature that we found to be essential to our system, Runelite, is the PluginManager. Since Runelite depends on providing plugins for users to use during gameplay, we believed that this manager is essential because it specifies how plugins will be loaded, added, removed, and when these events will happen. These plugins range from metronomes that play a metronome that is set to a specific tick delay to loot trackers that track the content and value of items dropped. The PluginManager must exist throughout the lifetime of the system to handle the event in which users want to use different kinds of plugins; the manager adds a plugin to an active list or removes a plugin that is not enabled from the list. Most importantly, this feature has to be structured in a way to be scalable because Runelite allows developers to create new features as plugins and these plugins can be integrated within the system. This also means that we expect plugins to be dynamically loaded and added to the client during runtime and for there to be external plugins that somehow behave the same way.

Our starting point begins under the runelite-client[client] package because this system is a client for Runescape. We found that the launcher class, or the entry point to the system resided in the Runelite class at runelite-client/src/main/java/net/runelite/client/Runelite.java. Within the main method, we see that an injector is created and takes in a ClientLoader instance as well as an instance of the Runelite class as arguments. We steered our focus to the start() method of the Runelite class because this is the location in the source code where various managers including PluginManager are used and loaded to allow the system to run properly. More specifically, what we found the most relevant methods can be seen in figure 1 on lines 295 and 296, where the plugin manager loads the core plugins and a different manager, the ExternalPluginManager, loads the external plugins. In figure 2, the call to startPlugins() on line 347 by the plugin manager also seemed significant, so we dove into the PluginManager class.

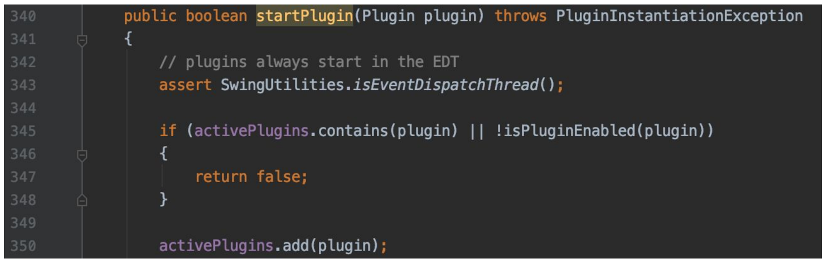


```
284        // Load user configuration
285        configManager.load();
286
287        // Load the session, including saved configuration
288        sessionManager.loadSession();
289
290        // Tell the plugin manager if client is outdated or not
291        pluginManager.setOutdated(isOutdated);
292
293        // Load the plugins, but does not start them yet.
294        // This will initialize configuration
295        pluginManager.loadCorePlugins();
296        externalPluginManager.loadExternalPlugins();
```

Figure 1: The code within start() from the Runelite class shows how the PluginManager and ExternalPluginManager make method calls to load core and external plugins on lines 295 and 296, respectively.

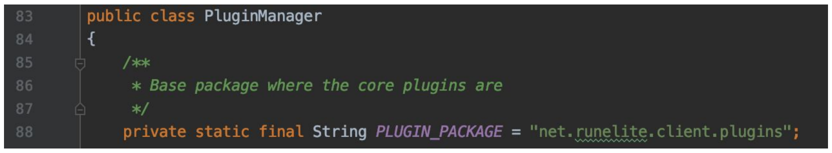
Figure 2: The start() method shows that the PluginManager starts the plugins immediately after registering managers to the event bus on line 347.

