

Integration of Interceptor with Postman Native App

Capstone Project Report

Submitted by

Vinit Shahdeo

15BIT0335

In partial fulfillment for the award of the degree of

B. TECH

In

INFORMATION TECHNOLOGY



VIT[®]

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING

JANUARY 2019

DECLARATION BY THE CANDIDATE

I hereby declare that the project report entitled “**Integration of Interceptor with Postman Native App**” submitted by me to School of Information Technology & Engineering, Vellore Institute of Technology University, Vellore in partial fulfillment of the requirement for the award of the degree of **B.Tech (Information Technology)** is a record of bonafide **Capstone Project** work carried out by me. I further declare that the work reported in this **project report** has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place: Vellore

Signature of the Candidate

Date:

Vinit Shahdeo



VIT[®]

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Information Technology & Engineering [SITE]

CERTIFICATE

This is to certify that the Capstone Project Report entitled “**PostDot Technologies Pvt. Limited**” submitted by **Vinit Shahdeo(15BIT0335)** to School of Information Technology & Engineering, Vellore Institute of Technology University, Vellore in partial fulfillment of the requirement for the award of the degree of **B.Tech (Information Technology)** is a record of bonafide **Capstone Project** work carried out by him/her in **PostDot Technologies Pvt. Limited, Bangalore**. The **Capstone** project fulfills the requirements as per the regulations of this Institute.

Examiner – Panel In-Charge

Date

(Name and Signature)

CERTIFICATE ISSUED BY THE ORGANIZATION



POSTMAN

The project report entitled “**Integration of Interceptor with Postman Native App**” is prepared and submitted by Vinit Shahdeo(15BIT0035), has been found satisfactory in terms of scope, quality and presentation as per the standards followed by the company.

The development process of the above mentioned project is being done under the supervision of Abhijit Kane, Co-Founder of PostDot Technologies Pvt. Limited.

Abhijit Kane

Co Founder

PostDot Technologies Pvt. Ltd.

ACKNOWLEDGEMENT

I would like to express my special thanks of gratitude to my project mentor **Abhijit Kane** as well as the **PostDot Technologies Pvt. Ltd.** who gave me the golden opportunity to do this project as Software Engineering Intern(Product Ops). This internship has also helped me in doing a lot of Research in Chrome APIs and I came to know about a lot of new tools used in Software world.

Also, I express my immense pleasure and deep sense of gratitude to **Prof. Laksmi Priya GG** (Guide) for giving me this opportunity and the guidance throughout the project.

And I would also like to thank my parents and co workers who helped me a lot in developing the project during internship period.

TABLE OF CONTENTS

CHAPTER NO.	TITLE
1	Abstract
2	Literature Survey
3	Hardware and Software requirements
4	Detailed Design
4.1	System Architecture
4.2	Flow Diagrams
4.3	Module description
4.4	Native App < > Interceptor Protocol
4.5	Screenshots
5	Implementation
7	Future Work
8	References

1. ABSTRACT

The Internship at **Postman** introduced me to various rules and ethics of the professional working. I had a wonderful opportunity to be a part of a great organization like PostDot Technologies Private Limited. The internship is very professional and being punctual is the biggest priority there whether it's the intern or other employees.

I have learnt JavaScript as I have to implement my project during the internship at Postman. **Postman is the only complete API development environment, used by nearly five million developers and more than 100,000 companies worldwide.** Postman is an elegant, flexible tool used to build connected software via APIs — quickly, easily and accurately. Postman is headquartered in San Francisco and has an office in Bangalore, where it was founded. Postman is privately held, with funding from Nexus Venture Partners. (Learn more at **<http://www.getpostman.com>**). The internship has gave me a thorough knowledge about the **APIs** and its various utilities. This internship also helped me to build some other qualities apart from gaining the technical knowledge.

Much importance was given on building skills like teamwork and being punctual throughout the internship. Mainly, Postman has three divisions, Product, Services and Platform. Some of the teams had the sole purpose of developing new feature, some were given the chance to improve the existing applications and software and continuously updating them, while some were in the think tank department whose sole purpose was to develop new ideas and also plan the entire software development process. I work in Product Operations Team as Software Engineering Intern. My responsibility is to add

new features in Interceptor so that an IPC bridge can be built between Native App and Chrome Interceptor. The objective of my work is to sync the browser cookies with the Postman Native App.

Besides the technical knowledge that I am learning many new tools which are used in Software World like **Git**, **GitHub**, **BitBucket**, **Jira** etc. This internship is also helping me to grow some personal skills like leadership and punctuality.



2. Literature Survey

The **Interceptor** is a Chrome *extension* (different from chrome app) that users install through the Chrome app store. The interceptor:

1. has access to all of Chrome's cookies. These cookies are included in XHR calls made from the Interceptor.
2. receives events when any tab in Chrome is about to make a request (with the request details)
3. can inject headers into the request AFTER the `xhr.send()` method is called, allowing addition of headers not supported by the XHR API.
4. Communicate with Chrome apps (fixed Chrome app ID) over simple Chrome-specific IPC.

We use these features of the interceptor to provide the following features in the Postman Chrome app:

1. Capture requests from the browser. The user can enable 'capture' in the interceptor UI. Any requests made in the browser UI will be sent to the open Postman Chrome app.
2. Forwarding requests from the app to the interceptor. Instead of the XHR call being made from the app (where the browser's session is absent), the app forwards the request object to the interceptor, which makes the XHR call on its behalf. This means that all browser session info (including auth) is added to the request, and users can make authenticated requests to websites they've logged in to in the browser.

Native App

Since the native app isn't restricted by XHR, it can modify the outgoing request as required. Cookies etc. can be managed purely in the app. The native app's proxy lets the user browse in the browser and get calls auto-synced to the app. However:

1. Cookies need to be added to the native manually, or received as part of a response. There's no way to use your browser's existing session to make requests from the native app.
2. The proxy does not support websites with HSTS. With HSTS enabled, browsers require that the certificate host is the same as the host being requested.

We need to define a protocol for the interceptor and **nativeServer** to communicate. There are a variety of messages that can pass through. For example:

1. Interceptor wants to send a new captured request to the app
2. Interceptor wants to send cookies for a requested domain
3. Interceptor wants to send a request's response to the app
4. The app wants to request cookies for a domain
5. The app wants to start request capture
6. The app wants to forward a request through the interceptor

Chrome Native Messaging

Extensions and apps can exchange messages with native applications using an API that is similar to the other *message passing APIs*. Native applications that support this feature must register a native messaging host that knows how to communicate with the extension. Chrome starts the host in a separate process and communicates with it using standard input and standard output streams.

Native Messaging Protocol

Chrome starts each native messaging host in a separate process and communicates with it using standard input (stdin) and standard output (stdout). The same format is used to send messages in both directions: each message is serialized using JSON, UTF-8 encoded and is preceded with 32-bit message length in native byte order. The maximum size of a single message from the native messaging host is 1 MB, mainly to protect Chrome from misbehaving native applications. The maximum size of the message sent to the native messaging host is 4 GB.

The first argument to the native messaging host is the origin of the caller, usually `chrome-extension://[ID of allowed extension]`. This allows native messaging hosts to identify the source of the message when multiple extensions are specified in the `allowed_origins` key in the native messaging host manifest.

Warning: In Windows, in Chrome 54 and earlier, the origin was passed as the second parameter instead of the first parameter.

When a messaging port is created using `runtime.connectNative` Chrome starts

native messaging host process and keeps it running until the port is destroyed. On the other hand, when a message is sent using *runtime.sendNativeMessage*, without creating a messaging port, Chrome starts a new native messaging host process for each message. In that case the first message generated by the host process is handled as a response to the original request, i.e. Chrome will pass it to the response callback specified when *runtime.sendNativeMessage* is called. All other messages generated by the native messaging host in that case are ignored.

On Windows, the native messaging host is also passed a command line argument with a handle to the calling chrome native window: **--parent-window=<decimal handle value>**. This lets the native messaging host create native UI windows that are correctly focused.

