SEC_R0

BUILD YOUR OWN

PLUGINS

Applicable to:

Burp Suite Community Edition

Burp Suite Professional

A short guide jotted around developing burp suite plugin in Java and IntelliJ.
8 Chapters from Hello-World to A plugin with tab UI.

IntelliJ is used as an IDE,
just to simplify jar build process, you are free to use any Java IDE of your choice.

Chapter 1 : What is Burp Extender ?

This chapter talks about:

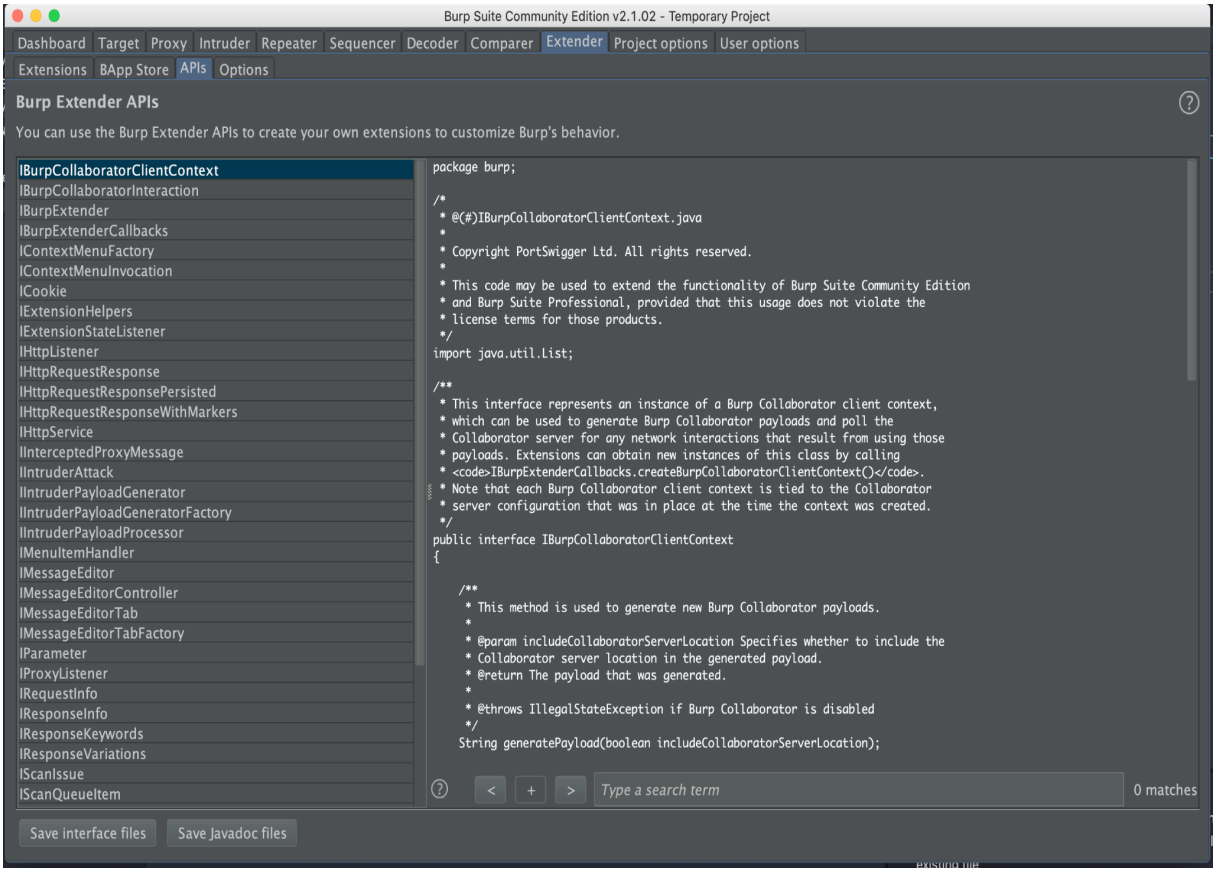
1. What is Burp Extender?
2. What are Burp API Extender interfaces?
3. Setup Dev Environment using IntelliJ
4. Using command line interface for building jars

Burp Extender

Burp Extender provides necessary functionality required for creation of Burp Suite extensions. The Extender tab exposes all APIs required for development of custom extensions in the form of Java Interfaces. These interfaces provide bindings for non Java environments as well, for Python through Jython and for Ruby through JRuby. The library support for Jython and JRuby is not so rich and is not usually recommended to pick these for plugin development.

This course also picks up plugin development through Java Interfaces.

You can see the list of all the Interfaces under Burp -> Extender -> APIs.



In the image above if you see, **on the left hand side you see the list of all the interfaces available** for you to implement a custom functionality and if you select any of those interfaces, you will see the in detail explanation of what that interface does and how/what can be achieved after implementing those interface.

IBurpExtender interface class is the entry point of any Extender plugin you write. And the class implementing this interface should implement void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks) abstract function. This is what the Java Doc of IBurpExtender also talks about itself.

The implementing class should belong to the **burp package**. This package will be automatically created once you export API Interface classes through Burp.

```
/*
 * @(#)IBurpExtender.java
 *
 * Copyright PortSwigger Ltd. All rights reserved.
 *
 * This code may be used to extend the functionality of Burp Suite Community Edition
 * and Burp Suite Professional, provided that this usage does not violate the
 * license terms for those products.
 */
/**
 * All extensions must implement this interface.
 *
 * Implementations must be called BurpExtender, in the package burp, must be
 * declared public, and must provide a default (public, no-argument)
 * constructor.
 */
public interface IBurpExtender
{
    /**
     * This method is invoked when the extension is loaded. It registers an
     * instance of the
     * <code>IBurpExtenderCallbacks</code> interface, providing methods that may
     * be invoked by the extension to perform various actions.
     *
     * @param callbacks An
     * <code>IBurpExtenderCallbacks</code> object.
     */
    void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks);
}
```

We will talk about it in good detail later, in the coming chapters.

Burp Extender Interfaces

All Burp Extender API interfaces are prefixed with **I** e.g **IIntruderPayloadProcessor**. The suffix after **I** is for the use case of API Interface, like:

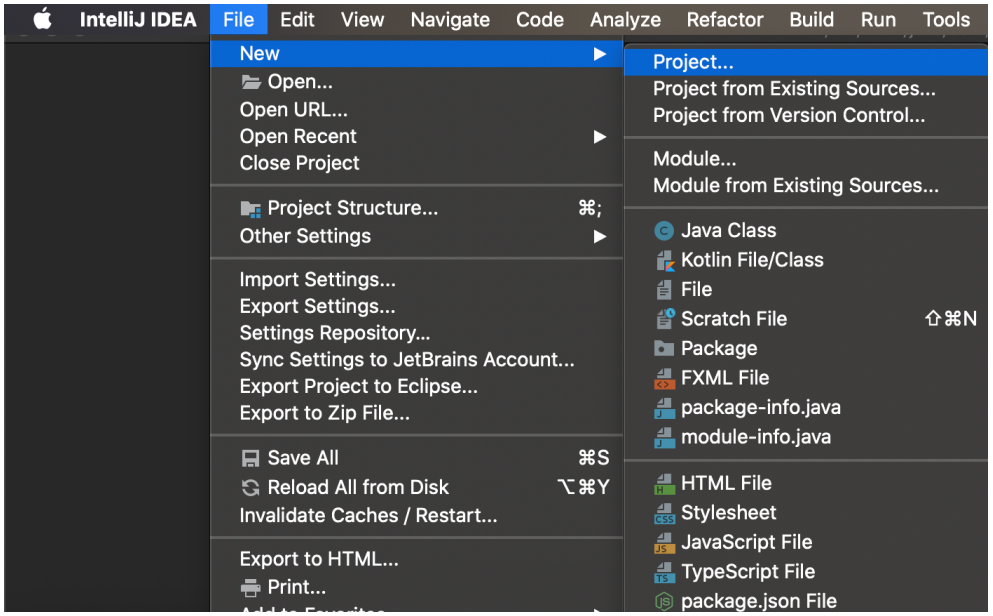
- **IBurpExtender**: All extensions must implement this interface.
- **IIntruderPayloadProcessor** - This interface is used for custom Intruder payload generators.
- **IParameter** - This interface is used to hold details about an HTTP request parameter, etc.

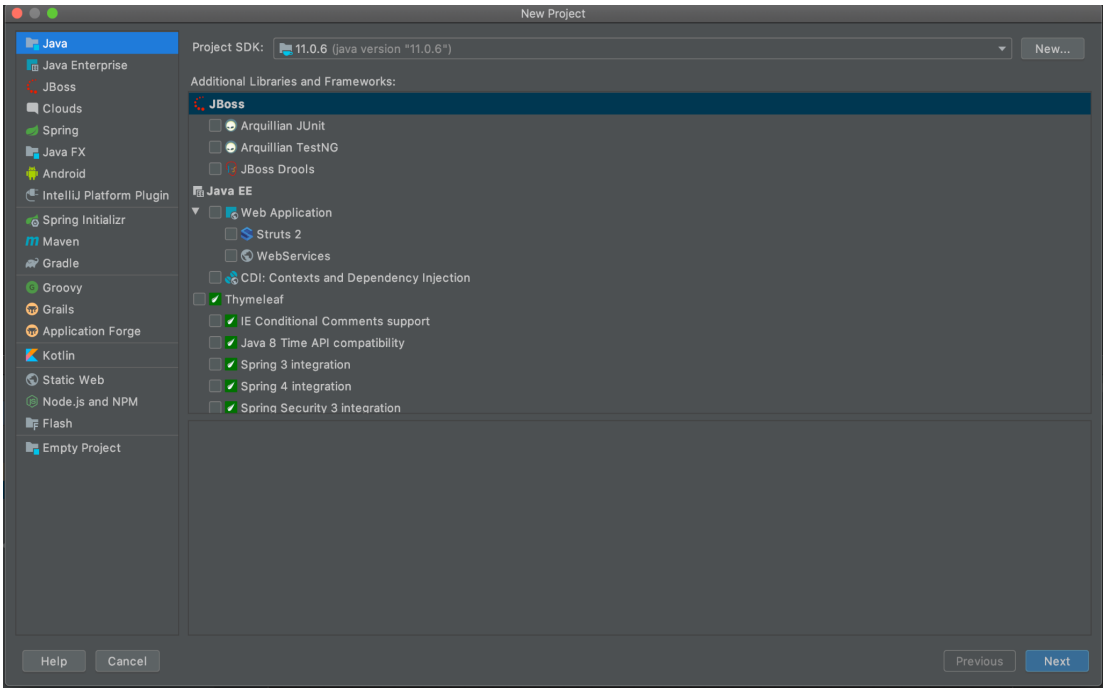
The full list of Interfaces can be checked here or in the Burp Interface list.

Setup DEV environment using IntelliJ

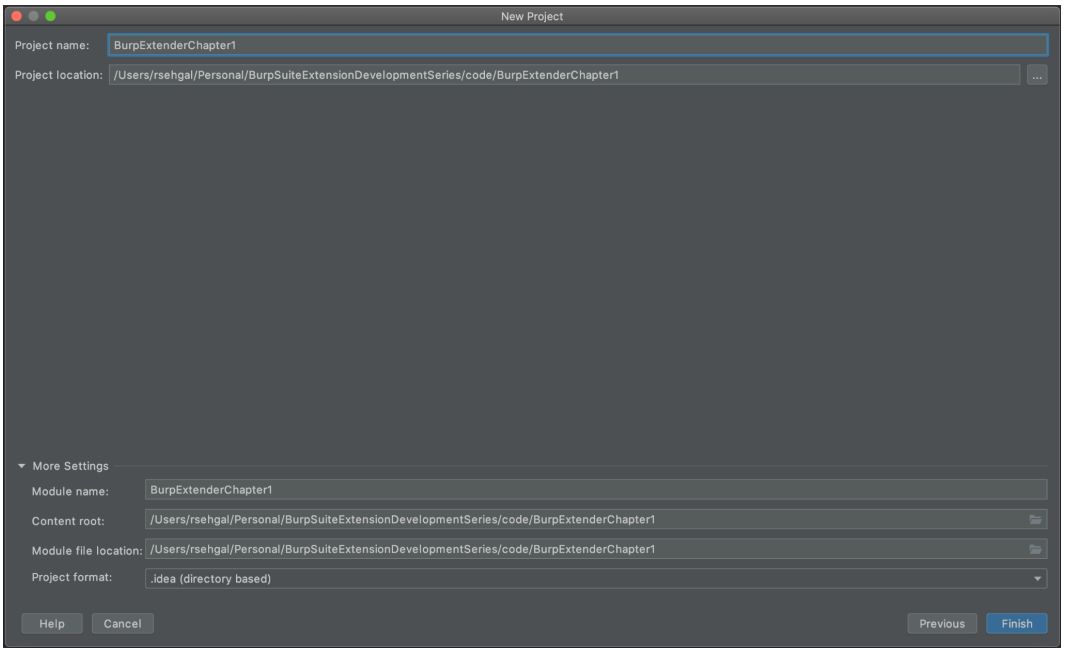
I am using IntelliJ for creating the septu, but you are free to use the IDE of your choice. Also in the next section I will also talk about a **couple of commands for building this jar file** from the command line itself. With an IDE like IntelliJ artifact creation which is jar here, becomes really simple.