Diving into the PluginManager class at runelite-client/src/main/java/net/runelite/client /plugins/PluginManager.java, we can see that the core plugins are loaded into the system within the loadCorePlugins() and loadPlugins() method. Together, these two classes collect the core plugins and create a graph, adding those plugins to a list. There is also a startPlugins() method implemented in this class which was called earlier from the Runelite class. In figure 3, each plugin is iterated over and passed to the startPlugin() method on line 340, which then adds the plugin to a list of active plugins if it is not already there or if it is enabled. As seen in figure 4, there is a file path to the plugins package which is referenced by the PluginManager to access all of the core plugins.



```
340     public boolean startPlugin(Plugin plugin) throws PluginInstantiationException
341     {
342         // plugins always start in the EDT
343         assert SwingUtilities.isEventDispatchThread();
344 
345         if (activePlugins.contains(plugin) || !isPluginEnabled(plugin))
346         {
347             return false;
348         }
349 
350         activePlugins.add(plugin);
```

Figure 3: The startPlugin method is called (line 340) and adds plugin to an active list if it is not already enabled or not already active.



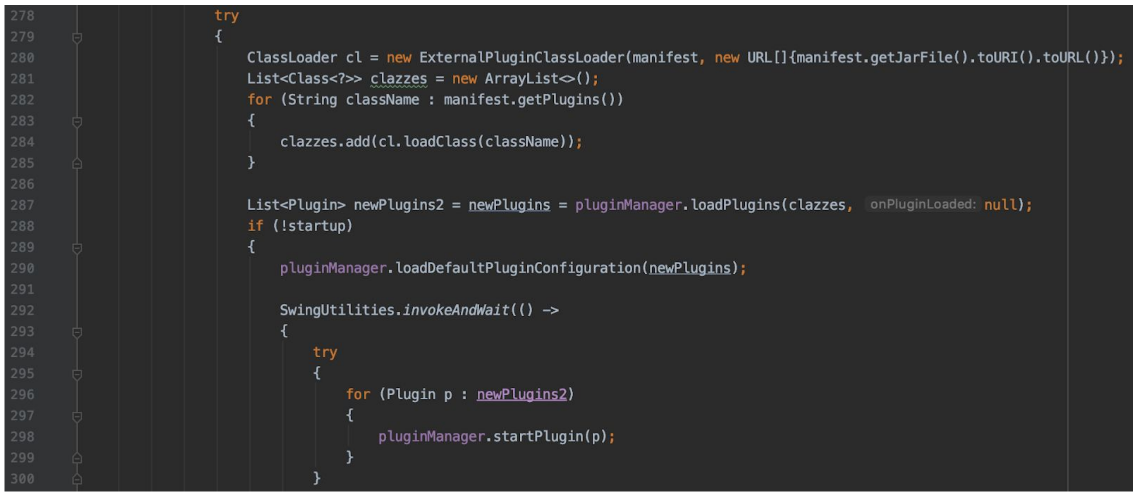
```
83  public class PluginManager
84  {
85      /**
86       * Base package where the core plugins are
87       */
88      private static final String PLUGIN_PACKAGE = "net.runelite.client.plugins";
```

Figure 4: The file path where all the plugins reside is specified as a constant (line 88).

By examining the Runelite and PluginManager classes and these methods so far, we were able to learn more intricately about the components of our system and how it is structured in the system to make it all work. In other words, the entire functionality of Runelite revolves around the use of plugins that may either be core to the system or external to it and thus imported. There has to be a manager to load the plugins and store references to them, adding them to an active list if the plugins are enabled or removing them if they are not. The PluginManager is the "essence" that we found in our system. It is the object that organizes plugins as child components and uses a list as the structure to keep references to them in order to manage them. If this plugin manager didn't exist, then none of the plugins would load or start and even if the plugins could start individually, they wouldn't be managed efficiently and can result in slowing down or even breaking the system. This also means that if someone wanted to make a change to this feature, it would be within the PluginManager class and specifically in the corresponding methods to add, stop, or load plugins. As a side note, we explored where the startPlugin() and stopPlugin() methods were called within the system but they were not as crucial. Plugins could be stopped if they had to be refreshed (in the refreshPlugins() method of both the PluginManager and ExternalPluginManager classes) or if they were toggled off in PluginListItem, which used a PluginListPanel object that had a reference to PluginManager. However, after examining these classes that relate to the toggling of plugins on the user interface, we understood that we didn't need to dive any deeper because it was a very straightforward call to start- and stopPlugins.

