Authentication Strategies

* Hvordan man sikre sig at alle ikke bare kan få fat i den applikation

Authentications strategies for specific web applications:

* Web Traditional, multipage
* Web SPA
* Web API’s
* Mobile friendly
* Server to server

Stateless passer ikke så godt sammen med at være logget ind da det jo er en state.

Authentication strategy filters

How to avoid brute force attacks

* Folk er normal ikke så gode til at lave stærke passwords
* Vi har et problem hvis folk kan lave alt for mange loginforsøg
* Cooldowns/delay or lockdown upon multiple login attempts
* Use CAPICHA puzzles to avoid brute force actions.

Consider for Authentication demo

* Reauthenticate for important actions
* Conceal whether a user exists
* Prevent brute force attacks
* Don’t use Default or hard-coded credentials
  + Tomcat plejede at have default admin passwords
  + Mange routere plejede at have deres passwords hard-coded ind i deres arkitektur
* Always hash and salt passwords
* Always authorize on server
  + Klienten skal ikke havde ansvaret for authorization.

Best security advice ever:

* Use existing authentication frameworks whenever possible instead of creating one yourself
  + Always update the when using well known frameworks, as per OWASP A6 & A9.
* Java
  + Apache Shiro
  + OACC
* Node.js
  + Passport framework

I HTTP protokollen ligger der også nogle security metoder (http er bare protokollen men vi skal stadig altid bruge http’s’, så vi som klienten altid kan sikre os at serveren er den rigtige og at serveren kan sikre sig at vi er eden rigtige klient)

* Basic http Authentication (part of the http protocol)
  + Client requests protected resource
  + Server requests username and password
  + Client sands username and password
  + Server authenticates username and password, and returns the requested recource
  + Hvis denne transaktion ikke er secure med SSL(https), vil både username and password være base64 encoded, hvilket IKKE er sikkert at bruge i sig selv.
* Form-based Authentication (Exist in a similar way in all web technologies)
  + Client requests the resource
  + The server redirects the client to a login page.
  + The client submits the form to the server
  + The server security checks and either redirects to an error page or to the initial resource
* Digest
* Client certificate
* Token based Authentication (What we will look for)
* Single sign on

Med restfull webservices ligger state ikke på serveren, men er i stedet hold hos klienten.

Vi skal have en måde at sende vores state over til klienten, men så andre ikke kan overtage eller lave state om.

Her bruger vi JWT for at sikre os at ingen kan pille ved state uden at den bliver invalid.

* Ved at kigge på signaturen og den tilhørende algoritme i et hash med en secret fra serveren, kan man kan se om nogen har ændret i payloaden ud fra hash-værdien. Klienten kan ikke selv generere hash-værdien, da det kun er serveren der kender den brugte secret, og derfor er det kun serveren der kan klargøre om nogen har ændret i payload og derved validere denne token, når den sendes i en request.

På populære sider bruger man ofte en load blancer der kan balancere load mellem applikationens forskellige servere.

* Et problem her at men ville kunne møde problemer hvor hver af disse servere har hver deres chashe og derfor ikke alle kan genkende vores sessionid fra browseren.
* En løsning ville måske være at samle alle applikationens serveres cashe, men dette ville lave en bottleneck og et sårbart punkt for en single point of faliure.
* En anden løsning ville være at synkronisere alle cashe for alle servere som applikationen snakker med, i en distributed cashe.
* Men den bedste løsning ville være hvis vi kan holde state i klienten, og s pare parse den med hver request vi laver gennem load balanceren.

I stedet for at skulle logge ind hver gang man laver en request til serveren vil det gennem en sikker kanal være bedre at gemme sin autoritet i en JWT og en strategi som Token based auth.

Token based auth:

* Good Things
  + It is stateless
  + It is self-contained where the payload contains all the information about the user, avoiding the need to query the database more than once.
  + It is signed by the server, so that we can verify the user.
  + It scales easily
  + Can be used among many services
  + Can be used among man technologies and languages.
* The problems
  + Once a token is issued, it is valid until it expires, we can revoke it only by blacklisting it
  + Logout. We can logout by deleting the token, but the token will still be valid!
  + If the string secret get compromised, all tokens are potentially compromised.
  + The strategy is vulnerable to XSS and CSRF attacks, so precautions must be taken.