1. Create a new Java Project in IntelliJ. Following the steps in the Image one after the other.

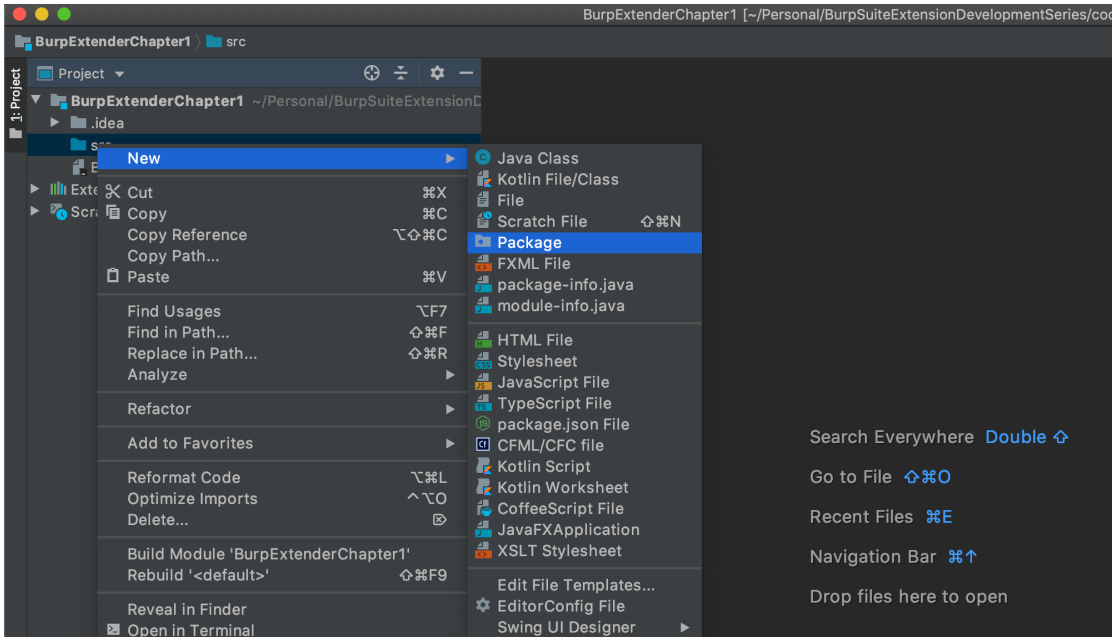


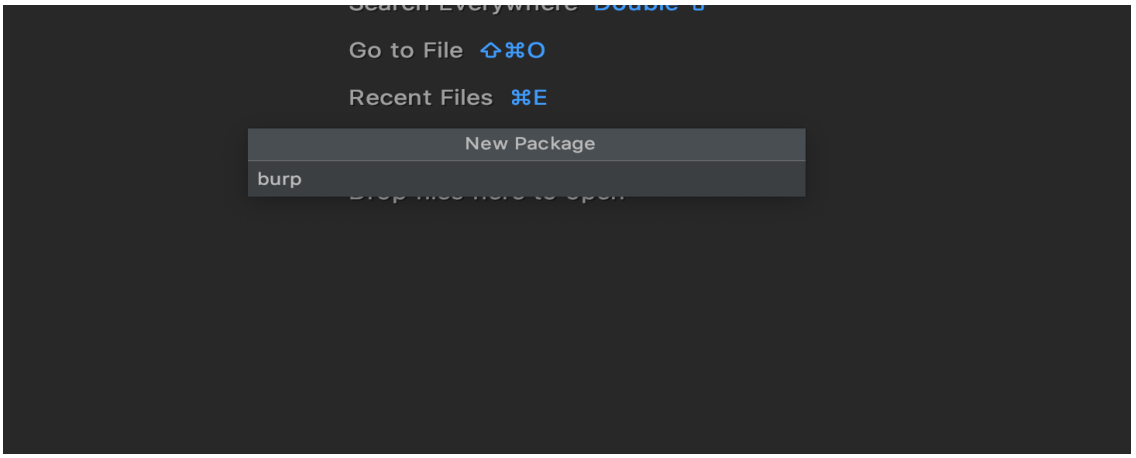

New Project
Java
Java Enterprise
JBoss
Clouds
Spring
Java FX
Android
IntelliJ Platform Plugin
Spring Initializr
Maven
Gradle
Groovy
Grails
Application Forge
Kotlin
Static Web
Node.js and NPM
Flash
Empty Project
Project SDK:
11.0.6 (java version "11.0.6")
New...
Additional Libraries and Frameworks:
JBoss
Arquillian JUnit
Arquillian TestNG
JBoss Drools
Java EE
Web Application
Struts 2
WebServices
CDI: Contexts and Dependency Injection
Thymeleaf
IE Conditional Comments support
Java 8 Time API compatibility
Spring 3 integration
Spring 4 integration
Spring Security 3 integration
Help
Cancel
Previous
Next

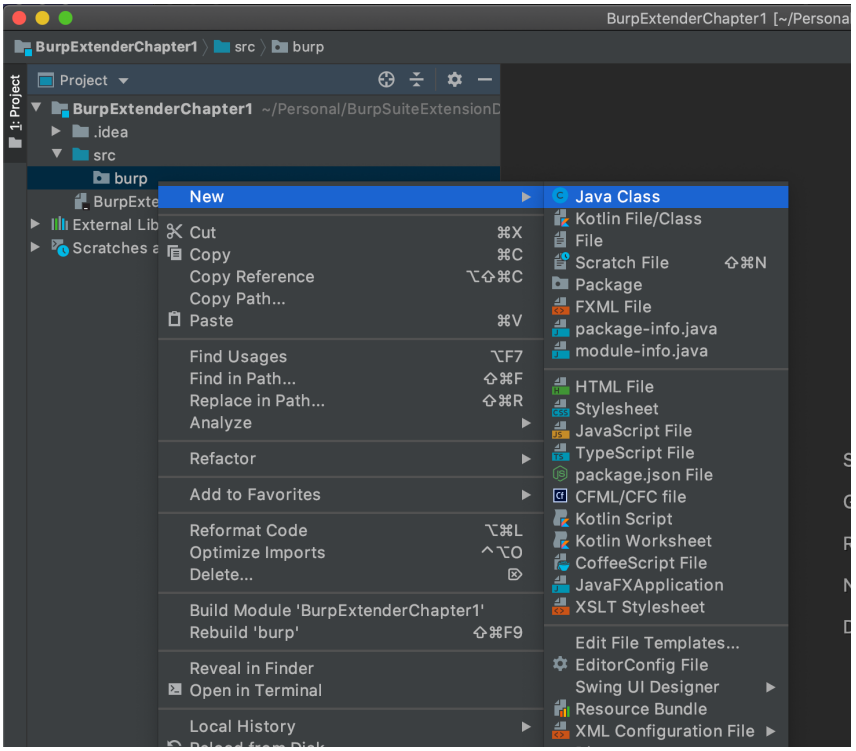

New Project
Create project from template
Command Line App
Help
Cancel
Previous
Next

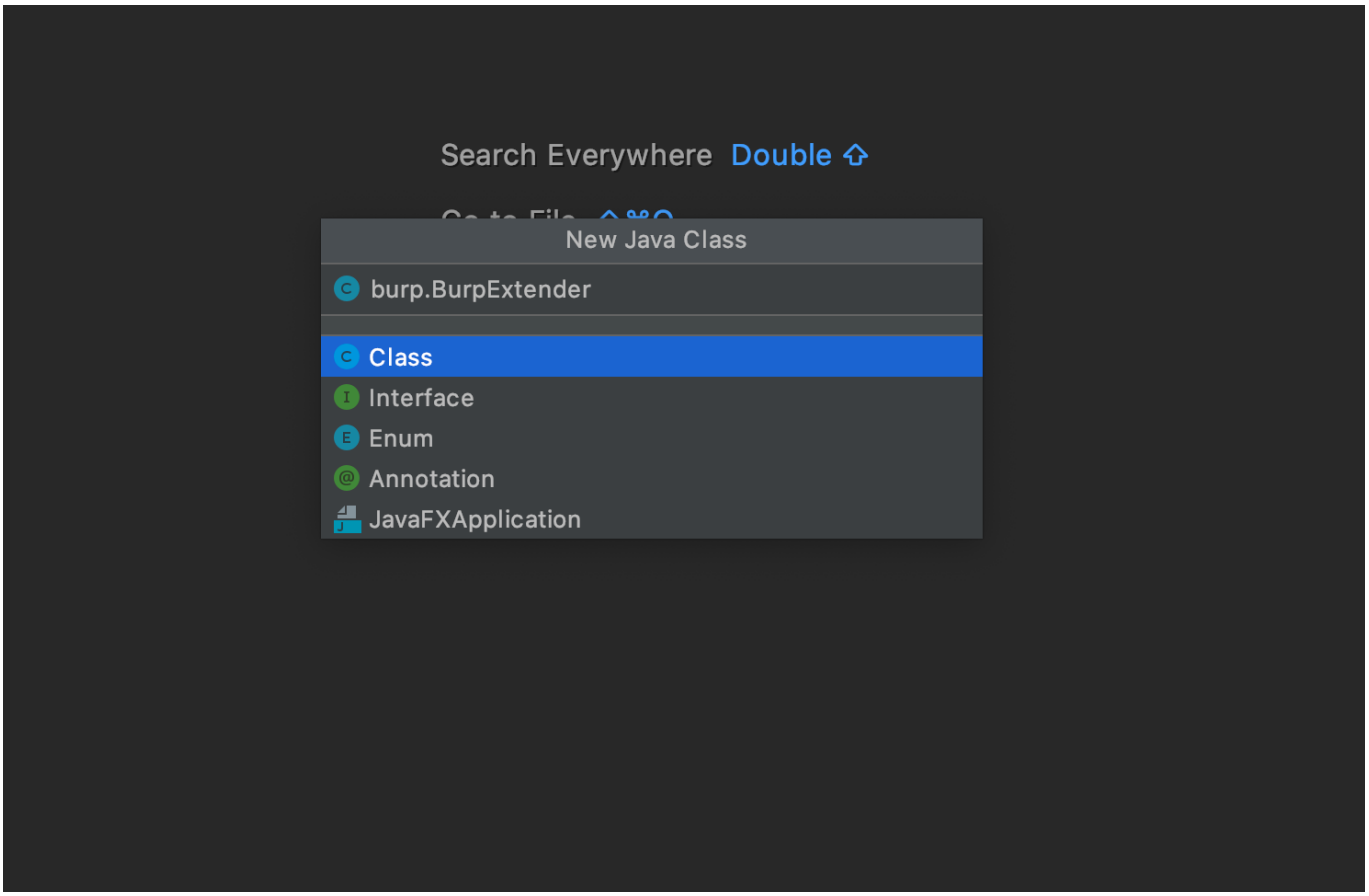


2. Create a package for your Java application, this is required for bundle(**jar**) creation. Once the package is created, **create a file under the package with name BurpExtender.java**. Follow the steps in the images below for more clarity.