ExternalPluginManager (hypothesized as relevant)

We hypothesized that the ExternalPluginManager was relevant to the PluginManager at runelite-client/src/main/java/net/runelite/client/externalplugins/ExternalPluginManager.java. We understood that its purpose was to load non-core plugins so we explored the class. This class has a handle on a PluginManager and uses it to load plugins in loadExternalPlugins(). However, the core functionality of this manager resided in refreshPlugins(). This method first maps each plugin to an ExternalPluginManifest object into a map data structure before grabbing the list of String of the already-installed external plugins in getInstalledExternalPlugins(). Both of these code blocks are executed to prepare for any necessary downloading of external plugins. The code block in which the PluginManager is actually used begins in the try block on line 278 as seen in figure 5. It uses a Set of ExternalPluginManifest objects (which specifies the external plugins that need to be added) in order to get back the corresponding plugin and finally load the external plugins and start them using the PluginManager.



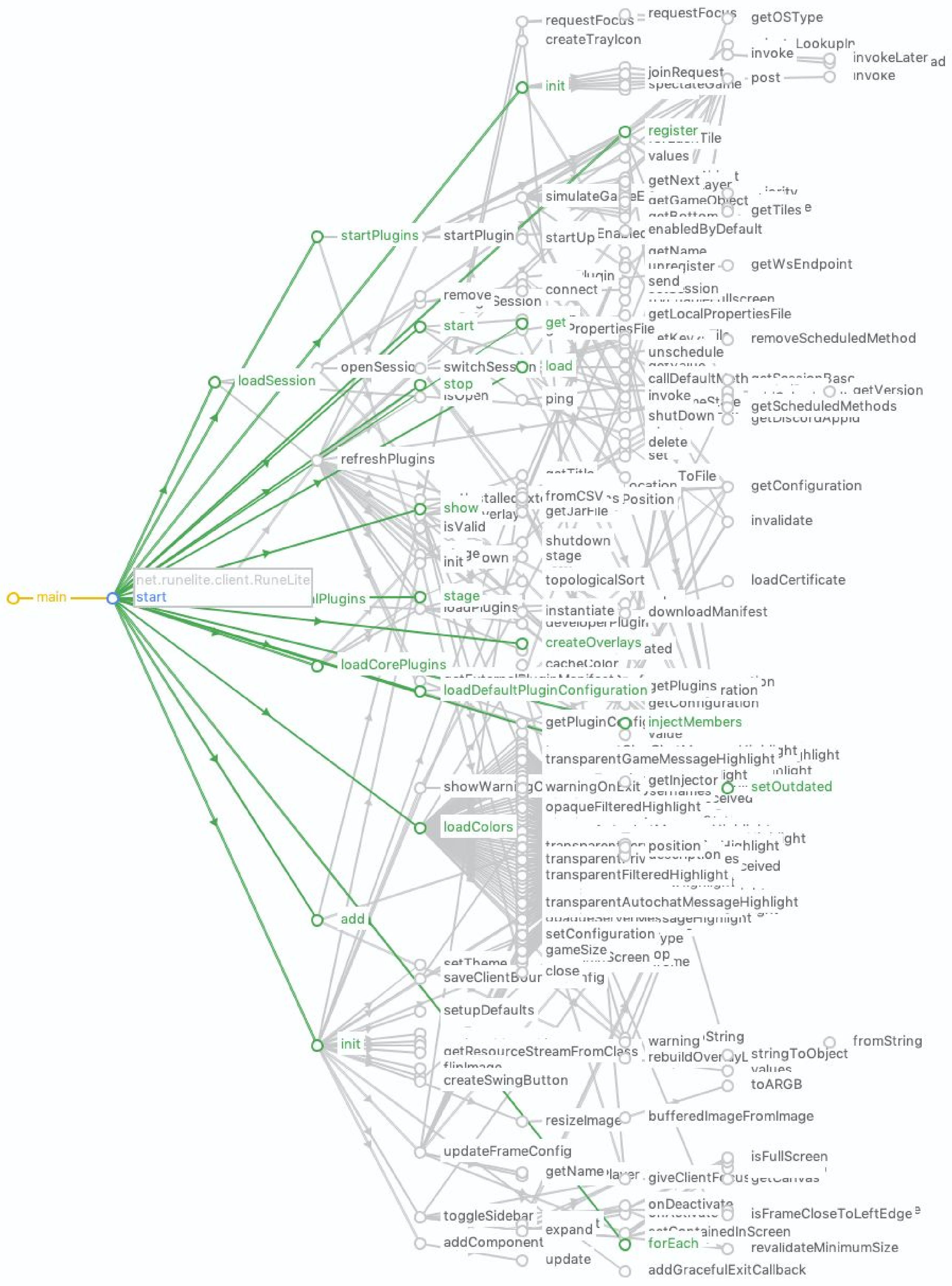
```
278 try
279 {
280     ClassLoader cl = new ExternalPluginClassLoader(manifest, new URL[]{manifest.getJarFile().toURI().toURL()});
281     List<Class<?>> clazzes = new ArrayList<>();
282     for (String className : manifest.getPlugins())
283     {
284         clazzes.add(cl.loadClass(className));
285     }
286
287     List<Plugin> newPlugins2 = newPlugins = pluginManager.loadPlugins(clazzes,  onPluginLoaded: null);
288     if (!startup)
289     {
290         pluginManager.loadDefaultPluginConfiguration(newPlugins);
291
292         SwingUtilities.invokeAndWait(() ->
293         {
294             try
295             {
296                 for (Plugin p : newPlugins2)
297                 {
298                     pluginManager.startPlugin(p);
299                 }
300             }
```