Connecting to Native Messaging Host

Sending and receiving messages to and from a native application is very similar to cross-extension messaging. The main difference is that *runtime.connectNative* is used instead of *runtime.connect*, and *runtime.sendNativeMessage* is used instead of *runtime.sendMessage*. These methods can only be used if the "nativeMessaging" permission is declared in your extension's manifest file.

Cookies

An **HTTP** cookie (web cookie, browser cookie) is a small piece of data that a server sends to the user's web browser. The browser may store it and send it back with the next request to the same server. Typically, it's used to tell if two requests

came from the same browser — keeping a user logged-in, for example. It remembers stateful information for the stateless HTTP protocol.

Cookies are mainly used for three purposes:

1. **Session management** - Logins, shopping carts, game scores, or anything else the server should remember
2. **Personalization** - User preferences, themes, and other settings
3. **Tracking** - Recording and analyzing user behavior

Cookies were once used for general client-side storage. While this was legitimate when they were the only way to store data on the client, it is recommended nowadays to prefer modern storage APIs. Cookies are sent with every request, so they can worsen performance (especially for mobile data connections). Modern APIs for client storage are the *Web storage API* (localStorage and sessionStorage) and IndexedDB.

XMLHttpRequest (XHR)

XMLHttpRequest is a built-in browser object that allows to make HTTP requests in JavaScript. Despite of having the word “XML” in its name, it can operate on any data, not only in XML format.

XMLHttpRequest has two modes of operation: synchronous and asynchronous. XMLHttpRequest (XHR) is a browser-level API that enables the client to script

data transfers via JavaScript. XHR made its first debut in Internet Explorer 5, became one of the key technologies behind the Asynchronous JavaScript and XML (AJAX) revolution, and is now a fundamental building block of nearly every modern web application.

Prior to XHR, the web page had to be refreshed to send or fetch any state updates between the client and server. With XHR, this workflow could be done asynchronously and under full control of the application JavaScript code. XHR is what enabled us to make the leap from building pages to building interactive web applications in the browser.

However, the power of XHR is not only that it enabled asynchronous communication within the browser, but also that it made it simple. XHR is an application API provided by the browser, which is to say that the browser automatically takes care of all the low-level connection management, protocol negotiation, formatting of HTTP requests, and much more:

- The browser manages connection establishment, pooling, and termination.
- The browser determines the best HTTP(S) transport (HTTP/1.0, 1.1, 2).
- The browser handles HTTP caching, redirects, and content-type negotiation.
- The browser enforces security, authentication, and privacy constraints.

Free from worrying about all the low-level details, our applications can focus on the business logic of initiating requests, managing their progress, and processing returned data from the server. The combination of a simple API and its ubiquitous availability across all the browsers makes XHR the “Swiss Army knife” of networking in the browser.

As a result, nearly every networking use case (scripted downloads, uploads, streaming, and even real-time notifications) can and have been built on top of XHR. Of course, this doesn’t mean that XHR is the most efficient transport in each case—in fact, as we will see, far from it—but it is nonetheless often used as a fallback transport for older clients, which may not have access to newer browser networking APIs. With that in mind, let’s take a closer look at the latest capabilities of XHR, its use cases, and performance do’s and don’ts.

An exhaustive analysis of the full XHR API and its capabilities is outside the scope of our discussion—our focus is on performance! Refer to the official W3C standard for an overview of the **XMLHttpRequest** API.

Brief History of XHR

Despite its name, XHR was never intended to be tied to XML specifically. The XML prefix is a vestige of a decision to ship the first version of what became known as XHR as part of the MSXML library in Internet Explorer 5:

Mozilla modeled its own implementation of XHR against Microsoft’s and exposed it via the XMLHttpRequest interface. Safari, Opera, and other browsers

followed, and XHR became a de facto standard in all major browsers—hence the name and why it stuck. In fact, the official W3C Working Draft specification for XHR was only published in 2006, well after XHR came into widespread use!

However, despite its popularity and key role in the AJAX revolution, the early versions of XHR provided limited capabilities: text-based-only data transfers, restricted support for handling uploads, and inability to handle cross-domain requests. To address these shortcomings, the “XMLHttpRequest Level 2” draft was published in 2008, which added a number of new features:

- Support for request timeouts
- Support for binary and text-based data transfers
- Support for application override of media type and encoding of responses
- Support for monitoring progress events of each request
- Support for efficient file uploads
- Support for safe cross-origin requests

In 2011, “XMLHttpRequest Level 2” specification was merged with the original XMLHttpRequest working draft. Hence, while you will often find references to XHR version or level 1 and 2, these distinctions are no longer relevant; today, there is only one, unified XHR specification. In fact, all the new XHR2 features and capabilities are offered via the same XMLHttpRequest API: same interface, more features.

XHR Use Cases and Performance

XMLHttpRequest is what enabled us to make the leap from building pages to building interactive web applications in the browser. First, it enabled asynchronous communication within the browser, but just as importantly, it also made the process simple. Dispatching and controlling a scripted HTTP request takes just a few lines of JavaScript code, and the browser handles all the rest:

- Browser formats the HTTP request and parses the response.
- Browser enforces relevant security (same-origin) policies.
- Browser handles content negotiation (e.g., gzip).
- Browser handles request and response caching.
- Browser handles authentication, redirects, and more...

As such, XHR is a versatile and a high-performance transport for any transfers that follow the HTTP request-response cycle. Need to fetch a resource that requires authentication, should be compressed while in transfer, and should be cached for future lookups? The browser takes care of all of this and more, allowing us to focus on the application logic!

However, XHR also has its limitations. As we saw, streaming has never been an official use case in the XHR standard, and the support is limited: streaming with XHR is neither efficient nor convenient. Different browsers have different behaviors, and efficient binary streaming is impossible. In short, XHR is not a

good fit for streaming.

Similarly, there is no one best strategy for delivering real-time updates with XHR. Periodic polling incurs high overhead and message latency delays. Long-polling delivers low latency but still has the same per-message overhead; each message is its own HTTP request. To have both low latency and low overhead, we need XHR streaming!

As a result, while XHR is a popular mechanism for “real-time” delivery, it may not be the best-performing transport for the job. Modern browsers support both simpler and more efficient options, such as Server-Sent Events and WebSocket. Hence, unless you have a specific reason why XHR polling is required, use them.

XHR enables a simple and efficient way to synchronize client updates with the server: whenever necessary, an XHR request is dispatched by the client to update the appropriate data on the server. However, the same problem, but in reverse, is much harder. If data is updated on the server, how does the server notify the client? HTTP does not provide any way for the server to initiate a new connection to the client. As a result, to receive real-time notifications, the client must either poll the server for updates or leverage a streaming transport to allow the server to push new notifications as they become available. Unfortunately, as we saw in the preceding section, support for XHR streaming is limited, which leaves us with XHR polling.

Few useful links

1. Postman Interceptor - GitHub Repository

(<https://github.com/postmanlabs/postman-chrome-interceptor>)

2. Postman Chrome App - Link to download

(<https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcdcbncdddomop?hl=en>)

3. Postman Native App - Link to download

(<https://www.getpostman.com/downloads/>)