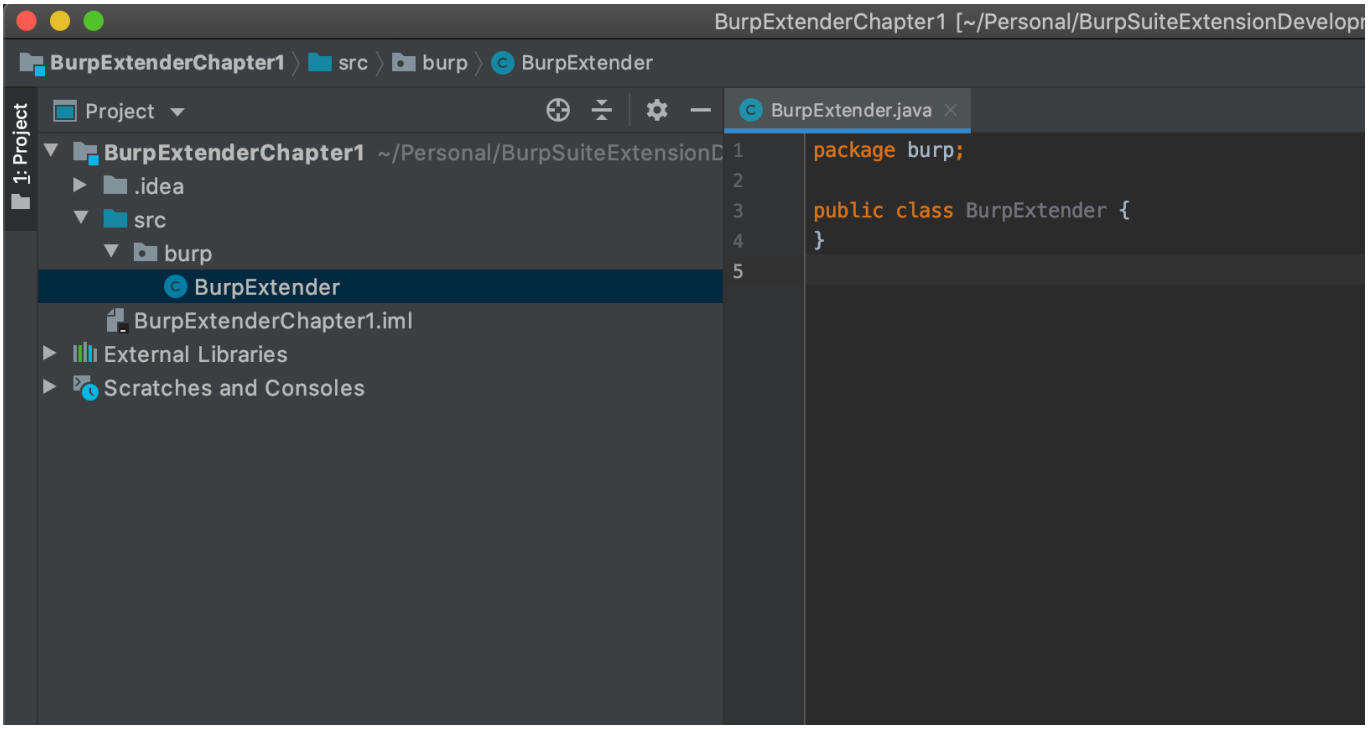



Go to File ⇧⌘O
Recent Files ⌘E
New Package
burp

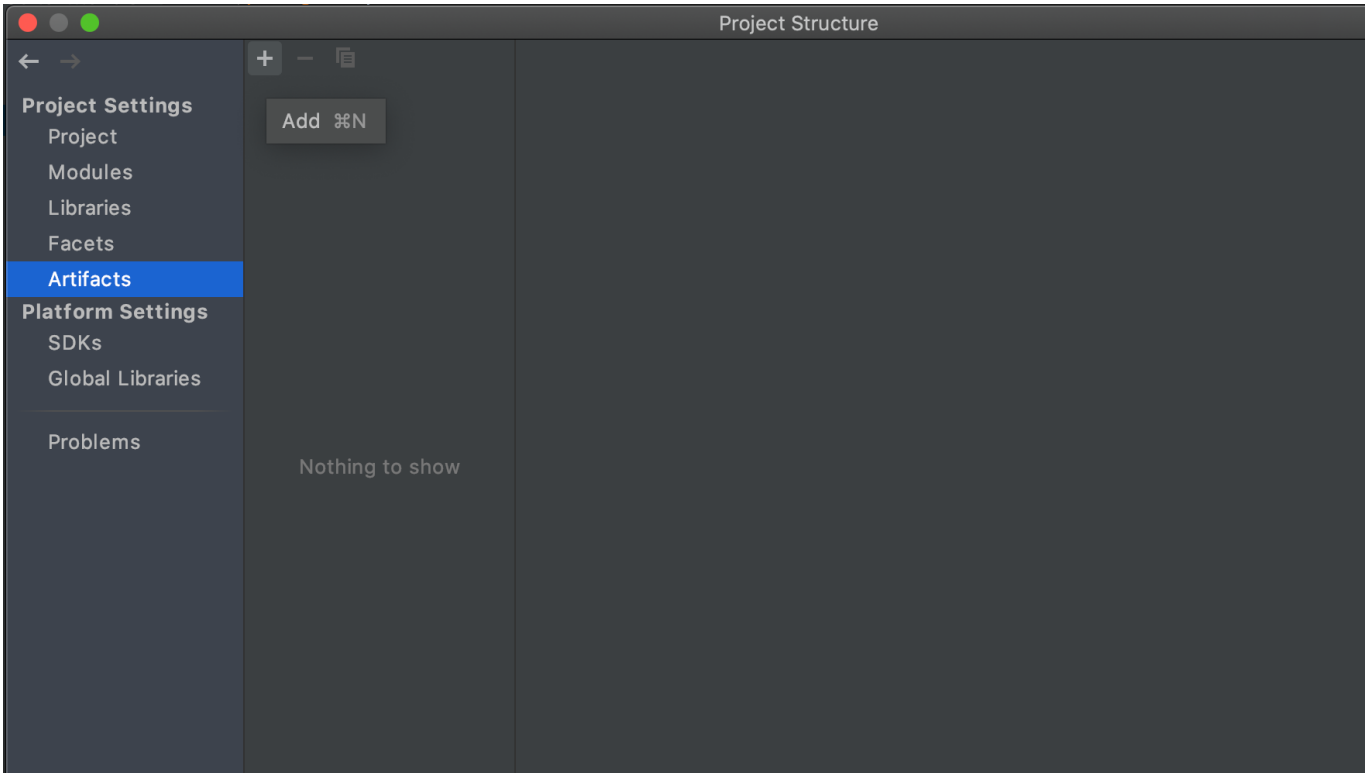

BurpExtenderChapter1 [~/Persona
BurpExtenderChapter1 src burp
Project
BurpExtenderChapter1 ~/Personal/BurpSuiteExtensionD
.idea
src
burp
External Lib
Scratches a
New
Cut ⌘X
Copy ⌘C
Copy Reference ⌥⇧⌘C
Copy Path...
Paste ⌘V
Find Usages ⌥F7
Find in Path... ⇧⌘F
Replace in Path... ⇧⌘R
Analyze
Refactor
Add to Favorites
Reformat Code ⌥⌘L
Optimize Imports ^⌥O
Delete...
Build Module 'BurpExtenderChapter1'
Rebuild 'burp' ⇧⌘F9
Reveal in Finder
Open in Terminal
Local History
Java Class
Kotlin File/Class
File
Scratch File ⇧⌘N
Package
FXML File
package-info.java
module-info.java
HTML File
Stylesheet
JavaScript File
TypeScript File
package.json File
CFML/CFC file
Kotlin Script
Kotlin Worksheet
CoffeeScript File
JavaFXApplication
XSLT Stylesheet
Edit File Templates...
EditorConfig File
Swing UI Designer
Resource Bundle
XML Configuration File

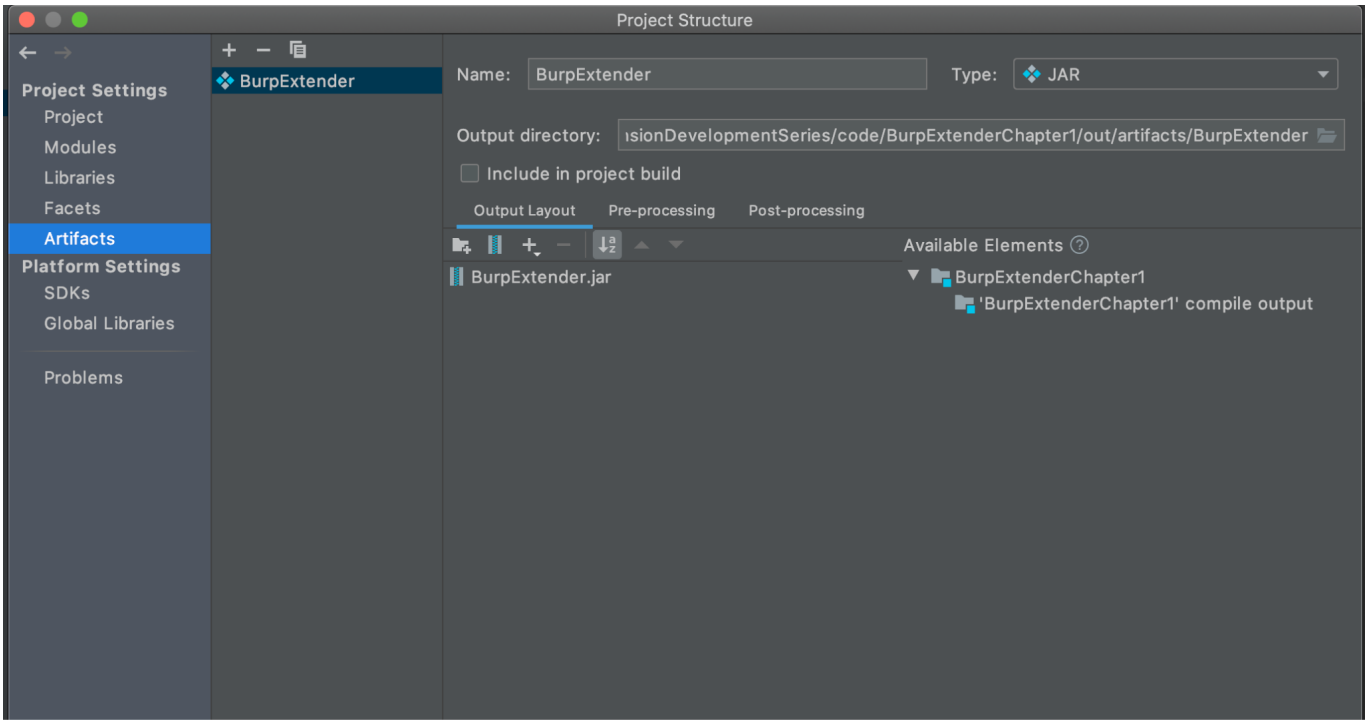


3. Setup **Artifact built** with IntelliJ. This step is necessary to create a jar for your Burp Extender that you will load in your Burp. Once you have completed the above step, you will have a directory structure of something like this.



Now the next step is to set up an artifact build which in our case will be Jar. Quickly navigate through, File -> Project Structure -> Project Settings -> Artifacts, then click on + which is Add.





Click Apply or Ok. With this, now everytime you run build, you will have your artifact which is Jar created in the build folder. This Jar can be directly imported in the Burp Suite.

Using CLI interface to build jars

1. Create two directories, **build** and **bin**. **Bin** for storing output **jar** and **Build** for storing compiled **java files, which are .class files**.
2. Once these folders/dirs are created, we are ready to compile java code into a jar, with the following commands.

```
mkdir  build
mkdir bin
# clean those directories if they are already existing

javac -d build src/burp/*.java
# above command will compile all the java files to .class files under the build
folder.

java cf bin/BurpExtender.jar -C build burp
```

Once the files are compiled, you can load the created binary in the Burp Extender interface.

Code for all the chapters including this is present inside the **code** folder along with this book.

Chapter 2 : Hello-World - Burp Suite Extender

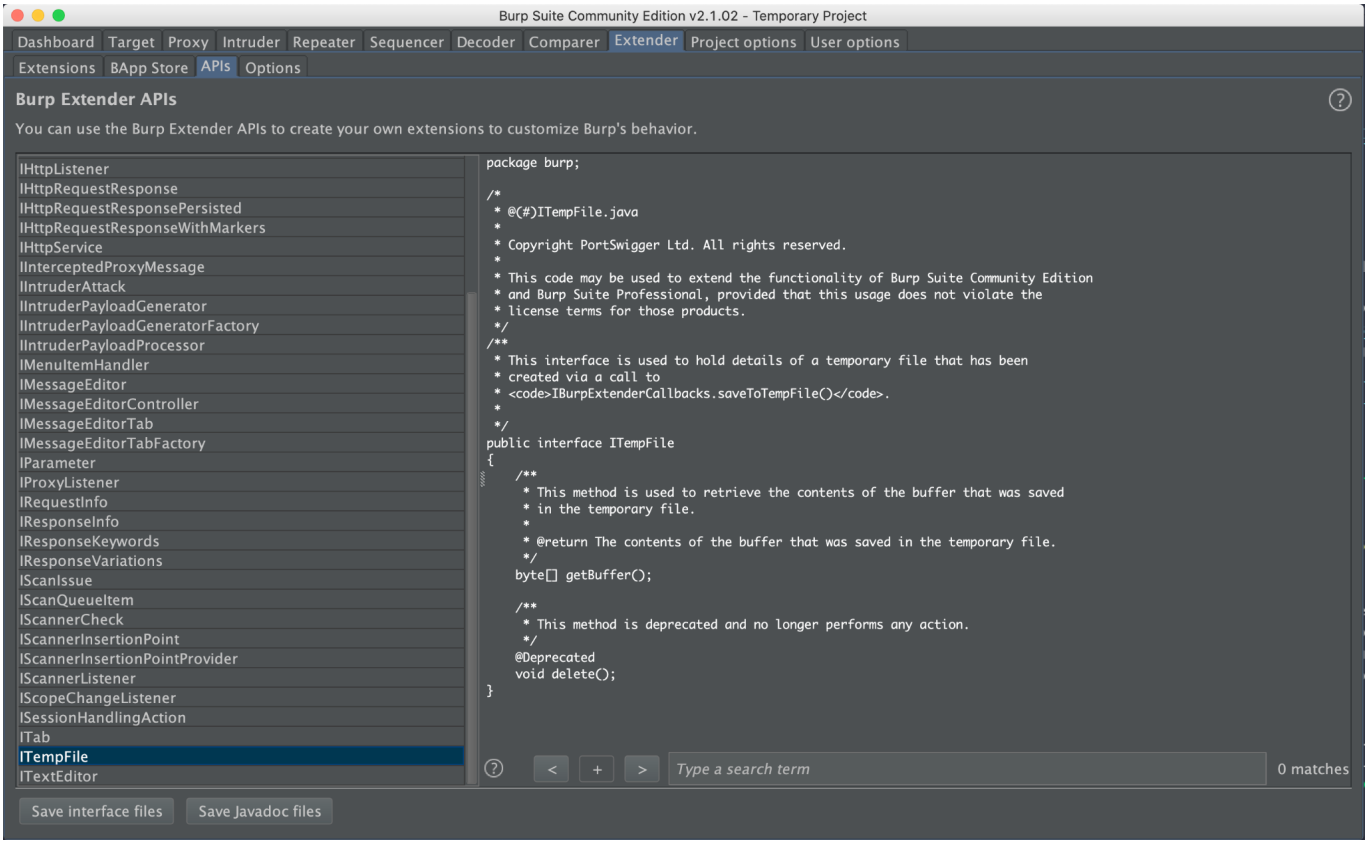
In this chapter we will build a simple hello-world kind of plugin for Burp suite. This plugin will just load in the Burp Runtime and will run in the background and print hello-world string to log.

This chapter talks about:

1. Setting up Extender Development Environment
2. Hello Burp
3. Understanding Hello Burp

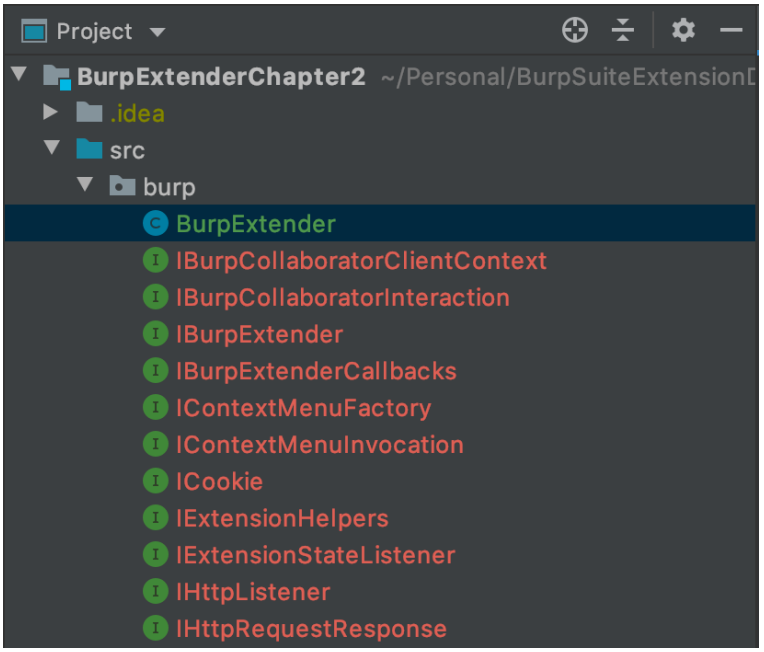
Setting up extender development environment

This step is the continuation of the steps discussed in Chapter 1. In this continuation, we will now discuss how to load Extender interfaces from Burp to our dev environment. This step is pretty straight forward. Navigate to Burp -> Extender -> APIs -> Save Interfaces.



Click on Save Interface Files and **save it in the directory where you have stored the BurpExtender.java file**. Make sure it belongs to the **burp** package, which is the same for **BurpExtender.java**.

Once you do this, your Extender directory would look something like this.



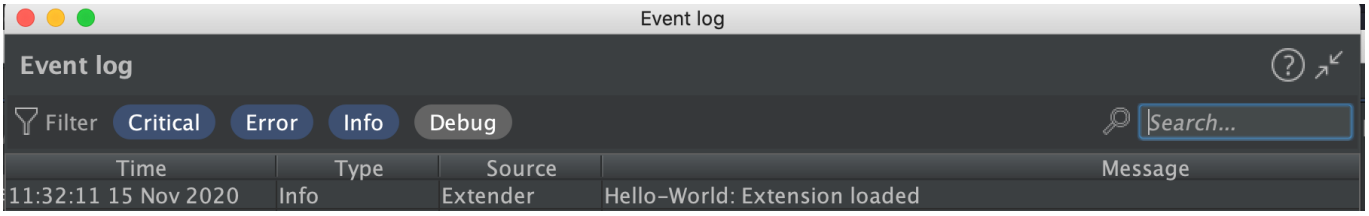
Hello Burp

Every Burp Extender plugin needs to implement **IBurpExtender interface**, which is nothing but the entry point of any plugin.
This interface receives a callback and this callback object is used to send and receive signal/events to the Burp runtime.
See the example below, it is using the **setExtensionName** method to set the name of my created extension and tell that to Burp runtime. And then it is using the **setAlert** method for sending messages to Burp runtime to display in the dashboard.

```
package burp;

public class BurpExtender implements IBurpExtender{
    @Override public void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks) {
        callbacks.setExtensionName("Hello-World");
        callbacks.setAlert("Extension Loaded")
    }
}
```

To see this in action, build the jar artifact either through IDE or through CLI interface as discussed in Chapter 1.
Once the artifact is built, you need to load it into the BurpSuite Environment. Click on Add, and select the location of the Jar file. This step is pretty straight forward.