Figure 5: The try block of refreshPlugins() uses each ExternalPluginManifest object within the list of the objects to add (not shown) in order to extract the plugins associated with that manifest object (line 282). Each plugin is added to a list to pass as an argument to PluginManager's loadPlugins() method (line 287), and then started by the PluginManager (line 298).

This method is very large but it is related to PluginManager because it uses an instance of the manager to handle loading, adding, and removing plugins that are not part of the core packages. However, we learned from this systematic approach of comprehending the code that the ExternalPluginManager isn't as essential as we thought it would be. The plugins that are loaded aren't the core plugins, and in actuality, players who use the Runelite client when playing Runescape find more value in the core plugins than in external ones. We found this ExternalPluginManager to be more of an extra step to allow players to use plugins on top of the core ones. The class was only related to the PluginManager because it instantiated an object of that type, and used the functionalities of PluginManager to manage plugins after downloading them.

PluginManager Call Graph

The call graph below was loaded and focused on the start() function within the Runelite class because this method was the entry point for the entire client. The output of the graph which showed upstream and downstream methods verified that loadCorePlugins() and startPlugin() was called thereafter. Clicking on loadCorePlugins() also verified that loadPlugins() was called. The call graph was helpful in that it helped us visualize what methods and ultimately which classes were related and directly interacted with the start() method. If someone were to make a change to the PluginManager, clicking the node would reveal the current file path that the method resides in along with the downstream methods that could potentially be impacted by the change. The graph would help this person navigate and traverse deep into each function called to handle the change.