4. Postman Official Website (<https://www.getpostman.com/>)

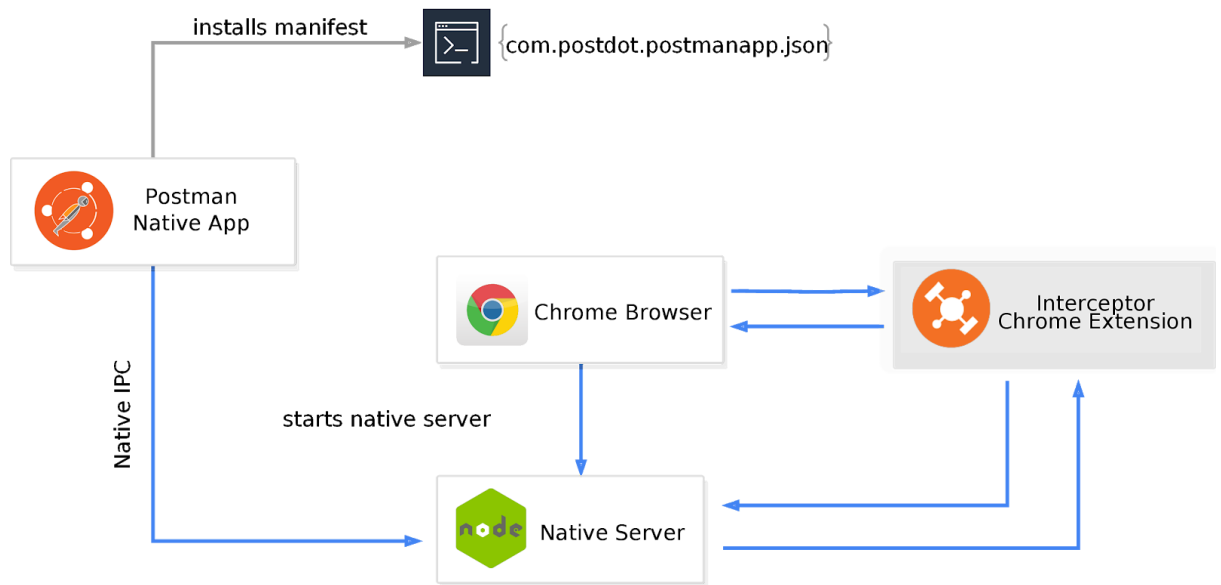
5. Postman Official Blog (<http://blog.getpostman.com/>)

3. Hardware and Software Requirements

Serial No.	Title
1.	JavaScript
2.	Git & GitHub
3.	BitBucket
4.	Postman Native App
5.	Postman Chrome App
6.	Interceptor
7.	Jira

4. System Architecture

Integration of Interceptor with Native App



Overview

The **Interceptor** is a Chrome extension (different from an app) that users install through the Chrome app store.

The interceptor:

1. has access to all of Chrome's cookies. These cookies are included in XHR calls made from the Interceptor.
2. receives events when any tab in Chrome is about to make a request (with the request details)

3. can inject headers into the request AFTER the `xhr.send()` method is called, allowing addition of headers not supported by the XHR API.
4. Communicate with Chrome apps (fixed Chrome app ID) over simple Chrome-specific IPC.

We use these features of the interceptor to provide the following features in the Postman Chrome app:

1. **Capture requests from the browser** - The user can enable 'capture' in the interceptor UI. Any requests made in the browser UI will be sent to the open Postman Chrome app
2. **Forwarding requests from the app to the interceptor** - Instead of the XHR call being made from the app (where the browser's session is absent), the app forwards the request object to the interceptor, which makes the XHR call on its behalf. This means that all browser session info (including auth) is added to the request, and users can make authenticated requests to websites they've logged in to in the browser.

Current Interceptor Flow

The Interceptor communicates with the Chrome app through a Chrome-provided async API. Either the app or the interceptor can initiate communication.

There are two modes the interceptor operates in:

- 1. *Request capture mode:*** Occurs only when the request capture is toggled on in the interceptor UI. Any requests made in the browser (page navigation / XHR) are captured by the interceptor and sent over the IPC bridge to the app (if it matches the URL filter provided). If the app is open with the Interceptor setting ON, the request flows into the history section.
- 2. *Forwarding mode:*** The app initiates this flow by sending a request object to the interceptor. The object contains a *postmanMessageId* (unique to each request sent), and a *fromAppId* (which is the ID of the chrome app - fhb...). The interceptor then fires off the request, with the callback handler in the closure. The callback handler needs to maintain the *postmanMessageId* to return to the app (so that the app may show the response in the correct tab). This ensures that concurrent multiple requests can be sent from the app/runner from different tabs. The Interceptor uses XHR for the request sending/callbacks, but also has access to certain events that Chrome fires (*onBeforeHeaders* etc.) that let it modify the request after the XHR part is done.

Native App

Since the native app isn't restricted by XHR, it can modify the outgoing request as required. Cookies etc. can be managed purely in the app. The native app's proxy lets the user browse in the browser and get calls auto-synced to the app.

However:

1. Cookies need to be added to the native manually, or received as part of a response. There's no way to use your browser's existing session to make requests from the native app.
2. The proxy does not support websites with HSTS. With HSTS enabled, browsers require that the certificate host is the same as the host being requested.

Native App < > Native Server < > Interceptor

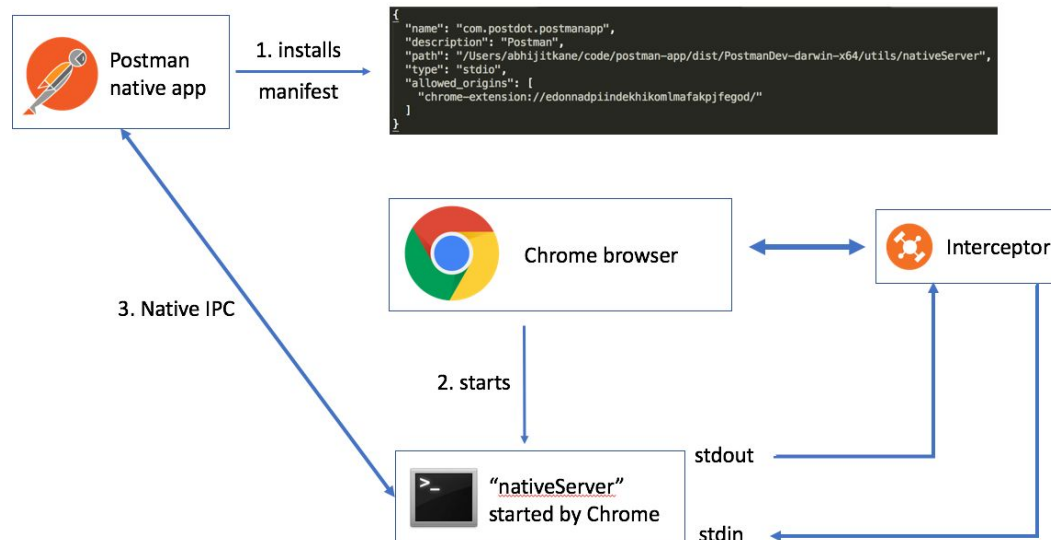


Proof of Concept

The POC achieves:

1. Installation of the nativeServer and manifest from the app.
2. The interceptor being able to capture requests and forward them to the nativeServer
3. nativeServer starting an IPC link with the app, once the app starts

Native Messaging



5. IMPLEMENTATION

Assumptions

1. Interceptor provides a protocol to communicate with native app via IPC
2. The native app sends a JSON payload to the chrome extension via Native Server.
3. Assuming JSON object contains two fields `url` : string and `sync` : boolean.
4. *url* is basically the regex for the domain to filter the cookies and sync it with the app.

```
{  
    url : "https://getpostman.com"  
    sync : "true"  
}
```

Note :

- If the app sends regex only (say `*.getpostman.com`) then we need to figure out a way to filter all the cookies belonging to the specified domain regex criteria.
- Filtering cookies based on regular expressions can be done by storing all the cookies in a temporary array say `allCookies` and applying the filter on domain of all the cookies stored in the array.

- All the cookies stored in the browser can be fetched through **chrome.cookies** API.

```
chrome.cookies.getAll({url : domain},function(cookiesData){
    forEach(cookie in cookiesData){
        allCookies.push(cookie);
    }
});
```

- Later, we can apply the filter on **allCookies** array based on the domain field of all the cookies. Filtering can be done using basic string manipulation.

End Goal

1. Interceptor send cookies for the requested domain
2. If the app and interceptor is in sync (sync = true) then anyone can update the cookies for the specified domain.

Working

- The interceptor receives a regex url and it stores it into the localStorage of the browser. Storing the url is needed for remembering the preferred sync URL by native app.

- Once it receives the request url, the interceptor sends the JSON object containing the cookies for the given URL. The transmission is done via NativeServer.

```
domain:  "getpostman.com"
expirationDate:  1548854858.7152
hostOnly:  true
httpOnly:  true
name:  "XXXXXXX"
path:  "/"
sameSite:  "strict"
secure:  true
session:  false
storeId:  "0"
value:  "XXXXXXXXXXXXX"
```

- The native app uses tough-cookie npm module to convert this JSON* string object into Cookie Object. (**Only required parameters are sent.*)
- If the cookies for the domain(url) in sync(i.e. stored in the **localStorage** of browser) are set/updated, it will call an event listener and sends the updated JSON payload to the native app.

chrome.cookies API has provided

chrome.cookies.onChange.addListener(function callback)

So, an event can be fired which will update the cookies in the native app.

