

**IIT Madras**  
ONLINE DEGREE

**Modern Application Development – I**  
**Professor Nitin Chandrachoodan**  
**Department of Electrical Engineering**  
**Indian Institute of Technology Madras**  
**Sessions**

Hello, everyone, and welcome to this course on modern application development.

(Refer Slide Time: 00:16)



## Sessions

Now, with all of the different mechanisms that can be used for implementing access control, let us look at the notion of sessions.

(Refer Slide Time: 00:26)



## Session management

- Client sends multiple requests to server
- Save some “state” information
  - logged in
  - choice of background colour
  - ...
- Server customizes responses based on client session information

### Storage:

- Client-side session: completely stored in cookie
- Server-side session: stored on server, looked up from cookie

Now, what do we mean by session management? The idea of a session is that a client might send multiple requests to the server, and the server needs to have some kind of state information. What is the client I mean have they logged in? Have they got anything in the shopping cart? What is the present state of information that I know about the client? And the server can then customize the responses based on the client session information.

Now, all this means that there has to be some form of storage, some information has to be stored somewhere. You could have something which is a so called client side session, where everything gets stored just in, inside a cookie itself. Or, alternatively, you could have something called a server side session where the cookie is still used.

The cookie sort of just tells the server, which is the client, and the server then needs to look up in its internal database and say, okay fine for this user use this background color, for this user use this shopping cart. And they have currently added these items to the shopping cart, all of that information is now stored in the server end. All this ultimately is, these are all mechanisms that have been put together, because HTTP by its very nature (()) (01:42).

And we need to sort of hack something on top of it in order to provide that sense of, the state of a system. And the final mechanisms that have come up are actually quite elegant and reasonably easy to use. But there are lots of cases, they were sort of added on after the fact. So, there are multiple cases where security issues were discovered only well after some ideas were introduced to it.

(Refer Slide Time: 02:08)

## Cookies

- Set by server with Set-Cookie header
- Must be returned by client with each request
- Can be used to store information:
  - theme, background colour, font size: simple no security issues
  - user permissions, username: can also be set in cookie
    - must not be possible to alter!

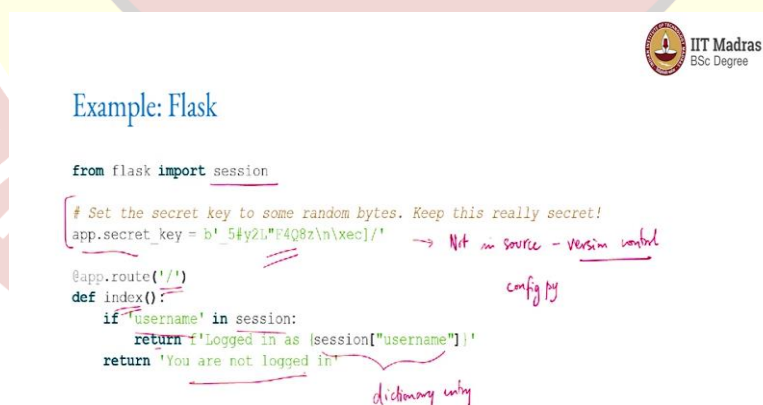


So, the cookie itself, as we saw earlier, now will be set by the server with the set cookie header, the client must return it on each request. Now the cookie by itself is after all just a string. It can itself be used to serve, store certain information. So, something like for example, what is the desired font or the desired background color, could be stored inside the cookie. And the server will then each time it gets a request, it sort of unwraps the cookie, looks inside, it finds out what information to use, changes the page based on that and sends it back.

It can be run that page, probably not being efficient, but it could be done if required. What that means is, if you decide to store something sensitive, like the user name inside a cookie, then you have to ensure that it cannot be altered. Because otherwise, let us say that a client logs in as some user ABC. They get a cookie, the client then goes and modifies that cookie and changes the name to...

Next time when they send back the cookie, the server goes, oh who is logging in, let me give them the full admin dashboard and in order to change anything they want. So, the client should not be able to change the cookie. That can also obviously be done, the server needs to have some kind of secret of its own, which it uses in order to encrypt the cookie and send it back. Which means that, the client just has that cookie, it can send it back to the server each time. But it cannot really modify it in any way because it does not know the secret but the server do.