Revoking tokens

* We can revoke tokens by blacklisting them to a list of invalid tokens.
* This however removes some of the benefits since we have to do additional lookups on the server side.

HTTPS med SSL er alfaomega når det gælder web security.

Exercises:

***a)*** *Create a MySQL database*

***b)*** *For the first exercises, Tomcat and not you, will be in charge of the authentication.*

**1) Basic Authentication (we will do this in the class)**

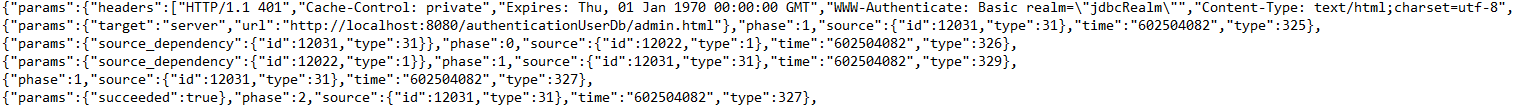
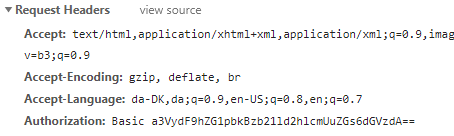
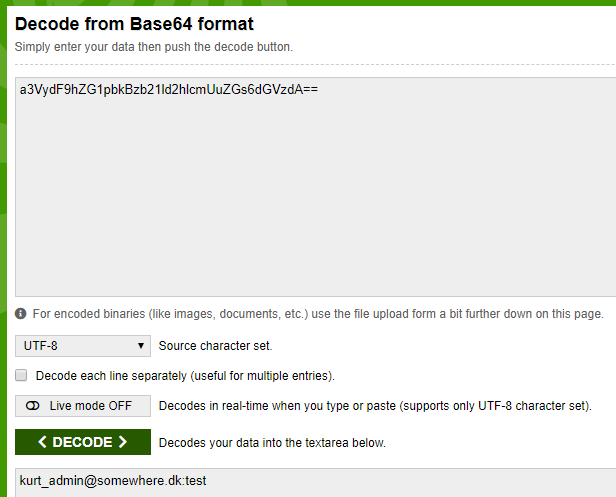
Create a new Maven Web-project with NetBeans and do the following:

**1a)** Setup Tomcat to use the database created above as a JDBC-realm

**b)** Create two new static pages: user.html and admin.html

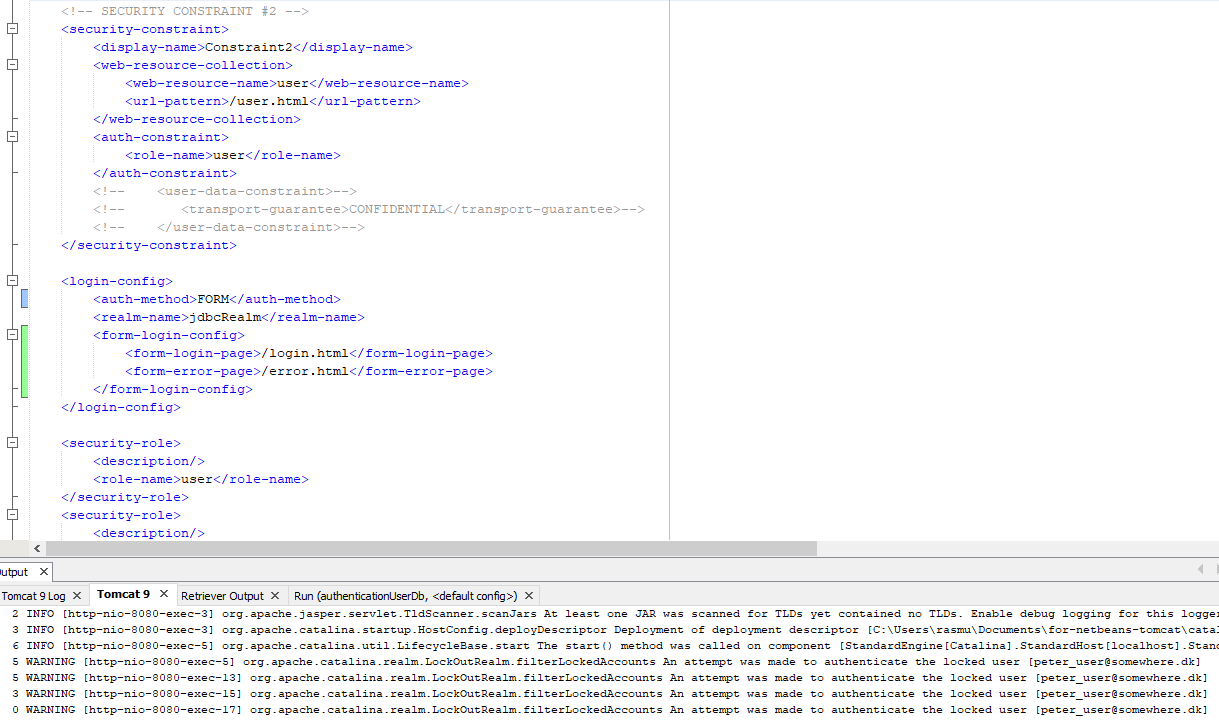
**c)** Create a web.xml file in the project

**e)** Monitor the requests and responses between client and server

* Explain how your Browser knows when to put up the login window (chrome://net-internals/#events)
  + With our <security-constraint> declaration for secure pages, we can declare which url patterns should be accessible to what role-names. With the <login-config>, we can decide what auth-method to use, in this cade BASIC for the JDBSRealm. This way the user will be met by a login window each time they try to access a constrained url-pattern, without having been authenticated or authorized as that role.  
    
  + With this the browser knows to open the login window upon reaching a code 401 which is the http error code for ‘unauthorized’. We can see this if we go to chrome://net-internals/#events and start logging the browser while we attempt to access this authorized resource.  
    
* Explain how your server knows that you are logged in for subsequent request
  + The basic authentication server sends an Authorization header for each subsequent request you do in the session after your first login. This means that you only are met by the login window once.  
    
* Copy the part that holds the Authorizations part and use the decode option on this link to comment what always must be done as a supplement when using Basic Http (and all other) authentication
  + cGV0ZXJfdXNlckBzb21ld2hlcmUuZGs6dGVzdA==  
    <https://www.base64decode.org/>  
    [peter\_user@somewhere.dk:test](mailto:peter_user@somewhere.dk:test)  
    
  + Since the Authorization header is just encoded with base 64 it is not safe and can be easily decoded. If we want it to be secure, we must encrypt it in something like a JWT token, with a hashed signature and secret from the server so that we know it has not been tampered with.
* **f)** Reflect (write down) the pros & cons of using Basic Authentication, and the use cases where it (still) could be relevant
  + Pros:
    - Implementation is pretty simple, as there is no encryption involved
    - Take relatively less time to respond as it has only one call
    - The lack of token creation and encryption method gives Client an advantage of using less code to call the API
    - The information is retrieved from the server with just one call, making it faster than other complex authentications
  + Cons:
    - Is no secure to use without SSL, since base64 in itself is not secure alt all.
    - Not possible to change appearance so it stays with the default look.
    - SSL takes time to run basic HTTP, so this will make the response time considerably slow
    - The lack of encryption makes the security risk fairly high
    - Only username and password can be sent with the request.
  + Relevant use
    - Basic authentication is simple, moderately secure with https and easy to use, and is therefore still widely used for simple identification purposes.