main
net.runelite.client.RuneLite
start
loadSession
startPlugins
startPlugin
init
start
stop
show
stage
loadCorePlugins
loadDefaultPluginConfiguration
createOverlays
loadColors
add
init
register
get
load
injectMembers
setOutdated
forEach
openSession
refreshPlugins
remove
switchSession
isOpen
isValid
showWarningOnExit
setTheme
saveClientBounds
setupDefaults
getResourceStreamFromClass
createSwingButton
updateFrameConfig
toggleSidebar
addComponent
requestFocus
createTrayIcon
simulateGameEvents
startUp
connect
ping
fromCSV
getJarFile
shutdown
topologicalSort
instantiate
cacheColor
getPluginConfig
transparentGameMessageHighlight
warningOnExit
opaqueFilteredHighlight
transparentFilteredHighlight
transparentAutochatMessageHighlight
setConfiguration
gameSize
close
resizeImage
getName
expand
update
joinRequest
spectateGame
values
getNext
getGameObject
enabledByDefault
getName
unregister
send
getLocalPropertiesFile
unschedule
callDefaultMethod
invoke
shutDown
delete
getPlugins
getConfiguration
value
getInjector
position
downloadManifest
warning
rebuildOverlay
bufferedImageFromImage
giveClientFocus
onDeactivate
addGracefulExitCallback
getOSType
invoke
post
invokeLater
getTiles
getWsEndpoint
removeScheduledMethod
getVersion
getScheduledMethods
getConfiguration
invalidate
loadCertificate
fromString
stringToObject
toARGB
isFullScreen
isFrameCloseToLeftEdge
revalidateMinimumSize

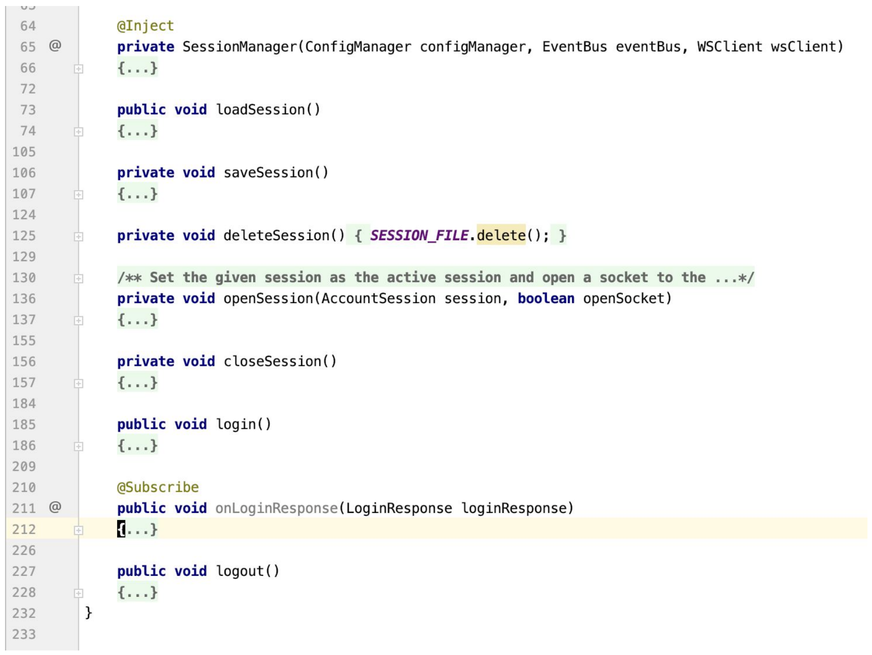
PluginManager Sequence Diagram

The sequence diagram was generated on the start() function of the Runelite class because we wanted to see when the PluginManager gets called and how it interacts with the Runelite and ExternalPluginManager classes. As seen below, the Runelite lifeline sends a message to the PluginManager lifeline to load the core plugins. The PluginManager and ExternalPluginManagers' lifelines interact with one another because ExternalPluginManager holds an instance of the PluginManager to load, add, and remove downloaded plugins to or from an active list.