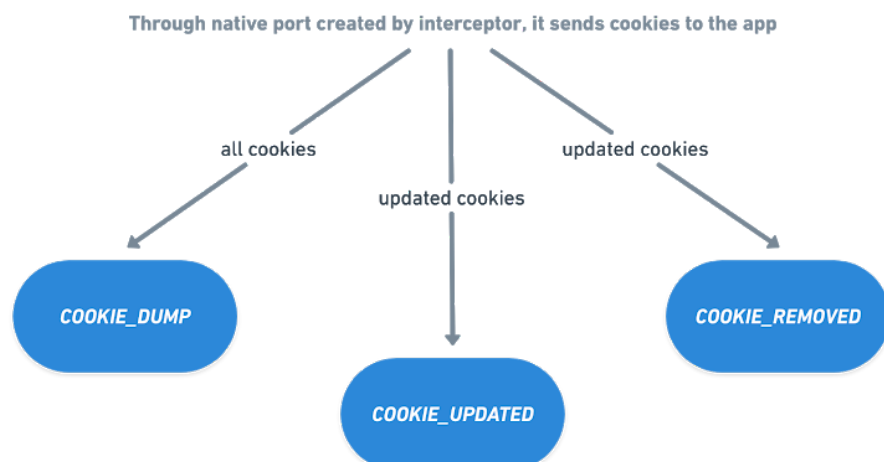
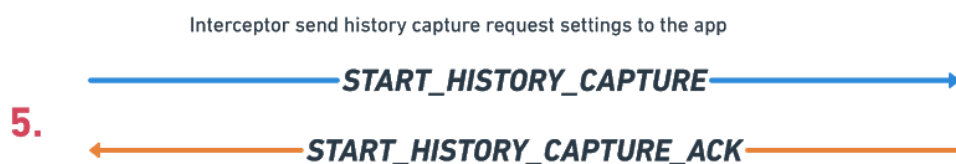
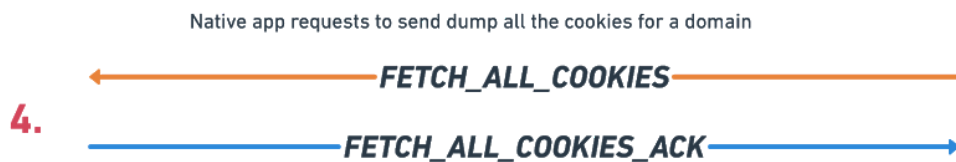
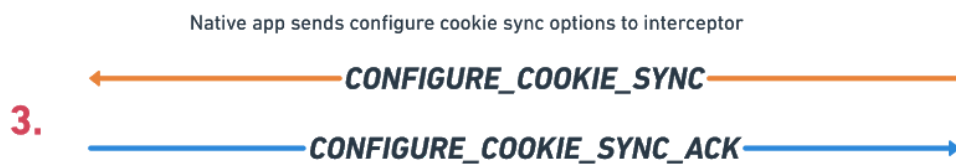
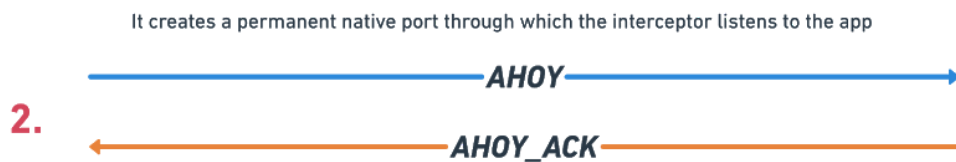
- If the cookies (for the domain in sync) are set/updated in the native app, it will send a JSON payload containing the cookies data to the interceptor.
- Interceptor will set the cookies in the browser for the domain in sync using `chrome.cookies.set(obj, callback)` API

I have also published a collection as my first week task. It is published as template.

Link - <https://documenter.getpostman.com/view/6186519/RznEKdvc>

Interceptor

Native App



Protocol used between Native App <> Interceptor

- Once the chrome is opened, Interceptor sends `msg : { type : 'HELO' }` to the native server that lies with the native app. (Note : `chrome.runtime.sendMessage` is used here to send the message without creating the port)
- If the nativeServer(app) replies with `msg : { type : 'HELO_ACK' }` , it means the Interceptor is able to ping the nativeServer.
- After this Interceptor creates a native port (through which the communication between app and Interceptor takes place) using `chrome.runtime.connectNative` API and it pings nativeServer again with `msg : { type : 'AHOY' }`
- If the Interceptor receives `msg : { type : 'AHOY_ACK' }` means the port is ready to listen. Here, the actual handshake is done.

Request (From Interceptor)	Response (By Native App)
<code>{ type : 'HELO' }</code>	<code>{ type : 'HELO_ACK' }</code>
<code>{ type : 'AHOY' }</code>	<code>{ type : 'AHOY_ACK' }</code>

App sends `CONFIGURE_COOKIE_SYNC` command to Interceptor.

```
{
  type : 'CONFIGURE_COOKIE_SYNC'
  postmanMessage : {
    domain : 'getpostman.com',
    enabled : true
  }
}
```

Interceptor acknowledges this message by sending the following JSON

payload :

CASE 1 : If `enabled == true` , Interceptor sends the same JSON payload with `type = CONFIGURE_COOKIE_SYNC_ACK` to acknowledge the cookie sync request received.

```
{
  type : 'CONFIGURE_COOKIE_SYNC_ACK',
  postmanMessage : {
    domain : 'getpostman.com',
    enabled : true
  }
}
```

After this, it does the following :

- It attaches a `chrome.cookies.onChange` listener.
- It updates the app cookie sync options(i.e. `syncDomain` & `syncEnabled`) in the browser's local storage using `localStorage` API of chrome.
- Initially, it sends all cookies present in the browser for the given domain.
- Later, it keep sending the updates (i.e. cookies that are being updated/added/removed)

All cookies	updated/added cookies	Removed cookies
<pre>{ type: COOKIE_DUMP message : [...] }</pre>	<pre>{ type: COOKIE_UPDATED message : [...] }</pre>	<pre>{ type: COOKIE_REMOVED message : [...] }</pre>

CASE 2 : If `enabled == false` , Interceptor sends the following message to acknowledge the cookie sync request received.

```
{
  type : 'CONFIGURE_COOKIE_SYNC_ACK',
  postmanMessage : {
    domain : 'getpostman.com',
    enabled : false
  }
}
```

- It removes the `chrome.cookies.onChanged` listener.
- It updates the values (i.e. `syncEnabled : false`) in chrome's local storage.

App can also send the following message :

```
{
  type : 'FETCH_ALL_COOKIES',
  postmanMessage : {
    domain : 'getpostman.com'
  }
}
```

The interceptor will respond with the following JSON payload to acknowledge it :

```

{
  type : 'FETCH_ALL_COOKIES_ACK',
  postmanMessage : {
    domain : 'getpostman.com'
  }
}

```

Later, Interceptor will respond by sending all the cookies for the received domain.

```

{
  type: COOKIE_DUMP
  message : [...]
}

```

(Note - It only sends list of cookies. It doesn't attaches any listener for chrome.cookies.onChange)

Interceptor sends the following message to the app when the request capture is toggled ON :

```

{
  type : 'CONFIGURE_HISTORY_CAPTURE'
  postmanMessage : {
    enabled : true
  }
}

```

The app sends the following response to the interceptor :

```

{
  type : 'CONFIGURE_HISTORY_CAPTURE_ACK'
}

```

```
    postmanMessage : {  
      enabled : true  
    }  
  }  
}
```

And the captured request in the browser starts being flushed into the history section of the native app.