You can see the result of the loaded extension in the dashboard.
WOW 😄 we just built and loaded an extension into Burp. Kudos to you, to achieve this landmark.



Understanding Hello Burp

Hello World for burp looks something like this below code snippet, and now let's try to understand the tech jargon.

```
public class BurpExtender implements IBurpExtender{
    @Override public void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks) {
        callbacks.setExtensionName("Hello-World");
        callbacks.setAlert("Extension Loaded")
    }
}
```

1. As discussed in Chapter 1, every burp extender plugin needs to implement IBurpExtender. This line `public class BurpExtender implements IBurpExtender` does the same thing with `@Override` annotation
2. **IBurpExtender** has a **registerExtenderCallbacks** which takes input from Burp UI and receives a callback as and when this plugin is called. Callback exposes multiple methods which can be used to interact Burp Suite Runtime such as :

- `setExtensionName()`: Set our extension name
- `makeHttpRequest()`: Send an HTTP request and retrieve its response.
- `issueAlert()`: Send a message to Burp's Alert Tab.
- `getBurpVersion()`: Figure-out the version of Burp Suite.
- `isInScope()`: Check if a URL is in Burp Project's scope or not.
  More information about all the callback methods can be found here. This callback object contains all the necessary functionality which is required to talk to Burp runtime.

3. The above code calls the `setExtensionName()` method to set the name of the extension that will be visible in Burp Suite UI; there-after `issueAlert()` is used to send an alert to the Alert tab of Burp Suite with a message. Please note that the parameters and return types of the functions have to be referred from their API reference. The Extender API reference is the to-go place to discover golden extensibility gems offered by Burp Suite.

Extension says Hello World!

Chapter 3 : Deep Dive into Extender API Interface

So far we learned how to write a basic Hello World extension showcasing interaction with Burp Suite.

In this chapter, we'll mostly cover the Extender API interfaces and their use cases. In the Chapter's end, we **will create an extension which will monitor HTTP requests from Burp Suite and display the domains passing through the Proxy in the Alert tab**. Nice isn't it ?

In this chapter we will be covering:

1. Helper Interface
2. Simple URL Encoder
3. Interface Registration
4. Listen for events from Proxy

Helper Interface

In the previous extender plugin that we created, we used callbacks object's methods twice for setting the name of the extension and displaying a message on Alert tab, right ? Callback interface is really great, it offers plenty of other resources as instance objects as well.

One of the important methods exposed by callbacks is `getHelpers()`. This method returns an object of `IExtensionHelperstype`, which as the name suggests will be going to help us in making boring and repetitive tasks easier. The object contains multiple methods, few of important ones are listed here to throw an insight:

- `analyzeRequest()`: This method can be used to analyze an HTTP request, and obtain various key details about it. Like request headers, cookies, etc.
- `analyzeResponse()`: This method can be used to analyze an HTTP response, and obtain various key details about it.
- `base64Encode()`: This method can be used to Base64-encode the specified data.
- `base64Decode()`: This method can be used to decode Base64-encoded data.
- `urlDecode()`: This method can be used to URL-decode the specified data.
- `urlEncode()`: This method can be used to URL-encode the specified data

  and many more. The full list of Extension helpers can be found here.

Simple URL encoder

A slight detour, to understand the helper interface.

Let's create a very simple static URL encoder with the helpers interface to get a better understanding of what helper interfaces are. We will use the same process of creating a base class to implement `IBurpExtenderInterface` which will receive a as a callback object, we will use it to get a helper instance and eventually create a encoded string for static URL and display encoded text in Alerts tab. Code is pretty straight forward and self explanatory.

```
package burp;
public class BurpExtender implements IBurpExtender {

    @Override
    public void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks) {
        callbacks.setExtensionName("URL Encoder");

        // This is how we receive helper's object reference from callback
instance.
        IExtensionHelpers helpers = callbacks.getHelpers();

        String encodedString =
helpers.urlEncode("http://www.example.com/dir/path?q=10&a=100");
        callbacks.issueAlert(encodedString);
    }
}
```

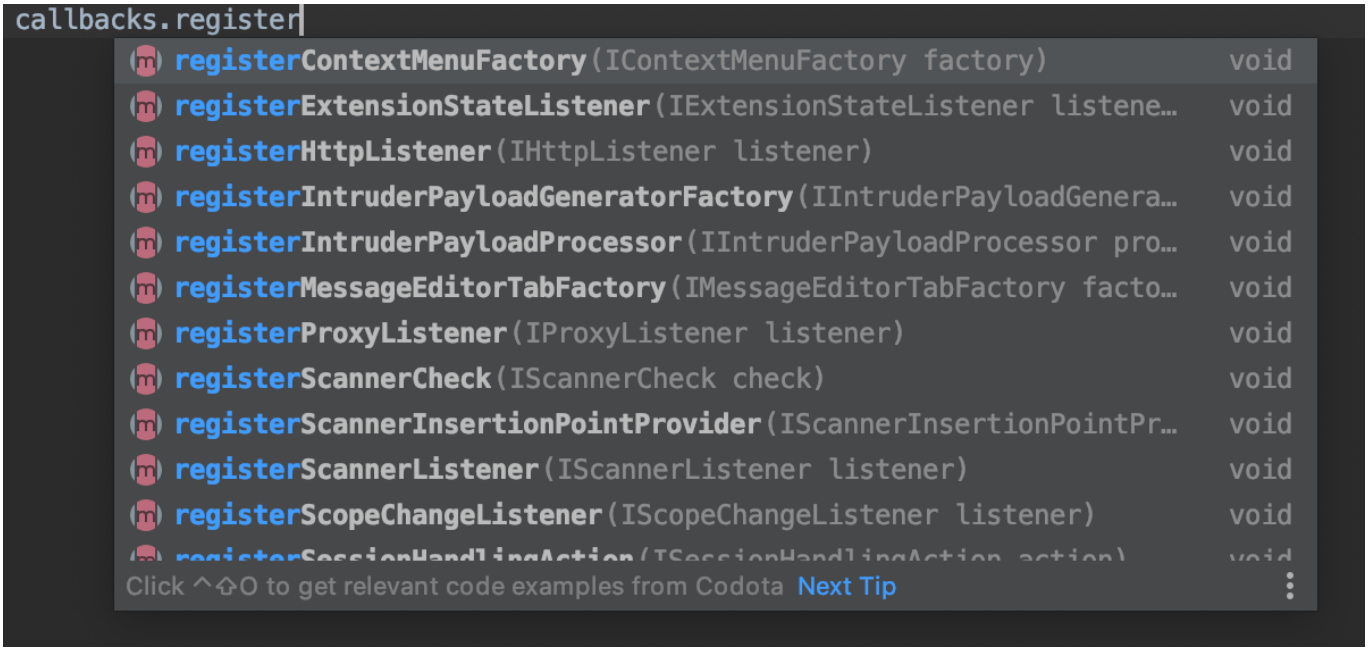
Interface registration

Let's come back to our original case of creating a plugin which will monitor all the HTTP requests.

Interface registration is important and is useful in telling Burp Runtime what our custom code is capable of handling. For doing this, you need to tell the Burp Runtime with callbacks's `register*` functionality. If you plan to handle different events you will need to register their respective interface registration function.

As an example, if you are implementing an `IHTTPListener` interface through any of the classes, then you must have the object of those classes passed to `callbacks.registerHttpListener()` method.

Note that callbacks itself is an object of type IBurpExtenderCallbacks. IntelliJ loaded with all interface files will show you suggestions something like this.



So, Each API interface that you will implement will have a separate register method corresponding to it.

Listen for Events from Proxy

Now finally lets create a class, which will implement the `IHTTPListener` interface. The class will:

1. Listen for events from the Burp Suite Proxy tab.
2. Class will be notified for request and response from Burp Proxy.
3. Class will log all the requests in the Alerts tab.

Create a class which will implement the `IHTTPListener` interface. In this interface we would need to implement `processHttpMessage` method.

Read the comments for an explanation.

```
/*
   LogProxyRequests.java
*/

package burp;

public class LogProxyRequests implements IHttpListener{

   private IBurpExtenderCallbacks iBurpExtenderCallbacks;
   private IExtensionHelpers iExtensionHelpers;
```

```
    public LogProxyRequests(IBurpExtenderCallbacks callbacks){
        /*
            For issuing alerts to the Alert tab.
         */
        iBurpExtenderCallbacks = callbacks;

        /*
            For parsing requests.
         */
        iExtensionHelpers = callbacks.getHelpers();
    }

            @Override public void processHttpMessage(int toolFlag, boolean messageIsRequest,
IHttpRequestResponse messageInfo) {
        IRequestInfo requestInfo = null;

        /*
            Only listen for events from Burp Suite proxy && Only listen for requests.
         */
        if(toolFlag == IBurpExtenderCallbacks.TOOL_PROXY && messageIsRequest == true)
            requestInfo = iExtensionHelpers.analyzeRequest(messageInfo);
            String domainName = requestInfo.getUrl().getHost();

            /*
                Log the domain name to the Alerts tab.
             */
            iBurpExtenderCallbacks.issueAlert("Proxy: " + domainName);
    }
}
```

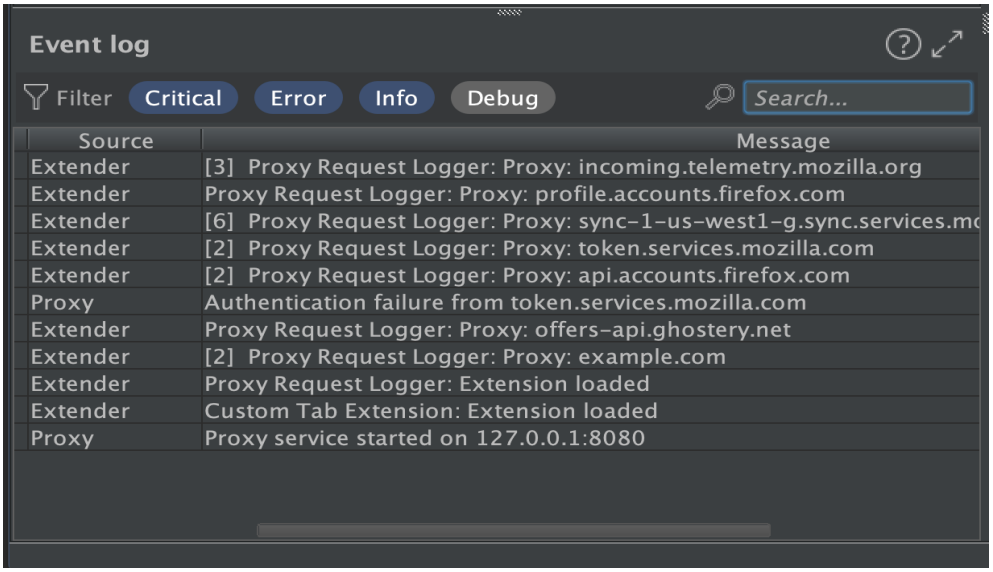
Register the implemented class in Our BurpExtender.java class file through callbacks.

```
/*
    BurpExtender.java
*/

package burp;

public class BurpExtender implements IBurpExtender{
    @Override public void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks) {
        callbacks.setExtensionName("Proxy Request Logger");
        callbacks.issueAlert("Extension loaded");
```

```
        /*
            Register our LogProxyRequest instance to burp suite proxy.
         */
        callbacks.registerHttpListener(new LogProxyRequests(callbacks));
    }
}
```

Now we have two different files where the logic for our plugin resides. That's awesome, also we can see dots are now getting connected.

Build the jar and load the jar in Burp Suite. If everything goes well, you will get domain names getting logged in alerts tab.



Hurrah !!! We finally created a plugin which can help you intercept burp suite traffic, modify on the fly and send the domains to the alert tab.
Although in the current state there might not be great use of this plugin, here I dont want to teach to build specific plugins but my idea is to make you learn about the methods which are possible which exist and how those methods can be used to to create a specific functionality.

Chapter 4: Example Plugin - Intruder Payload Processor

This chapter is a short one that discusses creating a plugin for a use case. **The plugin will process the payload from the Intruder and will execute the payload after processing**. To implement this we need to implement IIntruderPayloadProcessor. The process is fairly simple as we have done in our previous cases. Create a class which implements this `IIntruderPayloadProcessor` and then register the instance of the class with `IBurpExtender.callbacks.registerIntruderPayloadProcessor()`.

So let's connect the dots again.

`IIntruderPayloadProcessor` interface contains two method of the following structure, these will be overridden in the implementing class which are necessary to bring the functionality that we need.

- `string getProcessorName()`: This will provide the name of the payload processor in the Burp Suite UI.

- `byte[] processPayload(byte[] currentPayload, byte[] originalPayload, byte[] baseValue)`: This is the responsible function for payload processing, the processed payload should be returned as a **byte array**. We can utilise **helpers to convert String to byte[]** by invoking `stringToBytes()`; to achieve the opposite `bytestoString()` can be used from the `callbacks.getHelpers()`

    - currentPayload - The value of the payload to be processed.
    - originalPayload - The value of the original payload prior to processing by any already-applied processing rules.
    - baseValue - The base value of the payload position, which will be replaced with the current payload.
- So to create a custom payload from the existing payload, change the value of current payload to something you wish.

For the demonstration purpose we would like to create a basic base64 payload processor. The idea is that it will take a current payload and then create a base64 encoded payload and use that payload as intruder payload.

Look at **this** self explanatory code.