<-- Insert Sequence Diagram generated on start() of Runelite class -->

Feature 2: SessionManager

The second feature that we found to be essential to our system was the SessionManager. We explored what other methods were called within the start() method of the Runelite class and the session manager was one such object. The SessionManager manages a user's current session so that the user can connect or reconnect to the server using a communication session via a web socket. With a very intuitive interface it has a few key methods like loadSession(), saveSession(), deleteSession(), closeSession(), login(), and logout() as seen in figure 6. To change the behaviour of a session's lifecycle or how a session is saved, simply modify any of these methods.

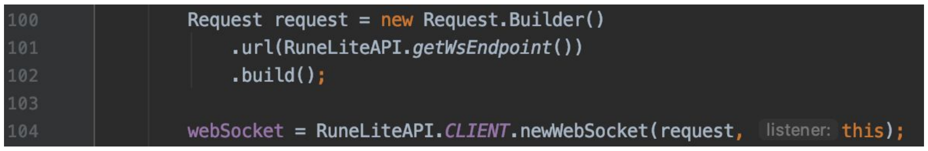


```
    @Inject
    private SessionManager(ConfigManager configManager, EventBus eventBus, WSClient wsClient)
    {...}

    public void loadSession()
    {...}

    private void saveSession()
    {...}

    private void deleteSession() { SESSION_FILE.delete(); }

    /** Set the given session as the active session and open a socket to the ...*/
    private void openSession(AccountSession session, boolean openSocket)
    {...}

    private void closeSession()
    {...}

    public void login()
    {...}

    @Subscribe
    public void onLoginResponse(LoginResponse loginResponse)
    {...}

    public void logout()
    {...}
}
```

Figure 6: Key methods of SessionManager that manages a session object of type AccountSession.

After a user logs in using the client, the login() method is called which tries to use the current session id if one is opened, or generate a new id for a new session. Only when a response is returned from onLoginResponse() does a session object of type AccountSession is obtained in order to open the session again and subsequently save the session. Saving the session is done by writing a json object to a file with the path given from the variable SESSION_FILE. When a user logs out from a click on the user interface, logout() is called and the session is closed in closeSession() and then the session file is deleted in deleteSession().

By examining the code flow within the SessionManager, we learned which parts of the class were important and interesting and which ones were not. Saving a json object to a file and deleting the file entirely are important but not interesting, so we felt that it wasn't the essence we were looking for. Opening and closing the session was more significant, because the SessionManager accesses the wsClient object and the WSClient class contains a WebSocket variable to handle the actual connection to the server. More specifically, we focused on the changeSession() and connect() methods of the WSClient class. The changeSession() method takes a session id as a parameter and checks if it is the same as the current session. It then closes a websocket if one exists, and then delegates to the connect() method. We found the essence of the SessionManager to be the statement that instantiated a new web socket with a built request. As seen on figure 7, this occurs on line 104 and the request is built on line 100.