If the history capture toggle is made 'OFF' through Interceptor UI, it sends the following message :

```
{  
  type : 'CONFIGURE_HISTORY_CAPTURE'  
  postmanMessage : {  
    enabled : false  
  }  
}
```

The app sends the following response to the interceptor :

```
{  
  type : 'CONFIGURE_HISTORY_CAPTURE_ACK'  
  postmanMessage : {  
    enabled : true  
  }  
}
```

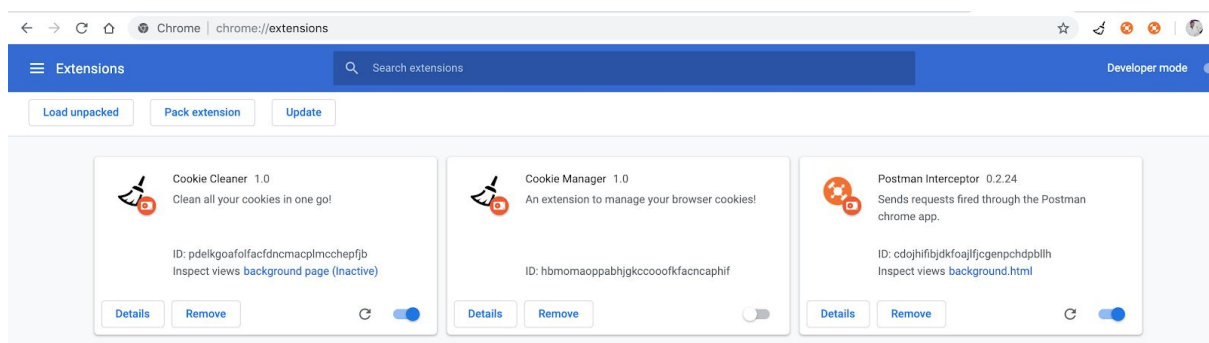
Interceptor Integration with Native App

[Interceptor ~ Source Code : Clone this repository and checkout feature/cookie-sync branch]

[Postman App ~ Source Code : Clone this repository and checkout feature/native-poc branch]

[Interceptor] How to install?

1. Unzip the postman-interceptor folder.
2. Open chrome and go to **chrome://extensions/**
3. Check the developer mode on the extreme right corner
4. Click load unpacked and navigate to postman-interceptor/extension folder
5. Here you go!



[Postman App] How to install?

1. Unzip the app folder
2. Open terminal and go to the following location :
~/Library/Application Support/Google/Chrome/NativeMessagingHosts/

3. Create a file here `com.postdot.postmannativeapp.json` (Don't create if it already exists)
4. This file is basically a manifest which registers a native messaging host and defines the native messaging host configuration.
5. This file should contain :

```
{
  "name": "com.postdot.postmannativeapp",
  "description": "Postman",
  "path":
    "/Users/vinitshahdeo/Desktop/postman-work/postman-app-develop/postman-app/node_modules/electron/dist/Electron.app/Contents/Resources/app/utils/nativeServer",
  "type": "stdio",
  "allowed_origins": [
    "chrome-extension://njnmpdehcbmgidmlcflijogkhpbbilhi/"
  ]
}
```

Note :

- Kindly ensure that the “path” in the above manifest should point to the absolute location of nativeServer file inside the app folder.
- The nativeServer file should be an executable
- Verify the extension ID mentioned in the manifest.json i.e
"chrome-extension://njnmpdehcbmgidmlcflijogkhpbbilhi/"

6. Go to the app folder and navigate to `src/main/utils/nativeServer` and replace the location in the following function (Line no. 12)

```
function flog(msg) {  
  
fs.appendFileSync('/Users/vinitshahdeo/desktop/yeslog.txt',  
msg + "\n\n");  
}
```

How to run the app?

1. Open terminal and navigate to app folder.
2. Install all the dependencies using **npm install**
3. Run **npm run start** and wait for while.
4. Once the build process is over, open a new tab in terminal and run **npm run open**
5. And here you!

To begin Cookie Sync

Open the renderer console and paste the below command and press enter and the cookie sync for the specified domain(i.e. facebook.com) will be started :

```
pm.appWindow.sendToElectron({ event:  
'forwardInterceptorRequest', message: {type :  
'CONFIGURE_COOKIE_SYNC', message : {domain :  
'facebook.com', enabled : true}}})
```

Now, you should be able to view the cookies in native app's cookie store. Refresh the website(facebook.com) in the browser and you can observe the cookies are being updated in cookie store of the app.

Instructions for Installation in Windows

Click [here](#) to download Postman interceptor.

Click [here](#) to download Postman App.

[Interceptor] How to install?

1. Unzip the postman-interceptor folder.
2. Open chrome and go to **chrome://extensions/**
3. Check the developer mode on the extreme right corner
4. Click load unpacked and navigate to **postman-interceptor/extension** folder
5. Here you go!

[Native Server] How to install?

1. Download and Unzip this folder.
2. Run `install_host.bat`
3. Open `manifest.json` and edit the chrome extension ID with the Postman Interceptor ID.

```
{  
  "name": "com.postdot.postmannativeapp",  
  "description": "Postman",  
  "path": "nativeserver.exe",  
  "type": "stdio",
```

```
"allowed_origins": [  
  
  "chrome-extension://njnmpdehcbmgidmlcflijogkhpbbilhi/"  
]  
}
```

4. And you're all done.

How to run the app?

1. Unzip this folder and open the Postman app i.e. PostmanBeta.exe
2. Open the renderer console by either by pressing ctrl+shift+i or go to View -> Developer -> Show DevTools(Current View) and paste the below command :

```
pm.settings.setSetting('useInterceptor',true)
```

3. Now, past the following command and press enter :

```
pm.appWindow.sendToElectron({ event:  
'forwardInterceptorRequest', message: {type :  
'CONFIGURE_COOKIE_SYNC', message : {domain : 'google.com',  
enabled : true}}})
```

4. And here you!

Now, you should be able to view the cookies in native app's cookie store. Refresh the website(google.com) in the browser and you can observe the cookies are being updated in cookie store of the app.

Screenshots

The live demo is available at
<https://github.com/vinitshahdeo/Cookie-Manager>

Cookie Manager

Total Cookies **1575**

Please enter the **url**, **name** & **value** pair and click **Set Cookies** button. ×

Enter Domain:

Name:

Value:

Clear All Cookies

Display Cookies

Set Cookies

Cookie Manager

Total Cookies **1571**

Invalid URL! Hint : Please enter complete url
including **http://** or **https://** below and press



Display Cookies


Enter Domain:











Clear All Cookies

Display Cookies

Set Cookies



Search Google or type a URL 

-  Gmail
-  vinitshahdeo (V...
-  vinit.shahdeo@...
-  LinkedIn
-  Quora
-  Bitbucket
-  Overview — Bit...
-  GitHub
-  Facebook
-  Add shortcut

Cookie Manager

Total Cookies **1572**

Enter Domain:

Clear All Cookies

Display Cookies

Set Cookies





Postman

[Report abuse](#)<https://www.getpostman.com/> help@getpostman.com Verified[Repositories 62](#) [People 15](#) [Projects 0](#)

Pinned repositories

postman-app-support

Forked from jeremys/Simple-Rest-Client-Chrome-Extension

Postman helps you be more efficient while working with APIs. Using Postman, you can construct complex HTTP requests quickly, organize them in collections and share them with your co-workers.

★ 3.6k  499

newman

Newman is a command-line collection runner for Postman

 JavaScript ★ 3.3k  472


postman-chrome-interceptor

Helper extension for the Postman packaged app. Also helps send restricted headers.

 JavaScript ★ 148  51

postman-collection

Javascript module that allows a developer to work with Postman Collections

 JavaScript ★ 119  50

postman-docs

Documentation for Postman, an API tool for Mac, Windows, Linux & Chrome.

 HTML ★ 38  31

Type: **All**Language: **All**


Working for Product Ops team
[Give us feedback](#)

Vinit Shahdeo

vinitshahdeo

Hello, Glad to see you here!

 SE Intern @postmanlabs

 VIT University Vellore

 vinitshahdeo@gmail.com

 <http://vinitshahdeo.com>

[Edit](#)[Overview](#) [Repositories 86](#) [Stars 8](#) [Followers 44](#) [Following 16](#)

Pinned repositories

[Customize your pinned repositories](#)

GitHub-LookBook

Look up the GitHub profiles with better UI experience. Build your GitHub Report Card!

 HTML ★ 4

jobtweets

This project is about searching the twitter for job opportunities using popular hashtags and applying sentiment analysis on this.