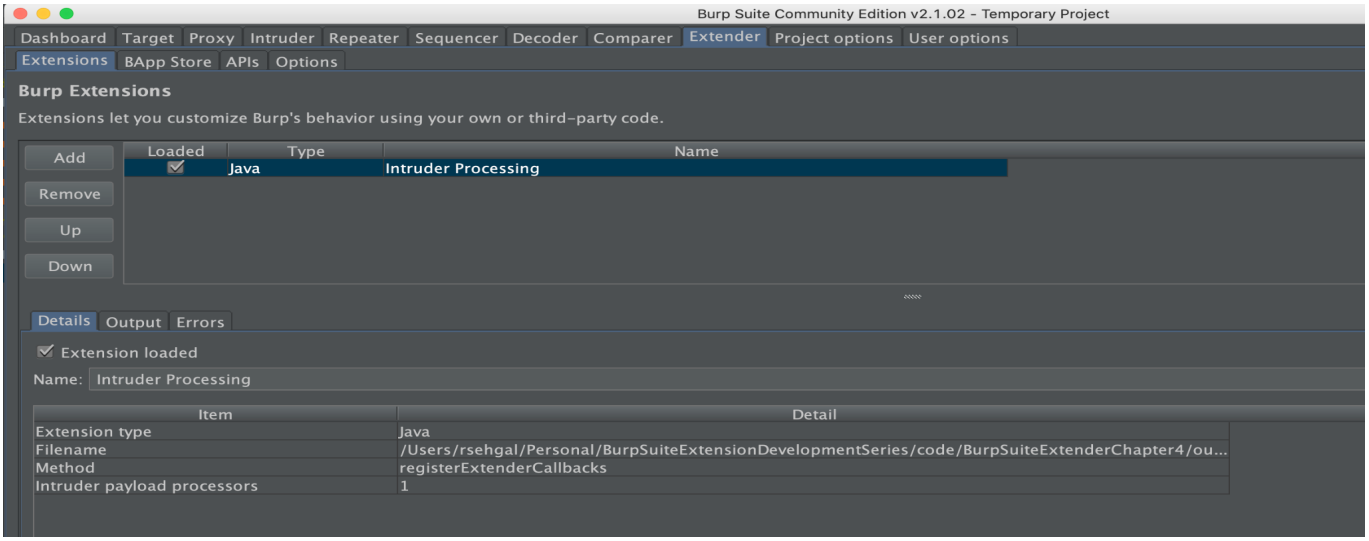
```
/*
IntruderProcessor.java
*/
package burp;

public class IntruderPayloadProcessor implements IIntruderPayloadProcessor{

    IExtensionHelpers helpers;
```

```
    public IntruderPayloadProcessor(IBurpExtenderCallbacks callbacks){
        /*
            We can use the helpers string to byte array method as processPayload need to return byte[]
         */
        helpers = callbacks.getHelpers();
    }

    @Override public String getProcessorName() {
        /*
            This name will be shown in the Burp Suite Intruder UI as processor name.
         */
        return "Base64 Processor";
    }

        @Override public byte[] processPayload(byte[] currentPayload, byte[] originalPayload, byte[]
baseValue) {
        if(currentPayload != null){
            return helpers.stringToBytes(helpers.base64Encode(currentPayload));
        }
        return null;
    }
}
```

Tell burp that you have a payload processor.

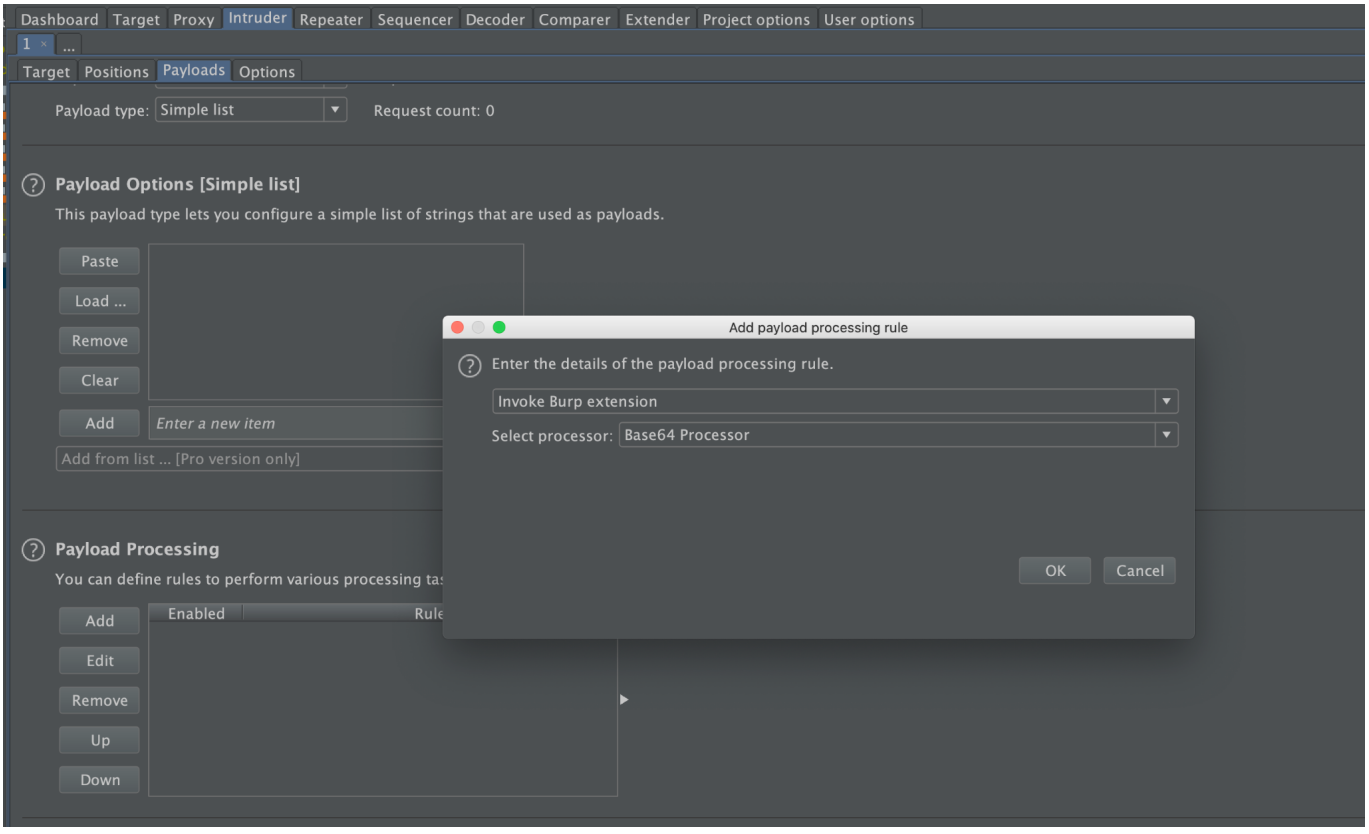
```
/*
BurpExtender.java
*/

package burp;

public class BurpExtender implements IBurpExtender {
    @Override public void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks) {
        callbacks.setExtensionName("Intruder Processing");
        callbacks.issueAlert("Plugin loaded");

        // Notice this.
        callbacks.registerIntruderPayloadProcessor(new IntruderPayloadProcessor(callbacks));
    }
}
```

Build the artifact, load it in Burp.



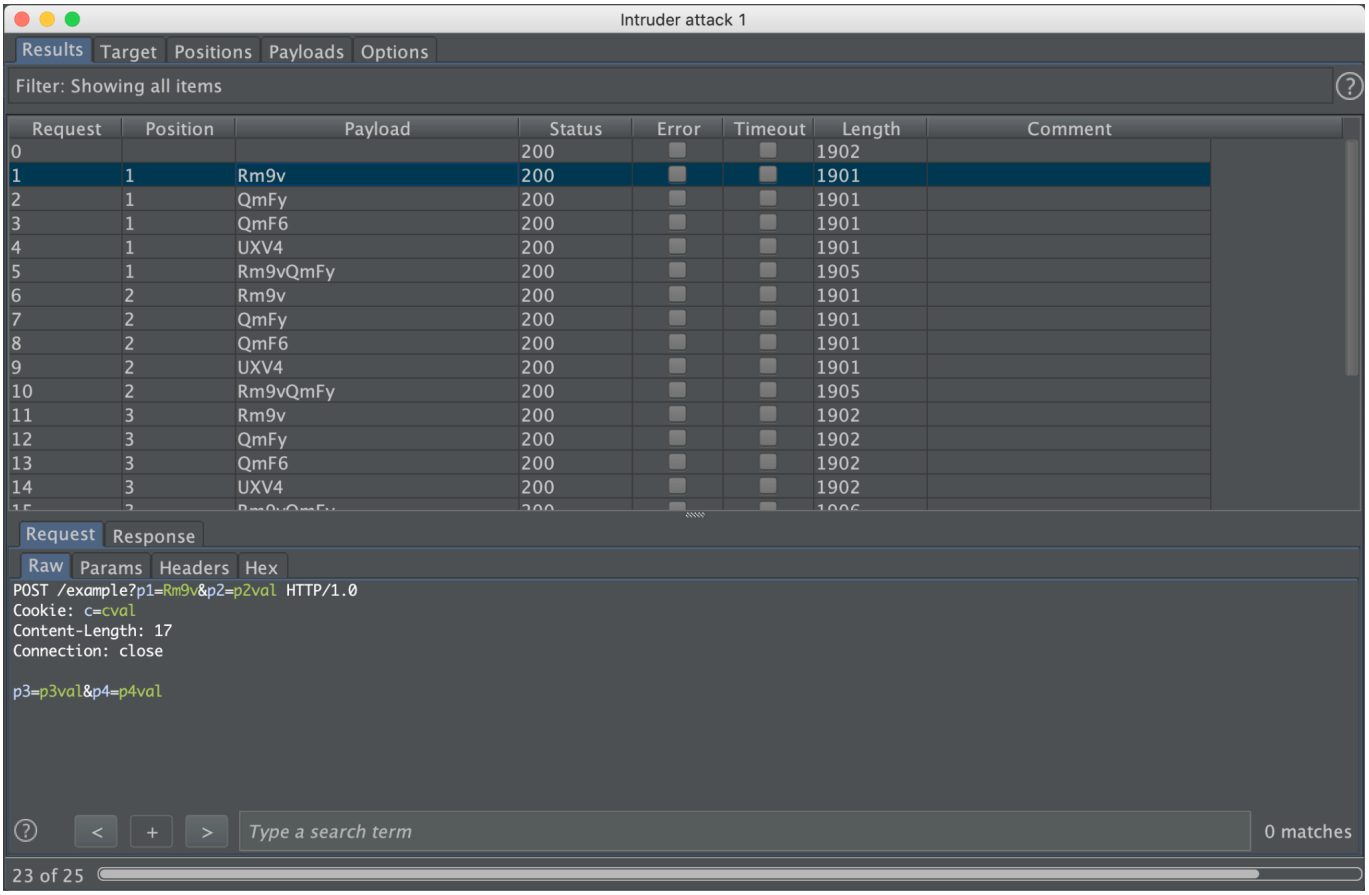
In the item you can see that, IntruderProcessor is coming up.

Now this Payload processor will pop up in Intruder, payload processor like follows.



The name you have entered for Payload processor in the implementing class will come under drop down. Since I entered the Base64 Processor, it is showing as it is.

Run it and check the payloads are getting converted to base64 by our processor and are executed likewise.



Now every payload that the intruder uses, will be processed first, converted to base64 first with our payload processor that we wrote and then use it to make an actual attack to target.

Chapter 5: Burp Suite Extender Plugin : Event Listeners

This chapter talks about how to register event listeners for various use cases. Event listener creation and registration we have already seen in the previous post Listen for events from Proxy.

This chapter is just the extension of those principles on similar ground. This chapter in detail talks about Event listeners, and how they work in flow with Burp Callbacks. This chapter will also demonstrate how different event listeners can be configured in one single BurpExtender class.

Code will be registering Event Handlers from : `IHttpListener, IProxyListener, IScannerListener, IExtensionStateListener` Interfaces after implementing them.

- `IHttpListener`:
  - The listener will be notified of requests and responses made by any Burp tool, any http request and response.
  - Extensions are useful in case of custom analysis or modification of HTTP messages.
- `IProxyListener`:
  - The listener will be notified of requests and responses being processed by the Proxy tool.
  - Extensions are useful in case of custom analysis or modification of these messages coming from the proxy tab. This will exclude the events from Scanner, Repeater etc.
- `IScannerListener`:
  - The listener will be notified of new issues that are reported by the Scanner tool.
  - Extensions are useful in case of custom analysis or logging of Scanner issues by registering a Scanner listener.
- `IExtensionStateListener`:
  - The listener will be notified of changes to the extension's state.
  - Extensions are useful in case you want to check if any other extension is loaded or not. This can be useful in checking if the extension will need to interact with other extensions. Yes in burp that too is possible.

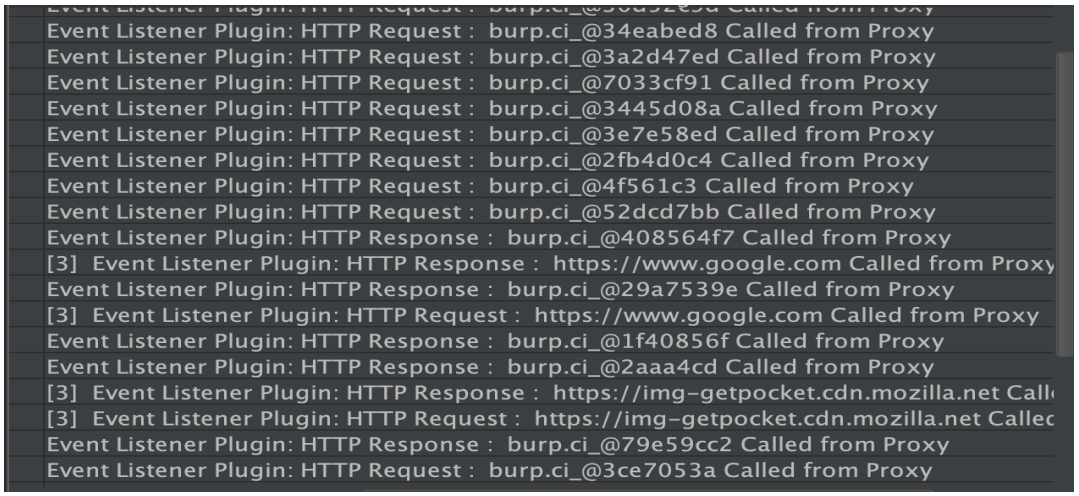
** Note: Any extensions that start background threads or open system resources (such as files or database connections) should register a listener and terminate threads / close resources when the extension is unloaded.