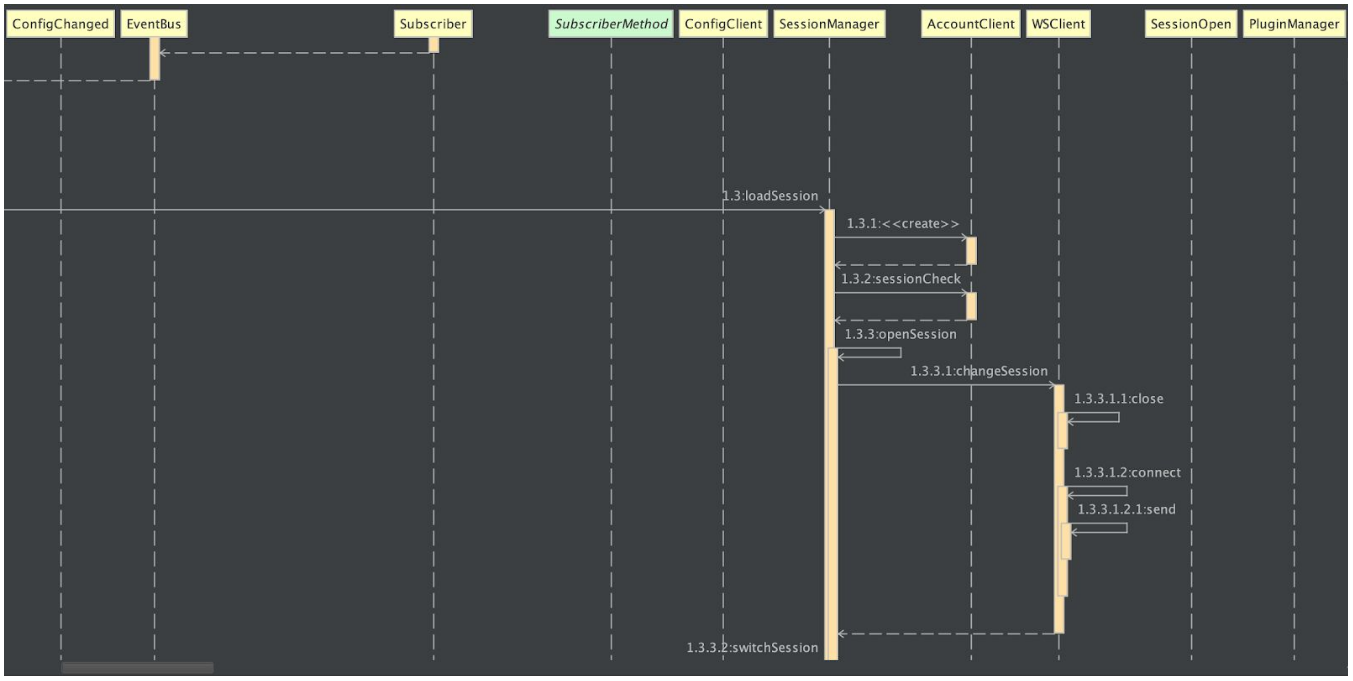
```
100         Request request = new Request.Builder()
101             .url(RuneLiteAPI.getWsEndpoint())
102             .build();
103
104         webSocket = RuneLiteAPI.CLIENT.newWebSocket(request,  listener: this);
```

Figure 7: The Request object is prepared (line 100) and built with the destination URL. A new web socket is instantiated (line 104).

So even though the essential feature involves the entirety of the SessionManager because it is the class that manages the session objects, the essence was found in the WSClient class and specifically when the web socket is created in order to connect to the server. It is true that the SessionManager class contains methods for file handling to read the session object from or write the session to a file, but the most important piece of code are the ones that utilized the web socket to handle connection and communication with the server. The SessionManager pieced these parts together to create a working client by interacting with an instance of the web socket. Without both the SessionManager and its use of the web socket, the system would not function property because the user wouldn't be able to connect to the server. The user must be able to login and connect to Runescape in order for the client to run, for plugins to work, and for users to use the client interface. Other parts of the SessionManager such as the EventBus or ConfigManager class are also essential to get the SessionManager to work, but we determined that these classes didn't truly envelop the essence of this feature.

SessionManager Sequence Diagram

We opted to use a sequence diagram to see when the SessionManager begins its interaction with the Runelite class and when and how it interacts with the WSClient class. As seen in the diagram below, the Runelite class interacts with the SessionManager class after a message call to loadSession() and subsequently, the SessionManager interacts with the WSClient lifeline via the call to changeSession(). This diagram also solidified our understanding of the class because the WSClient delegates the handling of connecting the client to the server to the web socket, evident in the self-loop to connect and send.


ConfigChanged
EventBus
Subscriber
SubscriberMethod
ConfigClient
SessionManager
AccountClient
WSClient
SessionOpen
PluginManager
1.3:loadSession
1.3.1:<<create>>
1.3.2:sessionCheck
1.3.3:openSession
1.3.3.1:changeSession
1.3.3.1.1:close
1.3.3.1.2:connect
1.3.3.1.2.1:send
1.3.3.2:switchSession