 Python ★ 11  6

CodeChefVIT

The website of CodeChef Student Chapter of VIT University Vellore.

 CSS  1

online-debate-system

Using google voice recogniton API to predict the "For the motion" and "Against the motion" using sentiment analysis

 PHP ★ 3  4

FaceRecognition

Face Recognition using Haar-Cascade Classifier, OpenCV and Python

 Jupyter Notebook

inspirational-quotes

A simple NPM Package which returns random Inspirational Quotes. Get your daily quote and stay motivated!

 JavaScript ★ 2

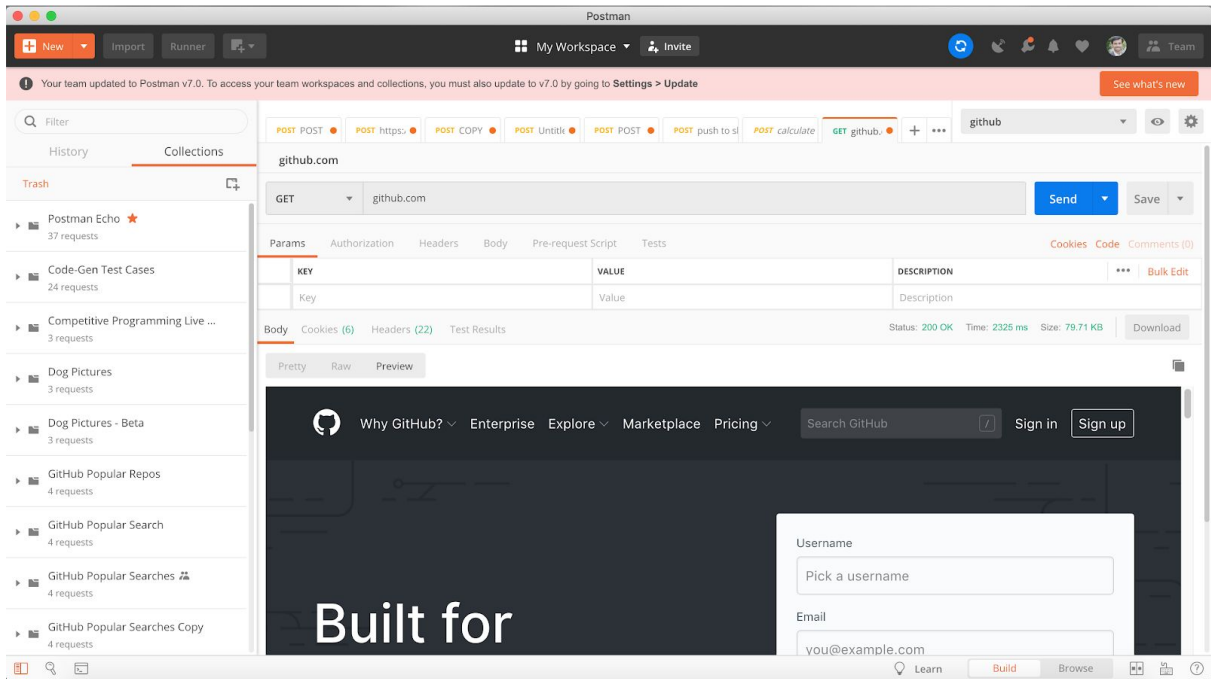
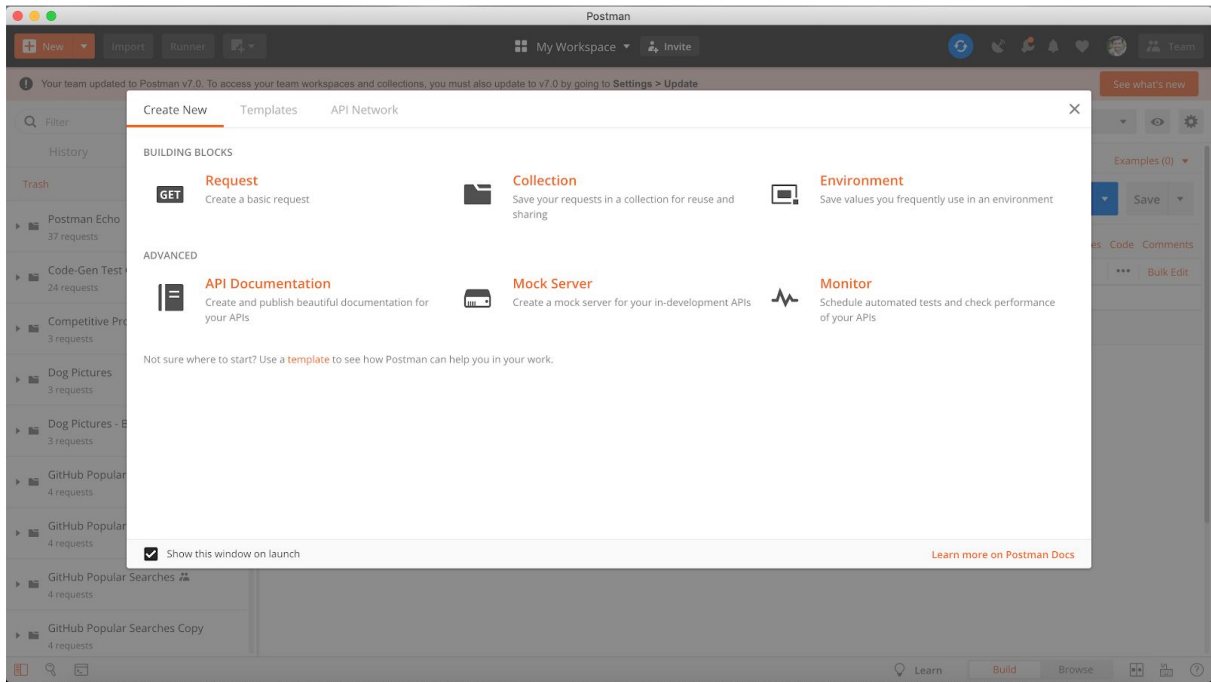
About Postman Desktop App

Postman Tools Support Every Stage of the API Lifecycle. Through design, testing and full production, Postman is there for faster, easier API development—without the chaos.

Features :

1. **Collections** - Postman Collections are executable descriptions of an API and are the cornerstones of Postman's built-in tools.
2. **Workspaces** - Postman Workspaces are collaboration spaces for teams of any size and can be organized to enhance your team's workflow.
3. **Built-In Tools** - Run requests, test, debug, create mock servers, monitor, run automated tests, and document your API with Postman's powerful built-in tools.

Every tool within Postman is based from the Postman Collection format: mocks, tests, documentation, monitors, and publishing all begin as Postman Collections. Postman Workspaces enhance team collaboration. Changes made to a team workspace sync in real-time so every team member is always working off of the most up-to-date version. Team admins can set permissions and manage contributors across multiple workspaces.



Need for Postman Automation

Postman Automation is truly important while testing products with a large number of integrations and/or frequent releases.

It is the ultimate tool for API automation. The main objective of QA automation is to reduce the combined amount of effort required for manually re-testing of a product which is fairly high.

Also, for removal of the manual testing efforts that are invested in testing a set of functionalities repeatedly.

For instance, Agile practices like continuous builds, the amount of time taken to receive a feedback for a manual regression test with the new code is too high.

API testing is also known as Integration testing. Integration testing focuses on verifying that the interactions of many small components can integrate together without issue. Since API tests bypass the user interface, they tend to be quicker and much more maintainable than GUI tests. Therefore, a good QA team will make fairly accurate projections based on the backlog at hand and the general information about the project and its architecture and use automation for regression test.

Pros and Cons of using Postman

Pros :

- Super easy API
- Wide range of functionality like support for all possible HTTP methods, saving progress, API to code conversion, changing environment of API development and many others.
- Helps to see the status codes, time taken for response and other performance parameters.
- Testing of APIs can be scheduled and automated.
- There is an option for importing of existing work so that you don't have to start from scratch.

Cons :

- Too many choices can overwhelm a beginner.
- It is not always true that an API developed in Postman will sure shot work in browser.
- Limited area of application(API testing and some other techniques).