See this Self Explained Example for Event listeners:

```
package burp;

public class BurpExtender implements IBurpExtender,
        IHttpListener, IProxyListener, IScannerListener, IExtensionStateListener {

    IBurpExtenderCallbacks callbacks;

    /* This we have already seen so far, this method we mostly use to register out Extender to Burp, by
setting name
    getting a helper, setting plugin level alerts etc.
     */
    @Override public void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks) {
        callbacks.setExtensionName("Event Listener Plugin");
        callbacks.issueAlert("Plugin loaded");

        this.callbacks = callbacks;

        /*
         Since this class is implementing all the interfaces, this class must register itself as listener for all
the
        implementations.
         */
        callbacks.registerHttpListener(this);
        callbacks.registerProxyListener(this);
        callbacks.registerScannerListener(this);
        callbacks.registerExtensionStateListener(this);

        /*
          Since we are implementing event listener for 4 different class, we are registering it for 4 times
for various
         Listeners.
         Plugin registration is required to let Burp know what all events this plugin is looking for.
         */
    }

    /*
    This function belongs to IHTTPListener Interface.

      @toolFlag : take the Burp Suite plugin number, number to tool name can be obtained from
callback.getToolName()
    @messageIsRequest : True if request, false id response
    @ messageInfo : Encapsulating details about an event.
     */
          @Override public void processHttpMessage(int toolFlag, boolean messageIsRequest,
```

```
IHttpRequestResponse messageInfo) {
        callbacks.issueAlert(
                String.format("%s %s Called from %s",
                        messageIsRequest ? "HTTP Request : " : "HTTP Response : ",
                        messageInfo.getHttpService(),
                        callbacks.getToolName(toolFlag))
            );
    }

    /*
    This function implements IProxyListener
    @messageIsRequest : True if request, false id response
    @message : Encapsulating details about an event.
     */
                    @Override    public    void    processProxyMessage(boolean    messageIsRequest,
IInterceptedProxyMessage message) {
        callbacks.issueAlert(
                String.format("%s %s Called from %s",
                        messageIsRequest ? "HTTP Request : " : "HTTP Response : ",
                        message.getMessageInfo(),
                        "Proxy")
        );
    }
    /*
    Implements IScannerListener
    @issue: encapsulates the details about the scan event.
     */
    @Override public void newScanIssue(IScanIssue issue) {
        callbacks.issueAlert("Scan triggered : " + issue.getIssueName());
    }

    /*
    This function implements IExtensionStateListener.
    */
    @Override public void extensionUnloaded() {
        callbacks.issueAlert("Extension Unloaded");
    }
}
```

If everything goes well, output would look something like this 😄


Event Listener Plugin: HTTP Request : burp.ci_@34eabed8 Called from Proxy
Event Listener Plugin: HTTP Request : burp.ci_@3a2d47ed Called from Proxy
Event Listener Plugin: HTTP Request : burp.ci_@7033cf91 Called from Proxy
Event Listener Plugin: HTTP Request : burp.ci_@3445d08a Called from Proxy
Event Listener Plugin: HTTP Request : burp.ci_@3e7e58ed Called from Proxy
Event Listener Plugin: HTTP Request : burp.ci_@2fb4d0c4 Called from Proxy
Event Listener Plugin: HTTP Request : burp.ci_@4f561c3 Called from Proxy
Event Listener Plugin: HTTP Request : burp.ci_@52dcd7bb Called from Proxy
Event Listener Plugin: HTTP Response : burp.ci_@408564f7 Called from Proxy
[3] Event Listener Plugin: HTTP Response : https://www.google.com Called from Proxy
Event Listener Plugin: HTTP Response : burp.ci_@29a7539e Called from Proxy
[3] Event Listener Plugin: HTTP Request : https://www.google.com Called from Proxy
Event Listener Plugin: HTTP Response : burp.ci_@1f40856f Called from Proxy
Event Listener Plugin: HTTP Response : burp.ci_@2aaa4cd Called from Proxy
[3] Event Listener Plugin: HTTP Response : https://img-getpocket.cdn.mozilla.net
[3] Event Listener Plugin: HTTP Request : https://img-getpocket.cdn.mozilla.net
Event Listener Plugin: HTTP Response : burp.ci_@79e59cc2 Called from Proxy
Event Listener Plugin: HTTP Request : burp.ci_@3ce7053a Called from Proxy

Chapter 6: Plugin Example : Custom Session Tokens

At times you might want to scan an application with a custom session token set right ? If yes this extender plugin cum tutorial can save you here 😄

This chapter talks about how Burp suite extender APIs can be used to create plugins which can modify the session related information on the fly. This chapter is more of an exercise than a plugin use case.

This chapter comes with a demo server to test the developed plugin. The application server code is here Server. Just run this server as `node server.js`, this will spin up the server at port 8000, where we will be testing our plugin.

If you carefully observe the server.js file, you can notice that session information is stored in the `SESSION_ID_KEY` variable which points to `X-Custom-Session-Id`.

```
const http = require('http');

const PORT = 8000;

const SESSION_ID_KEY = 'X-Custom-Session-Id'
```

So in this chapter or this exercise our task would be to record this session variable in one of the burps macro and then transfer it along with all the subsequent requests and we will do it with our plugin. This might sound complex but the way I have split the chapter, you will see it will be way easy than you have anticipated.

Let's Start.

What is Macro ?

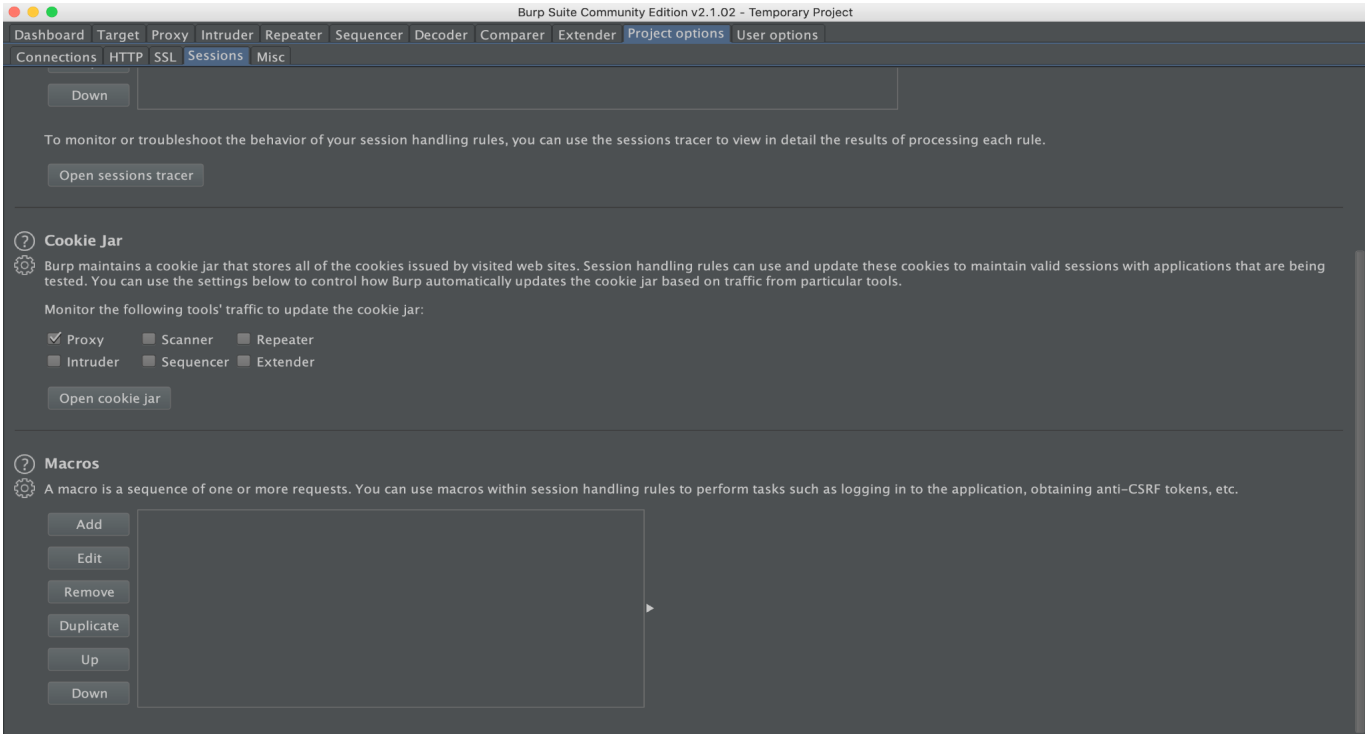
A word about Burp Macro.

- A macro in Burp Suite is a series of HTTP requests to be sent to the server prior to requests which have been proxied by Burp. Once the macro requests have been carried out, the set of parameters can be taken from the response of the final macro request and can then be passed on to the request that called the macro.
- So in short macros are a series of steps which will/can be called before making an actual request by any burp suite entity.

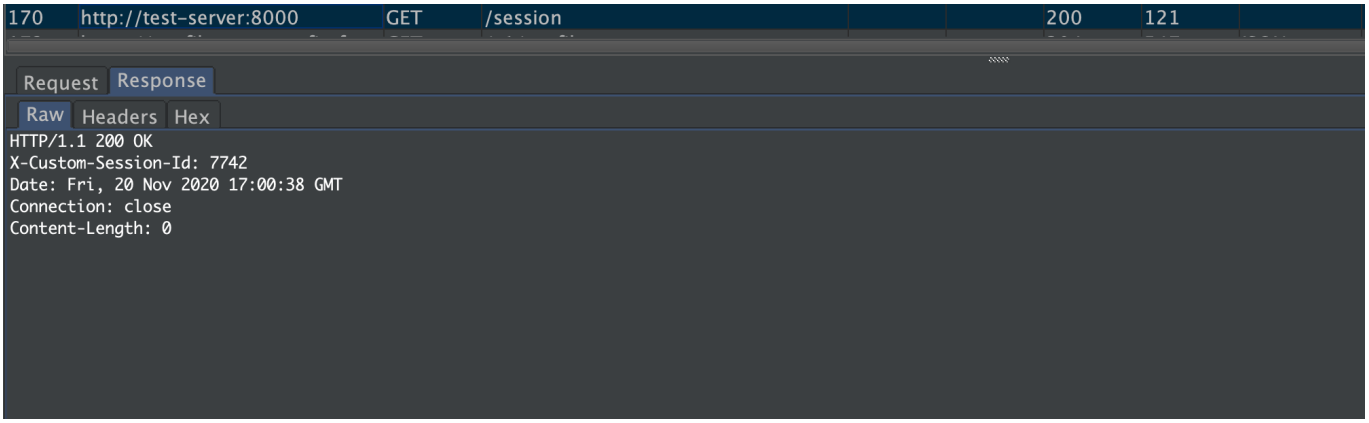
Create a macro

It's simple again. Navigate like this :

Burp -> **Project options** -> **Sessions** -> **Macros** -> **Add**



The custom server code shipped with this chapter has /sessions endpoint that provides a custom session token using `curl localhost:8000/session`, the new session request can be made through `curl -XPOST localhost:8000/session`. So you have to test the server, then you need to request a session token from Macro at `/session` endpoint and then use it to make the call to the actual server target endpoint.



So to record macro for session token generation, make a request to /session endpoint Burp Proxy and record macro as discussed below:

1. Proxy post request **:**
    - **Send a post request from curl:** `curl -XPOST http://test-server:8000/session -x http://localhost:8080` This will send the post request over burp proxy**.**
2. **Goto : Burp -> Project options -> Sessions -> Macros -> Add,** just pick the post request from the history tab. I have named the macro : test-server: Session Token Creation , remember this, as we will need it. 😄

Create Extender Plugin to consume Macro

To consume a macro for the session handling mechanism we need to implement : `ISessionHandlingAction`. So flow is simple again, create a class implementing this Interface then register this call for events.

```
package burp;

import java.util.List;

public class BurpExtender implements IBurpExtender, ISessionHandlingAction {
   IBurpExtenderCallbacks callbacks;
   IExtensionHelpers helpers;

   private static String SESSION_ID_KEY = "X-Custom-Session-Id";
   private static final byte[] SESSION_ID_KEY_BYTES = SESSION_ID_KEY.getBytes();
   private static final byte[] NEWLINE_BYTES = new byte[] { '\r', '\n' };

   @Override public void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks) {
      this.callbacks = callbacks;
      this.helpers = callbacks.getHelpers();

      callbacks.setExtensionName("Custom Session Token");
      callbacks.issueAlert("Plugin Loaded");
      callbacks.issueAlert(String.format("Session ID being used : %s", SESSION_ID_KEY));


      /*
      Register the SessionHandler
       */
      callbacks.registerSessionHandlingAction(this);
   }

   @Override public String getActionName() {
      return "Read user session token from Macro";
   }

   /*
   This function is executed after macro call and before a subsequent Scanner or Intruder call.
    */
               @Override public void performAction(IHttpRequestResponse currentRequest,
IHttpRequestResponse[] macroItems) {
```

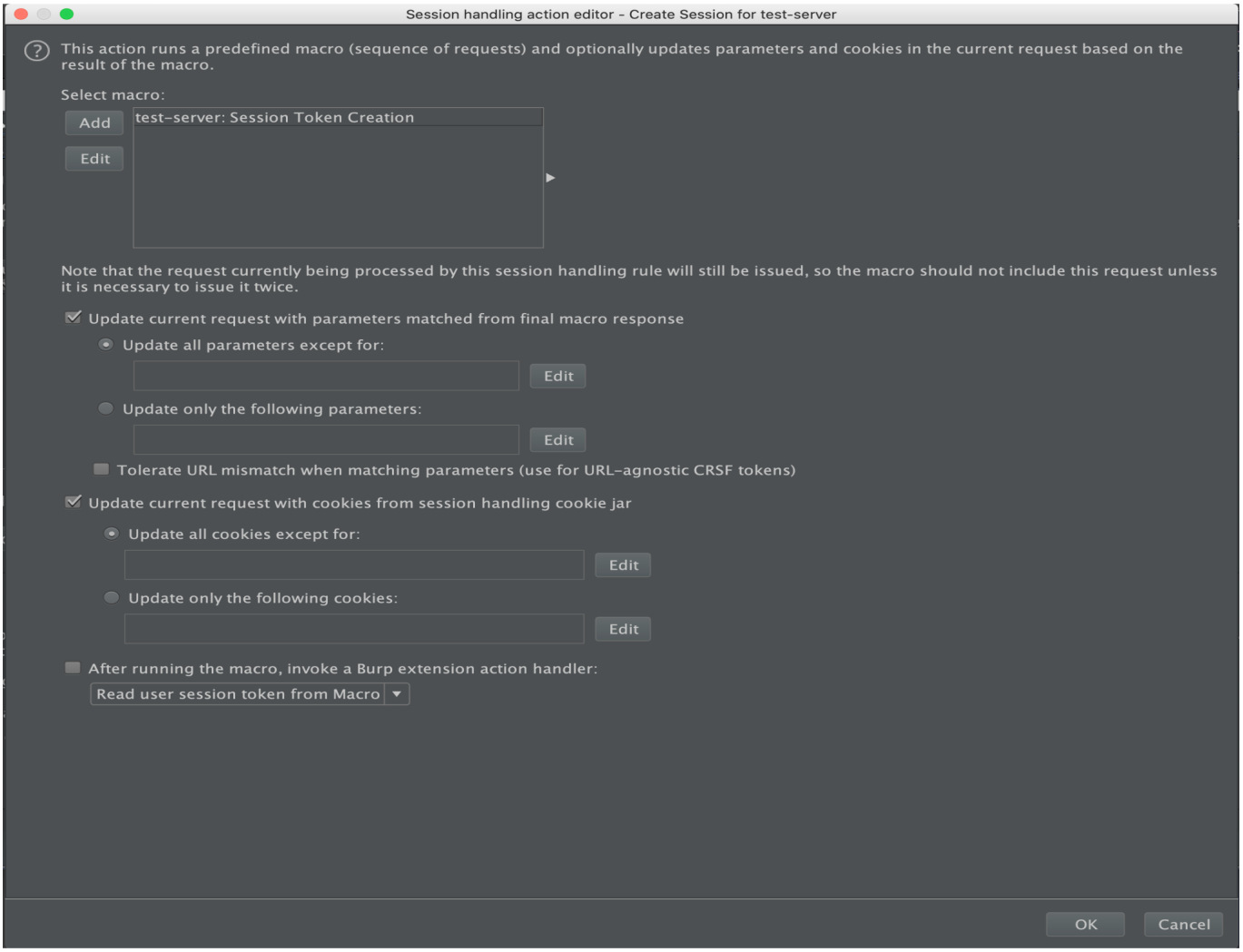
```
/*
Don't execute anything if there is no macro.
 */
if(macroItems.length == 0) return;

/*
Extract Macro response
 */
final byte[] macroResponse = macroItems[macroItems.length - 1].getResponse();

/*
Extract all headers from response
 */
final List<String> headers = helpers.analyzeResponse(macroResponse).getHeaders();

/*
Extract the Custom Session token header from all headers
 */
String sessionToken = null;
for(String header : headers){
    if(!header.startsWith(SESSION_ID_KEY)) continue;

    sessionToken = header.substring(SESSION_ID_KEY.length()).trim();
}

/*
If the session token is not identified, skip.
 */
if(sessionToken == null) return;

/*
Otherwise, append the session token to currentRequest
 */
final String req = helpers.bytesToString(currentRequest.getRequest());
final int sessionTokenKeyStart = helpers.indexOf(helpers.stringToBytes(req),
                                    SESSION_ID_KEY_BYTES,
                                    false,
                                    0,
                                    req.length());
final int sessionTokenKeyEnd = helpers.indexOf(helpers.stringToBytes(req),
                                   NEWLINE_BYTES,
                                   false,
                                   sessionTokenKeyStart,
                                   req.length());
```

```
        /*
        Join together First line + Session header line + rest of request
         */
        String newRequest = req.substring(0, sessionTokenKeyStart) +
                String.format("%s: %s", SESSION_ID_KEY, sessionToken) +
                req.substring(sessionTokenKeyStart + 1, sessionTokenKeyEnd);

        /*
        Update the current request headers
         */
        currentRequest.setRequest(helpers.stringToBytes(newRequest));
    }
}
```