(Refer Slide Time: 03:43)



```
Example: Flask

from flask import session

# Set the secret key to some random bytes. Keep this really secret!
app.secret_key = b'5#y2L"F4Q8z\n\xec]/'

@app.route('/')
def index():
    if 'username' in session:
        return f'Logged in as {session["username"]}'
    return 'You are not logged in'
```

Handwritten notes on the slide:

- Not in source - version control (pointing to the secret key)
- config.py (pointing to the @app.route decorator)
- dictionary entry (pointing to session["username"])

So, an example of this using flask. So, flask has a module called session. This is basically for simple client side sessions. So, server side sessions require something a little bit more complicated, they usually require some additional items to flask. So, like I said, there has to

be a secret key associated, which is used for encoding the cookie. If you do not keep this secret, that is bad news, because it means that the client will be able to alter the cookie if required.

So, this is important. When you are creating an app, you need to make sure that any secret keys that you put in are actually kept secret. The other thing is they should not ever be in your source code. In particular, it should never go into version control systems. Usually the place to put it would be in some kind of config.py or some kind of file of that sort, where, that content of py is never put into version control by itself.

You need to create a config.py each time, fresh when you are creating a new `()` (04:58). So, how does this session work? It is as simple as this and so, basically the root path, there is a function index designed over there. It just checks, so session finally ends up becoming a dictionary. And what we are saying is, if the key user name is present in the session dictionary, then you can just return logged in as session of username.

So, this basically is a dictionary. If not, then it basically says you are not logged in. So, it becomes very easy to check certain values in the cookie. Now, what happens when I actually want to log in?

(Refer Slide Time: 05:42)



### Example: Flask

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
    <form method="post">
      <p><input type="text" name="username">
      <p><input type="submit" value="login">
    </form>
    '''

@app.route('/logout')
def logout():
    # remove the username from the session if it's there
    session.pop('username', None)
    return redirect(url_for('index'))
```

*Very insecure*

*Form*

## Example: Flask

```
from flask import session

# Set the secret key to some random bytes. Keep this really secret!
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