API Driven Development

An API-first design is where the API is the first artifact that is created during the app development process. API contracts (API specification and signature, including the name, parameters, types, etc.) are either created by dedicated API architects or by front-end developers who are responsible for creating the end user experience. API contracts are finalized in collaboration with front-end and back-end developers.

Once the API contracts are finalized, the front-end developers build mocks around APIs and create and refine the end-user experience. In parallel, the back-end developers implement the underlying logic of the APIs. Dedicated test suites are created around these APIs and, in a way, they foster the idea of test-driven development. Finally, the implementations of the front-end and back-end developers are brought together. This is bound not to fail as long as the developers have coded honoring the API contracts as established in the first step.

At a code implementation level, APIs these days are typically designed using the REST architecture with JSON payloads. SOAP, XML, and other standards are found to be heavy and heading towards oblivion.

Security Review

The communication between Native App and Interceptor is not secure. The cookies that are being sent are in plain/text. We need to send the data in encrypted form. So I planned to use AES(Advanced Encryption Standard) algorithm to encrypt/decrypt the data at both ends.

Securing cookies is an important subject. Think about an authentication cookie. When the attacker is able to grab this cookie, he can impersonate the user. This article describes HttpOnly and secure flags that can enhance security of cookies.

HTTP, HTTPS and secure Flag

When HTTP protocol is used, the traffic is sent in plaintext. It allows the attacker to see/modify the traffic (man-in-the-middle attack). HTTPS is a secure version of HTTP – it uses SSL/TLS to protect the data of the application layer. When HTTPS is used, the following properties are achieved: authentication, data integrity, confidentiality. How are HTTP and HTTPS related to a secure flag of the cookie?

Let's consider the case of an authentication cookie. As was previously said, stealing this cookie is equivalent to impersonating the user. When HTTP is used, the cookie is sent in plaintext. This is fine for the attacker eavesdropping on the communication channel between the browser and the server – he can grab the cookie and impersonate the user.

Now let's assume that HTTPS is used instead of HTTP. HTTPS provides confidentiality. That's why the attacker can't see the cookie. The conclusion is to send the authentication cookie over a secure channel so that it can't be eavesdropped. The question that might appear in this moment is: why do we need a secure flag if we can use HTTPS?

Let's consider the following scenario to answer this question. The site is available over HTTP and HTTPS. Moreover, let's assume that there is an attacker in the middle of the communication channel between the browser and the server. The cookie sent over HTTPS can't be eavesdropped. However, the attacker can take advantage of the fact that the site is also available over HTTP. The attacker can send the link to the HTTP version of the site to the user. The user clicks the link and the HTTP request is generated. Since HTTP traffic is sent in plaintext, the attacker eavesdrops on the communication channel and reads the authentication cookie of the user. Can we allow this cookie to be sent only over HTTPS? If this was possible, we would prevent the attacker from reading the authentication cookie in our story. It turns out that it is impossible, and a secure flag is used exactly for this purpose – the cookie with a secure flag will only be sent over an HTTPS connection.

HttpOnly Flag

In the previous section, it was presented how to protect the cookie from an attacker eavesdropping on the communication channel between the browser and

the server. However, eavesdropping is not the only attack vector to grab the cookie.

Let's continue the story with the authentication cookie and assume that XSS (cross-site scripting) vulnerability is present in the application. Then the attacker can take advantage of the XSS vulnerability to steal the authentication cookie. Can we somehow prevent this from happening? It turns out that an HttpOnly flag can be used to solve this problem. When an HttpOnly flag is used, JavaScript will not be able to read this authentication cookie in case of XSS exploitation. It seems like we have achieved the goal, but the problem might still be present when cross-site tracing (XST) vulnerability exists (this vulnerability will be explained in the next section of the article) – the attacker might take advantage of XSS and enabled TRACE method to read the authentication cookie even if HttpOnly flag is used. Let's see how XST works.

XST to bypass HttpOnly flag

GET and POST are the most commonly used methods by HTTP. However, there are not the only ones. Among the others is HTTP TRACE method that can be used for debugging purposes. When the TRACE request is sent to the server, it is echoed back to the browser (assuming that TRACE is enabled). It is important here, that the response includes the cookie sent in the request.

Let's continue the story of the authentication cookie from previous sections. The authentication cookie is sent in HTTP TRACE requests even if the HttpOnly flag is used. The attacker needs a way to send an HTTP TRACE

request and then read the response. Here, XSS vulnerability can be helpful. Let's assume that the application is vulnerable to XSS. Then the attacker can inject the script that sends the TRACE request. When the response comes, the script extracts the authentication cookie and sends it to the attacker. This way the attacker can grab the authentication cookie even if the HttpOnly flag is used.

As we have seen, the HTTP TRACE method was combined with XSS to read the authentication cookie, even if the HttpOnly flag is used. The combination of HTTP TRACE method and XSS is called cross-site tracing (XST) attack.

It turns out that modern browsers block the HTTP TRACE method in XMLHttpRequest. That's why the attacker has to find another way to send an HTTP TRACE request. One may say that XST is quite historical and not worth mentioning. In my opinion, it's good to know how XST works. If the attacker finds another way of sending HTTP TRACE, then he can bypass the HttpOnly flag when he knows how XST works. Moreover, the possibility/impossibility of sending an HTTP TRACE request is browser-dependent – it would just be better to disable HTTP TRACE and make XST impossible. Finally, XST is a nice example that shows how an attacker might use something that is considered to be harmless itself (enabled HTTP TRACE) to bypass some protection offered by the HttpOnly flag. It reminds us that details are very important in security, and the attacker can connect different pieces to make the attack work.

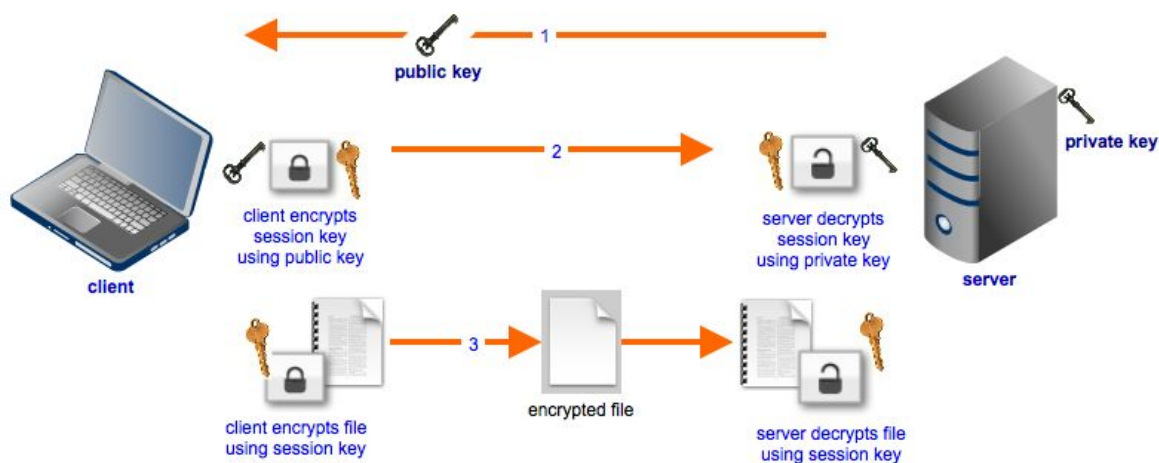
Advanced Encryption Standard

AES is an iterative rather than Feistel cipher. It is based on '*substitution-permutation network*'. It comprises of a series of linked

operations, some of which involve replacing inputs by specific outputs (substitutions) and others involve shuffling bits around (permutations).

Interestingly, AES performs all its computations on bytes rather than bits. Hence, AES treats the 128 bits of a plaintext block as 16 bytes. These 16 bytes are arranged in four columns and four rows for processing as a matrix –

Unlike DES, the number of rounds in AES is variable and depends on the length of the key. AES uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys. Each of these rounds uses a different 128-bit round key, which is calculated from the original AES key.