Code above is self explanatory. Build it and load it in Burp.

Link Burp Macro and Session Handling Plugin together

Goto : Burp -> Project options -> Sessions -> Session Handling Rule -> Add Session handling rule



Now if you make any request through Scanner or Intruder, It will be routed through Macro and then through our Plugin, which will inject the session header on request.

Is this amazing ? Yes it is and don't forget to check everything by yourself.

Chapter 7: Plugin Example - HTTP Proxy JWT decoder

If you have made this far, this would have definitely enticed you about Burp Suite Extender development a bit. And to take this journey forward, Now we will be creating a very practical plugin which will decode the JWT token present in the request header under Proxy tab by creating a separate tab under proxy which will display the decoded JWT token present in the request. And guess what we can also design the UI for that component.

In this chapter we will be creating a proxy tab extender plugin which will decode the JWT token present in

the request header and output its decoded value on the go. To implement such functionality we would need to implement `IMessageEditorTabFactory` interface and register it like we normally do. But there is something special with this Interface, that It can generate a UI component for Burp Suite.

Important stuff to remember : `IMessageEditorTabFactory` implementation function will return an `IMessageEditorTab` instance which tells Burp what a new tab under Burp Proxy would look like.

Implement `IMessageEditorTab` Interface

- This will
    - Tell Burp what to do in the new tab.
    - be returned by the `IMessageEditorTabFactory`

This is how this Tab class would look in an abstract way. Stay with me, We will fill up the space, one by one

```
package burp;

import java.awt.*;

public class JWTDecodeTab implements IMessageEditorTab {

    private boolean editable;
    private ITextEditor txtInput;
    private byte[] currentMessage;
    private IBurpExtenderCallbacks callbacks;

    public JWTDecodeTab(IMessageEditorController controller, boolean editable) {
    }

    @Override public String getTabCaption() {
        return null;
    }

    @Override public Component getUiComponent() {
        return null;
    }

    @Override public boolean isEnabled(byte[] content, boolean isRequest) {
        return false;
    }

    @Override public void setMessage(byte[] content, boolean isRequest) {

    }
```

```
    @Override public byte[] getMessage() {
        return new byte[0];
    }

    @Override public boolean isModified() {
        return false;
    }

    @Override public byte[] getSelectedData() {
        return new byte[0];
    }
}
```

Fill up constructor

```
public JWTDecodeTab(IMessageEditorController controller, boolean editable,
IBurpExtenderCallbacks callbacks) {
        this.editable = editable;
        this.callbacks = callbacks;

        /*
         Create an instance of Burp's text editor, to display our JWT decode data.
         */
        txtInput = this.callbacks.createTextEditor();
        txtInput.setEditable(editable);
    }
```

Other functions

Explanations are inline with functions.

```
/*
    This will set the name for this tab under Proxy.
     */
    @Override public String getTabCaption() {
        return "JWT Decode";
    }
```

```
/*
    This will return the UI component to be display under Tab, Since we just want to display the text-editor
   (editable=false), we will return it. This has been created in Constructor.
    */
   @Override public Component getUiComponent() {
       return txtArea.getComponent();
   }

   /*
    This function will code the logic which will enable or disable this Tab. So since this is JWT decode tab plugin this should enable when there is a JWT token present in there.

    To keep this simple I am assuming that, JWT token is present as part of the parameter name `jwtToken`.
    This logic can be complex and process for all the parameters and can check if there is any field with data JWT deserializable or not.

   We may see this use case later.
    */
   @Override public boolean isEnabled(byte[] content, boolean isRequest) {
       return isRequest && null != helpers.getRequestParameter(content, "jwtToken");
   }
  @Override public byte[] getMessage() {
       return currentMessage;
   }

   @Override public boolean isModified() {
       return txtArea.isTextModified();
   }

   @Override public byte[] getSelectedData() {
       return txtArea.getSelectedText();
   }
```

Important decode function

Explanation Inline.

```
@Override public void setMessage(byte[] content, boolean isRequest) {
```

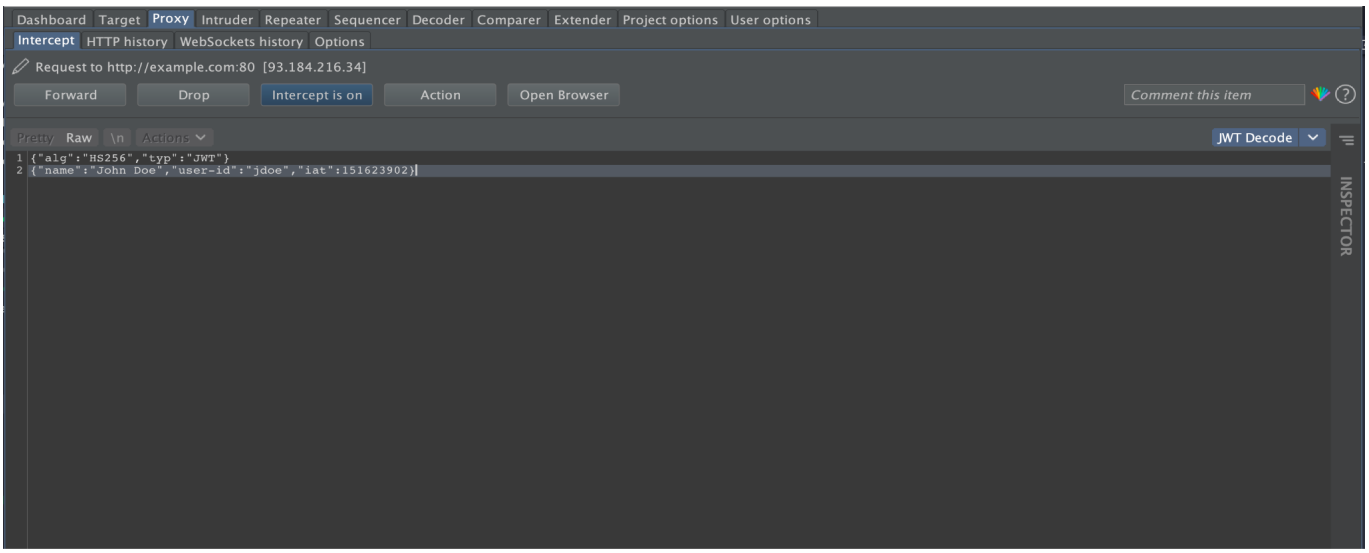
```
        /*
        If no data present in parameter
        */
        if (content == null) {
            /*
            clear our display, which is textArea
             */
            txtArea.setText(null);
            txtArea.setEditable(false);
        }
        else{
            /*
            Get the parameter value and decode it.
             */
            String jwtToken = helpers.getRequestParameter(content, "jwtToken").getValue();

              /*Since JWT token is <base64(alg)>.<base64(data)>.<base64(signature)> in simple terms, so we can use
            normal base64 decode functionality present in helpers to take this out data in plain text fmt.

            Steps :
                    - split on '.', '.'(Period) is regex in Java which points to all chars so make sure you use patterns.
                    https://stackoverflow.com/a/3481842
                - decode the first two parts as third part is just signature.
             */
            List<String> jwtTokenParts = Arrays.asList(jwtToken.split(Pattern.quote(".")));
            String decodedJwtToken =
                    helpers.bytesToString(helpers.base64Decode(jwtTokenParts.get(0))) +"\r\n" +
                    helpers.bytesToString(helpers.base64Decode(jwtTokenParts.get(1)));

            callbacks.issueAlert("Decoded JWT token " + decodedJwtToken);
            /*
            Set this data in the text field under the tab.
             */
            txtArea.setText(helpers.stringToBytes(decodedJwtToken));
            txtArea.setEditable(editable);
        }
        currentMessage = content;
    }
```

And last but not the least, the main BurpExtender class, to bring everything to life.

```
package burp;

public class BurpExtender implements IBurpExtender, IMessageEditorTabFactory{
    private IBurpExtenderCallbacks callbacks;

    @Override public void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks) {
        this.callbacks = callbacks;

        callbacks.setExtensionName("Proxy Tab - JWT decoder");
        callbacks.issueAlert("Plugin loaded");

        callbacks.registerMessageEditorTabFactory(this);
    }

        @Override public IMessageEditorTab createNewInstance(IMessageEditorController controller,
boolean editable) {
        return new JWTDecodeTab(controller, editable, callbacks);
    }
}
```

Test

- Send a Curl request with JWT token : `curl -XPOST -d "jwtToken=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiSm9obiBEb2UiLCJ1c2VyLWlkIjoiamRvZSIsImlhdCI6MTUxNjIzOTAyMn0.HQfUO7XHwp-Sx8oCQQBz90cGcvLI_43KdUNb4qzQ9Ag" http://example.com -x http://localhost:8080` through proxy.
- Check the plugin is decoding JWT Token.



Did you ever think that you would be able to bring any of such plugins to life on your own ? And just now we did it. Stare it for a moment, yes you only did it.

Chapter 8: Create a Separate tab plugin : JWT Encode/Decode

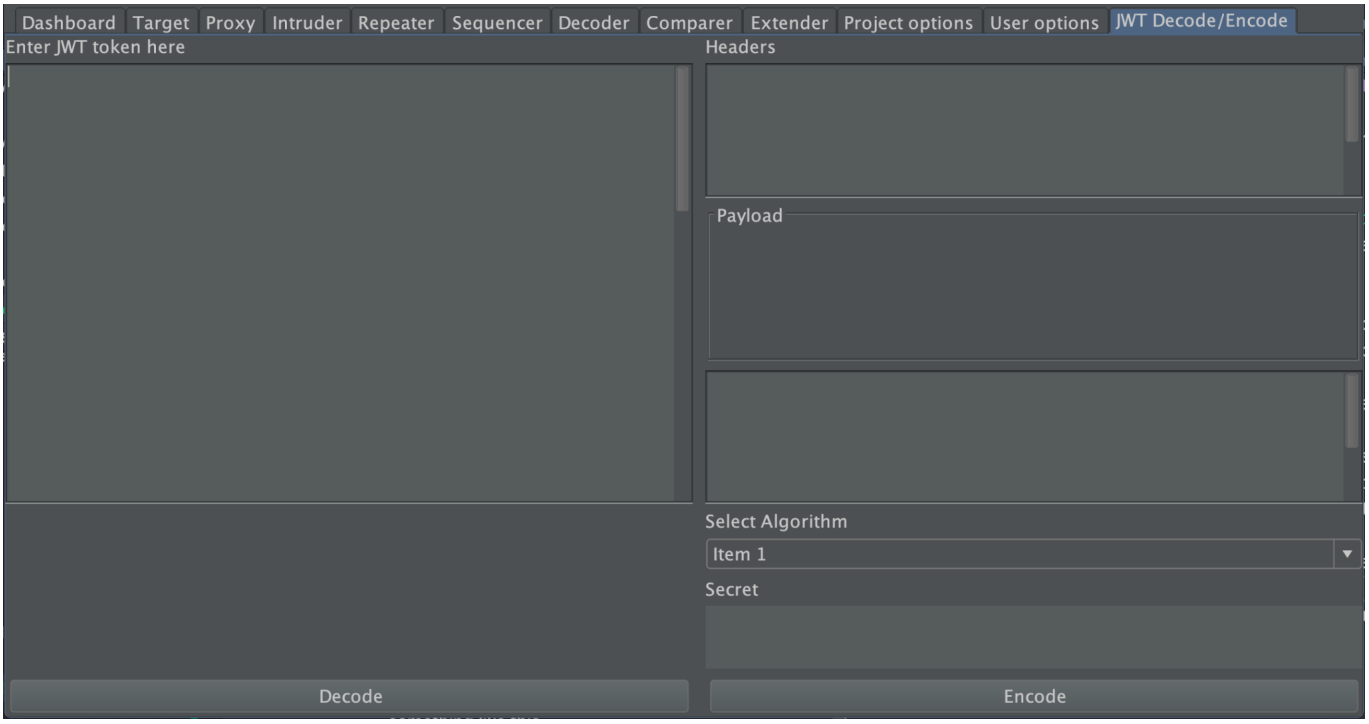
This chapter is going to be bit length and tricky so try to be with the flow, otherwise you have to read from start 😉 😝

This chapter needs some knowledge of **Java Swing Framework**, but if you don't have any knowledge, it won't be too difficult to deal with.

In this chapter we will be creating a separate tab plugin Yes, you will have a tab like shown below if you successfully complete this..

- This tab will show up in the Burp's Top tab menu like this.