**2) Form-based  Authentication (we will do this in the class)**

**a)** Repeat step 1d+1e, but change the Authentication strategy from HTTP to Form-based Authentication

**b) Limit the number of login tries**

**c)** Reflect (write down) the pros & cons of using Form-based Authentication, and the use cases where it could be relevant

* Pros:
  + Full control over the authentication code
  + Full control over appearance of the login form
  + It works with any browser
* Cons:
  + Takes longer time to set up since you have to create your own login interface
  + You have to maintain a catalog with user credentials
  + You have to take additional precautions against the interception of network traffic
  + Heavy, slow.
* Relevant use
  + Existing security implementations that depend on form-based configuration for legacy and security reasons.
  + Server-side form-based authentication addresses some outstanding issues relating to password modification.

**3) Basic/Form-based authentication with a REST-API**

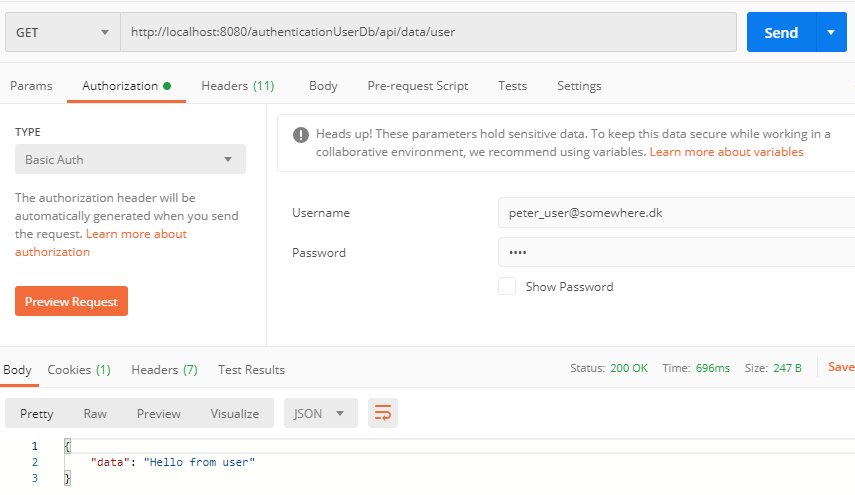
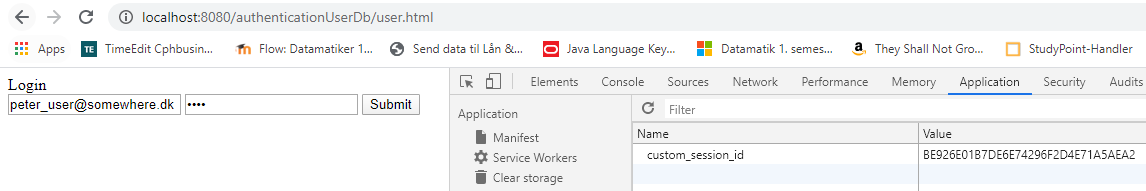
**a)** Repeat step **a)** from exercise-1, since we, one more time are leaving security to Tomcat.

**b)** Create, using the NetBeans Wizard, the skeleton code for a Restful web Service

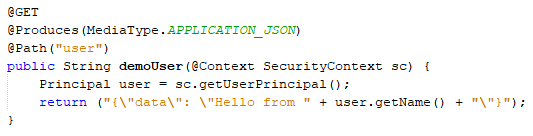
**c)** Create the following two GET-endpoints

**d)** Secure the two endpoints, declaratively, similar to what you did in ex-1+2.

**e)** Test the two endpoints, using Postman

* Basic authentication  
  
* Form-based  
  

**f)** change the two endpoints to return the responses sketched below

* xx/api/demo/user : should return a simple json-string like: “Hello from USER: *users-email*”
* 

**4) REST authentication with annotations**

* <security-constraint> element must be present; the web container checks for security before JAX-RS does and without a <security-constraint>, the proper security context is not set. Thus when JAX-RS invokes isUserInRole(role), it always returns false.
* In addition, either <security-role> element(s) in web.xml or @DeclareRoles annotation must be present.
* Lastly, if using Jersey, RolesAllowedDynamicFeature needs to be registered in the Application class to enable annotation-based security.

**Implementing our own JWT-based Authentication Strategy for a REST-API.**

**a) Getting ready for the exercise:**