For encryption/decryption of cookies, we are using CryptoJS.

```
var CryptoJS = require("crypto-js");  
var data = [{id: 1}, {id: 2}]
```

```
// Encrypt  
  
var ciphertext = CryptoJS.AES.encrypt(JSON.stringify(data), 'secret key  
123');  
  
// Decrypt  
  
var bytes = CryptoJS.AES.decrypt(ciphertext.toString(), 'secret key  
123');  
  
var decryptedData = JSON.parse(bytes.toString(CryptoJS.enc.Utf8));
```

AES features

The selection process for this new **symmetric** key algorithm was fully open to public scrutiny and comment; this ensured a thorough, transparent analysis of the designs submitted.

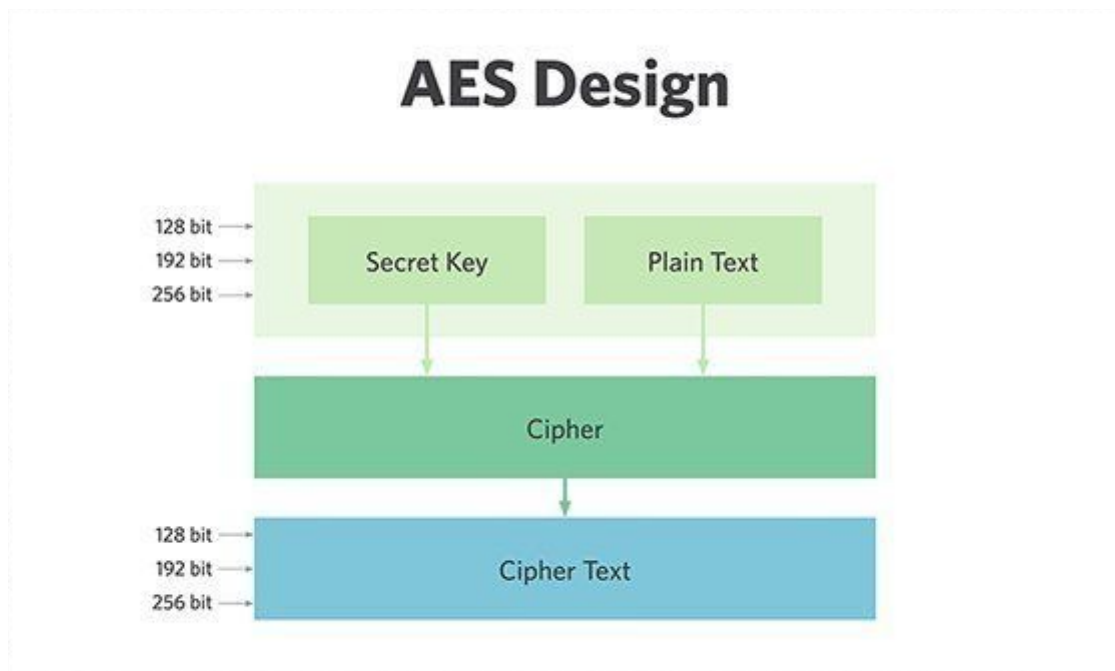
NIST specified the new advanced encryption standard algorithm must be a block cipher capable of handling 128 bit blocks, using keys sized at 128, 192, and 256 bits; other criteria for being chosen as the next advanced encryption standard algorithm included:

- **Security:** Competing algorithms were to be judged on their ability to resist attack, as compared to other submitted ciphers, though security strength was to be considered the most important factor in the competition.
- **Cost:** Intended to be released under a global, nonexclusive and royalty-free basis, the candidate algorithms were to be evaluated on computational and memory efficiency.

- **Implementation:** Algorithm and implementation characteristics to be evaluated included the flexibility of the algorithm; suitability of the algorithm to be implemented in hardware or software; and overall, relative simplicity of implementation.

How AES encryption works?

AES comprises three block ciphers: AES-128, AES-192 and AES-256. Each cipher encrypts and decrypts data in blocks of 128 bits using cryptographic keys of 128-, 192- and 256-bits, respectively. The Rijndael cipher was designed to accept additional block sizes and key lengths, but for AES, those functions were not adopted.



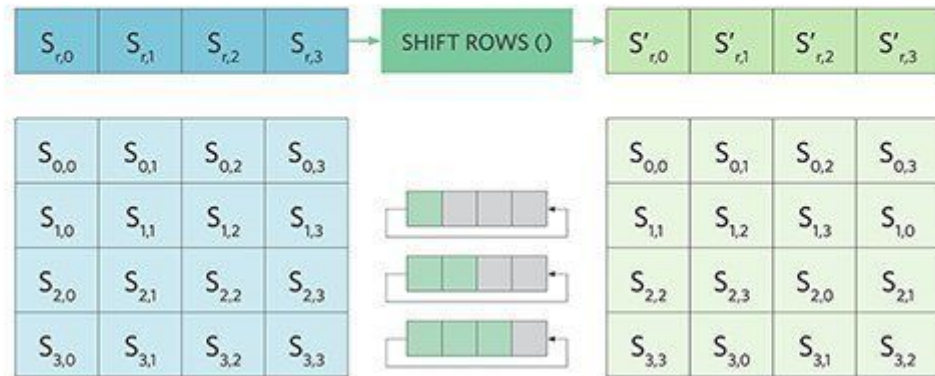
Symmetric (also known as secret-key) ciphers use the same key for encrypting and decrypting, so the sender and the receiver must both know -- and use -- the

same secret key. All key lengths are deemed sufficient to protect classified information up to the "Secret" level with "Top Secret" information requiring either 192- or 256-bit key lengths. There are 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys -- a round consists of several processing steps that include substitution, transposition and mixing of the input plaintext and transform it into the final output of ciphertext.

The AES encryption algorithm defines a number of transformations that are to be performed on data stored in an array. The first step of the cipher is to put the data into an array; after which the cipher transformations are repeated over a number of encryption rounds. The number of rounds is determined by the key length, with 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys.

The first transformation in the AES encryption cipher is substitution of data using a substitution table; the second transformation shifts data rows, the third mixes columns. The last transformation is a simple exclusive or (XOR) operation performed on each column using a different part of the encryption key -- longer keys need more rounds to complete.

AES ShiftRows() Transformation Step



Security - Summary

Security of cookies is an important subject. HttpOnly and secure flags can be used to make the cookies more secure. When a secure flag is used, then the cookie will only be sent over HTTPS, which is HTTP over SSL/TLS. When this is the case, the attacker eavesdropping on the communication channel from the browser to the server will not be able to read the cookie (HTTPS provides authentication, data integrity and confidentiality).

When HttpOnly flag is used, JavaScript will not be able to read the cookie in case of XSS exploitation. It was also presented how the combination of HTTP TRACE method and XSS might be used to bypass HttpOnly flag – this combination is cross-site tracing (XST) attack. It turns out that modern browsers block the HTTP TRACE method in XMLHttpRequest. However, it's still important to know how XST works. If the attacker finds another way of sending HTTP TRACE, then he can bypass an HttpOnly flag when he understands how XST works.

6. Future Work

1. ***E2E Client side Installation for Windows, MacOS, Linux*** - There are different installation steps for different platforms as the Chrome Native Messaging API is OS specific.
2. ***Releasing nativeServer as NPM module*** - NativeServer will act as middlemen between Native App and Interceptor. It will be a binary executable file which will be originally written using NodeJS.
3. ***Securing the IPC bridge*** - The socket being opened by NativeServer should secure. No third party Apps should listen to this.
4. ***Cookie Serialization/Deserialization*** - The transfer of cookie should follow some format which will be used of serialization and deserialization of the cookies.
5. ***Encryption/Decryption*** - The cookies should be transferred in the form of encrypted string. AES with a shared secret key between Postman Native App and Interceptor will be used to achieve this

7. References

1. Postman Official Blog (<https://blog.getpostman.com/>)
2. Chrome APIs (https://developer.chrome.com/extensions/api_index)
3. Postman GitHub Repositories (<https://github.com/postmanlabs>)
4. Issues raised by users on GitHub
(<https://github.com/postmanlabs/postman-app-support/issues>)
5. Tough-cookies NPM module (<https://www.npmjs.com/package/tough-cookie>)
6. MDN Web Docs (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>)
7. Crypto (<https://nodejs.org/api/crypto.html>)