- We will design(**not code**) the **UI** of this Tab.
    - But we need to encode the UI behavior though Swing Listener.
    - At the end we will have a plugin that looks for something like this.



Trust me this UI design won't be difficult at all if you are using any IDE like I am using intelliJ.

- This tab plugin will be very simple version of what we have in [jwt.io](jwt.io)

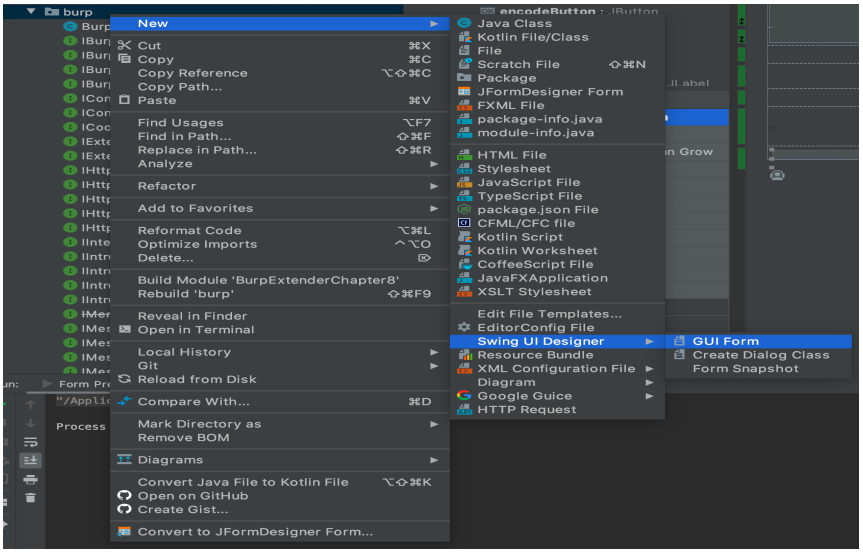
Creating a New tab in Burp

To create a new Tab which will be visible in the Burp toolbar, You need to implement : definitely `IBurpExtender` but `ITab` as well. **`ITab`** declaration looks below. ITab is simple, that you create a swing component, and return it through a function implementing **`getUiComponent()`** and that's it.

```
public interface ITab
{
    /**
     * Burp uses this method to obtain the caption that should appear on the
     * custom tab when it is displayed.
     *
     * @return The caption that should appear on the custom tab when it is
     * displayed.
     */
    String getTabCaption();

    /**
     * Burp uses this method to obtain the component that should be used as the
     * contents of the custom tab when it is displayed.
     *
     * @return The component that should be used as the contents of the custom
     * tab when it is displayed.
     */
    Component getUiComponent();
}
```

The content returned by getUiComponent() is what is rendered under the plugins tab window.

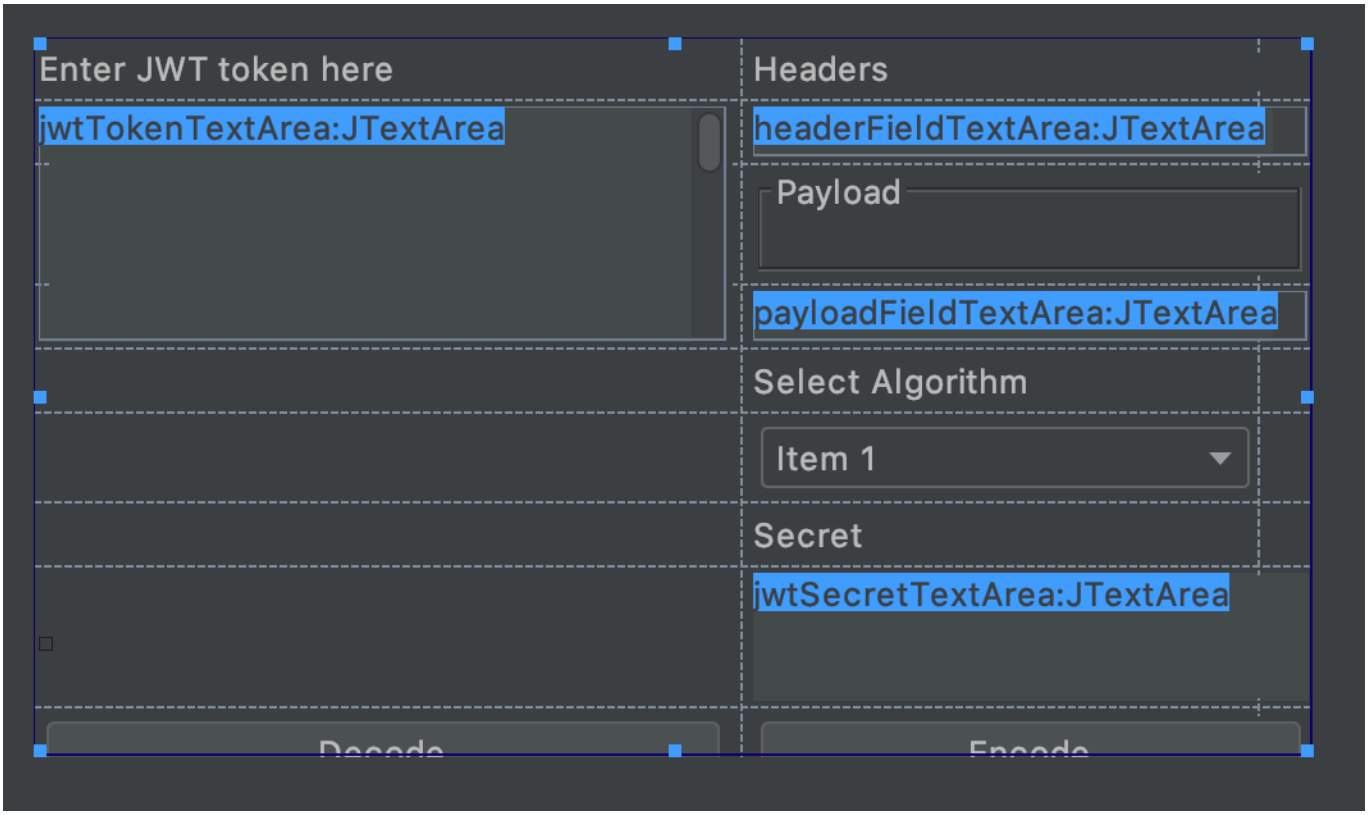
To create the UI for the tab I will be using, **IntelliJ's** in-built **form designer**, through this you can create UI very quickly with drag and drops and code only the event listeners.

Its available under : 

Once you select this, Two files will be created, one will be Form in UI and another will be its implementation at java class level. I have named those files as **`JWTDecodeTabForm.java`** & **`JWTDecodeTabForm.form`**. I have kept the class **`JWTDecodeTab`** separate which implements **`ITab interface`**.

Create UI for tab

- Select any layout of your choice for form.
- Create UI by drag and drop.
- I have created something like this, mimicking the UI from [jwt.io](jwt.io)



you may or may not need the external deps to be installed. I usually prefer deps installation through maven, but it's completely your choice. With IntelliJ and maven it's easy go to **File -> Project Structure -> Project Settings -> Libraries -> + -> From maven**. Download sources too, for intellij to provide autocomplete syntax.

- We will define the functionality of each of the buttons later.

Complete the Burp Extender ITab plugin

- Implement IBurpExtender like always and register Tab.

```
/*
BurpExtender.java
*/
package burp;

public class BurpExtender implements IBurpExtender {
    @Override public void registerExtenderCallbacks(IBurpExtenderCallbacks callbacks) {
        /*
            This is extension name, not the tab name. Tab name is picked up from ITab concrete class.
        */
        callbacks.setExtensionName("JWT Decode Extension");
        callbacks.addSuiteTab(new JWTDecodeTab(callbacks));
```

```
    }
}
```

Create a class implementing `ITab`, which we have called in `BurpExtender as JWTDecodeTab`.

```
/*
JWTDecodeTab.java
*/
package burp;

import java.awt.*;

public class JWTDecodeTab implements ITab {
  IBurpExtenderCallbacks callbacks;

  public JWTDecodeTab(IBurpExtenderCallbacks callbacks) {
    this.callbacks = callbacks;
  }

  @Override public String getTabCaption() {
    return "JWT Decode/Encode";
  }

  @Override public Component getUiComponent() {
    return new JWTDecodeTabForm(callbacks).getFrame();
  }
}
```

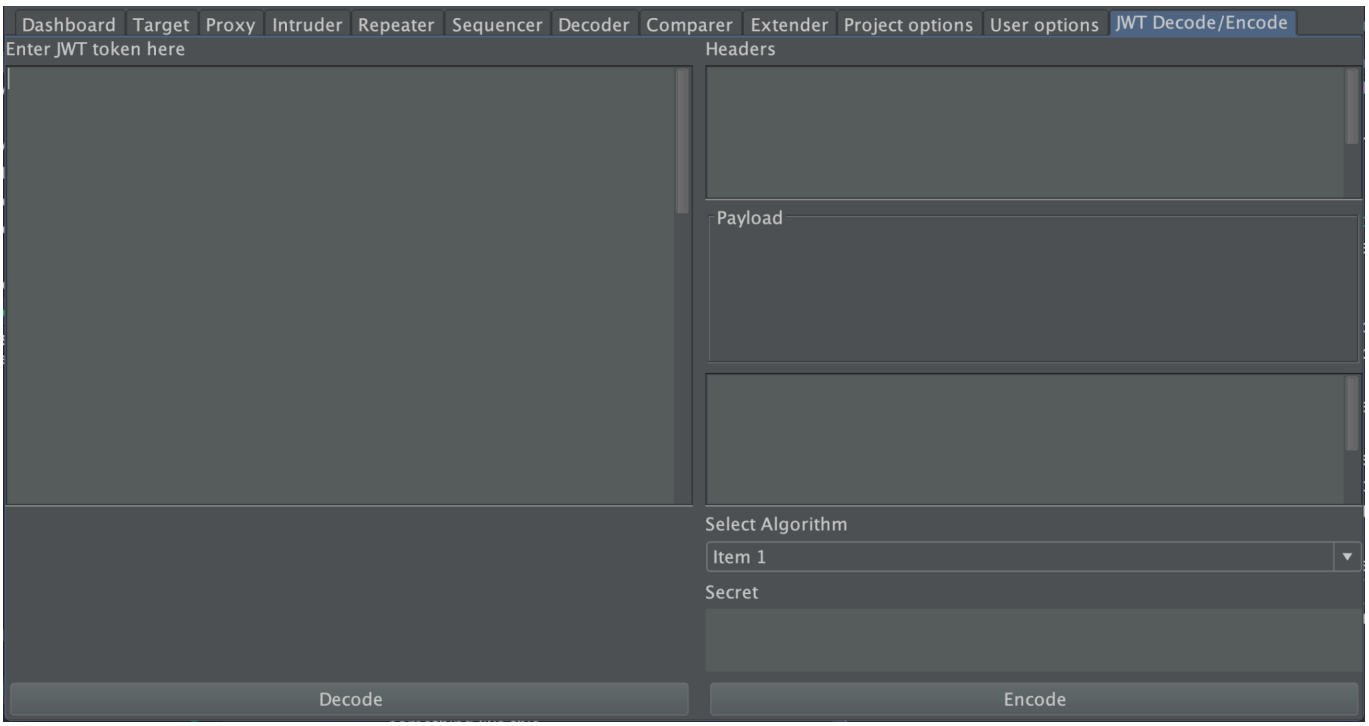
`JWTDecodeTabForm` is the class, which we designed Swing UI for through IntelliJ designer. On Bare minimum this class looks like.

```
public class JWTDecodeTabForm {
private JPanel basePanel;
private JTextArea jwtTokenTextArea;
private JTextArea headerFieldTextArea;
private JButton decodeButton;
private JButton encodeButton;
private JTextArea payloadFieldTextArea;
private JTextArea jwtSecretTextArea;
private JComboBox comboBox1;
```

```
private JLabel jwtValidationErrorLabel;

IBurpExtenderCallbacks callbacks;

public JWTDecodeTabForm(IBurpExtenderCallbacks callbacks) {
    this.callbacks = callbacks;
}

public Component getFrame(){
    return this.basePanel;
}
```

basePanel is the container component which contains all the other components, so I have returned this only which will display the UI for the tab.

Once you do all these steps and build artifact, upon loading that into Burp will result in something like this:



This UI though is not functional as no click events are registered at all for Decode & Encode buttons.

Giving life to Encode/Decode buttons

- Till now we have created a simple ui but the onclick events which are nothing but the lives of this plugin are not created.

- Right click on the decode button and select **Create listener -> Action Listener**. Then put in this code.

```
decodeButton.addActionListener(new ActionListener() {
      @Override public void actionPerformed(ActionEvent actionEvent) {
         String jwtToken = jwtTokenTextArea.getText().trim();
         try {
            List<String> jwtTokenParts = Arrays.asList(jwtToken.split(Pattern.quote(".")));


headerFieldTextArea.setText(helpers.bytesToString(helpers.base64Decode(jwtTokenParts.get(0))));

payloadFieldTextArea.setText(helpers.bytesToString(helpers.base64Decode(jwtTokenParts.get(1))));

         }catch (Exception e){
            /*
            For pupup, display error in PopUp
             */
                                    JOptionPane.showConfirmDialog(basePanel,   e.getMessage(),   "Error",
JOptionPane.OK_CANCEL_OPTION);
         }
      }
   });
```

- Do the same for Encode, but its implementation will be slightly different.

```
encodeButton.addActionListener(new ActionListener() {
      @Override public void actionPerformed(ActionEvent actionEvent) {
         try{
         Mac sha512Hmac;
         String secret = jwtSecretTextArea.getText();

         final byte[] byteKey = secret.getBytes(StandardCharsets.UTF_8);
         sha512Hmac = Mac.getInstance("HmacSHA512");
         SecretKeySpec keySpec = new SecretKeySpec(byteKey, "HmacSHA512");
         sha512Hmac.init(keySpec);

         String partialJwt = helpers.base64Encode(headerFieldTextArea.getText()) +
               "." +
               helpers.base64Encode(payloadFieldTextArea.getText());

         byte[] macData = sha512Hmac.doFinal(partialJwt.getBytes(StandardCharsets.UTF_8));

                                                   jwtTokenTextArea.setText(partialJwt     +     '.'     +
helpers.base64Encode(helpers.bytesToString(macData)));

         }catch (Exception e){
```

```
                    JOptionPane.showConfirmDialog(basePanel, e.getMessage(), "Error",
JOptionPane.OK_CANCEL_OPTION);
                }
            }
        });
```

There might be some issues in JWT decode and encode functionality but again the motive of this series is to teach you how to create extensions not how to create logic for extensions.

Rebuild the jar and load it again in Burp and see the magic that you have created. Believe in yourself, you only did it :)