@app.route('/')
def index():
    if 'username' in session:
        return f'Logged in as {session["username"]}'
    return 'You are not logged in'
```

*Handwritten notes:*

- Not in source - version control
- config.py
- dictionary entry

That requires some additional piece of logic over there, that I could create a slash login root with multiple GET and POST methods. If it is a POST, it means that I am actually logging in. What I do is I just basically take this (06:02) session of username, take whatever is provided from the form as username and set it into this and redirect to index. And if it was a GET request, on the other hand, I will show HTML form.

Now, this is very insecure. And obviously, you should only take this as a basic example of how sessions work. Why? Because it is completely trivial for me I mean, this form does not ask for a password, it does not ask for anything. I could just type in any name I want, and enter it over there. So, this is just a demonstration of how I can use a cookie. So, you should not in any way think that this is actually a valid way of (06:48) log in.

So, one thing, of course, is whatever you entered over here is what you get set in the session of username. After that, once you get the cookie back, you will still not be able to modify it. Because you do not know the secret key. How about logging out? That is also quite easy to do. I can just `session.pop` allows me to just take out a username key and just remove it from the session.

And then redirect me back to index, which means that at that point, I will go back and it will show me this thing, it will say you are not logged in. And maybe redirect, maybe log in or show me the login page or whatever it is, that is up to you if you are interested in.



(Refer Slide Time: 07:34)



### Security issues

- Can user modify Cookie?
  - Can set any username
- If someone else gets Cookie, can they log in as user?
  - Timeout
  - Source IP
- Cross-site requests
  - Attacker can create page to automatically submit request to another site
  - If user is logged in on other site when they visit attack page, will automatically invoke action
  - Verify on server that request came from legitimate start point

So, of course the security issues, can the user modify the cookie, in which case they can change the user name to anything they want? What happens if someone else gets hold of the cookie? Can they log in as the same user? Quite possible, especially this is dangerous in public machines. So, let us say you go to a browsing center, internet center, you log in to something, your cookies are all there on the system.

Unless you explicitly type log out, you make sure that you log out of the system, then the server deletes the cookies. Then you somebody who gets hold of that machine, they cannot do anything with it. Because there is no confidential information stored in the cookies itself. But if you forget to log out, it means that somebody else who is coming along and using that machine can log in as you.

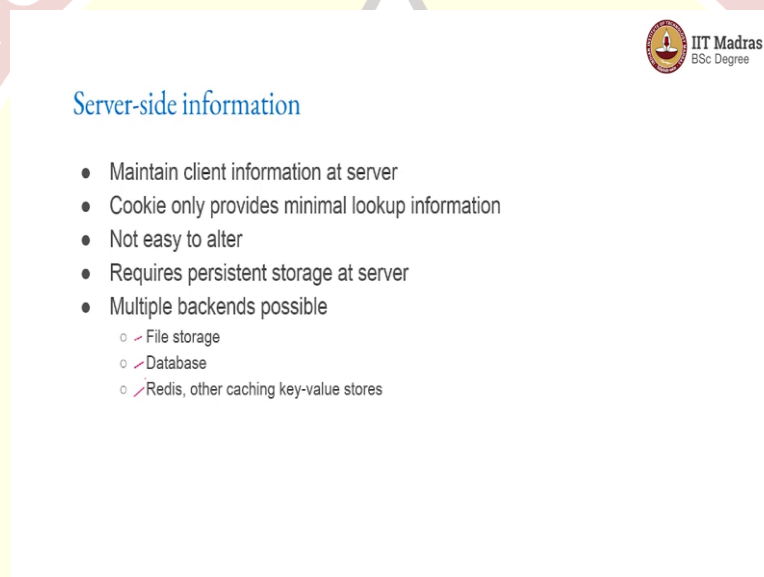
There is also something called a cross site request, which can be used in order to get you to automatically submit a request to another site. Let us say I am logged in to Gmail on one tab of a browser. And I go to some other site where there is one malicious piece of code, which basically says, just go and change the password on Gmail, submit to a form on Gmail directly information after you have to change the password.

Now, since I have already logged into Gmail, and I know the email server cannot easily make out whether the request came from that particular page where I clicked on it, or from somewhere else, because after all, it is coming from the same machine, from the same browser. And all the authentication cookie, all of that is perfectly fine. What happened is there was a different site, which somehow induced me to send a request to Gmail to change my password or change my account information.

So, the server cannot really make that out. There is a way of getting around this which is basically the server actually sets a token along with each form where something sensitive like emails or passwords or anything changed. And if that is the case, then that you can only get to the email change form, from the known path. Again, like I said, that is a little bit out of scope of what we need to do over here.

But it is something that you need to be aware of. Cross site request forgeries are major security issue. And usually what happens is that, most frameworks have ways of dealing with this. So, make sure you are aware of it and take it into account in whatever you are implementing.

(Refer Slide Time: 10:11)



**Server-side information**

- Maintain client information at server
- Cookie only provides minimal lookup information
- Not easy to alter
- Requires persistent storage at server
- Multiple backends possible
  - File storage
  - Database
  - Redis, other caching key-value stores

Now, what happens on the server side? You need to maintain the client information on the server. The cookie by itself only provides a minimal look up information. All that it says is, it is not easy to alter. But the server then needs to take care of storing the rest of the information about the client. There are multiple back ends that are available to directly store something in a file, or you could store it in a database.

There are other kinds of, the so called key value stores like Redis, which are very high performance things. That are just meant for storing the information about who is currently logged in. It makes it very easy to check whether a particular user is logged in, it is lightweight, and a lot of information about the user could be stored in that.

But it is also temporary, because Redis does not really save things to this by default. So, usually, what you would do is all this temporary session storage would be in Redis. And then



eventually, if something needs to be updated in the master database, you get that done separately.

(Refer Slide Time: 11:15)



### Enforce authentication

- Some parts of site must be **protected**
- How?
  - Enforce existence of specific token for access to those views
- Views:
  - determined by controller
- Protect access to controller!
  - Flask controller - Python function
  - Protect function - add wrapper around it to check auth status
    - Decorator!

Now, how do you go about enforcing authentication? Some parts of the site must be protected. And the question becomes, how should I enforce it? We already looked at the mechanisms, which means that some kind of a specific token, whether it is basic authentication token, which came about (( ))(11:38), or it is an API level token, or some other token, must be given by the client to the server in order to access those views.

Now, at the application level, how do you do this sort of fine-grained control and not sort of, the server itself is not sort of saying, you either have access to the entire site, or you do not. It is saying; your grades, this particular piece of information you have access to. So, how do I do that kind of fine-grained control? I need to protect a specific view. And who determines which view I get to see? The controller, which means that I now need to basically protect access to the controller.

And the simple way of thinking about it is I have a controller already in place, which has been coated specific way. Can I add some extra functionality around it to check the authentication status? And how do I add functionality? Use a decorator.

(Refer Slide Time: 12:37)



### Example - flask\_login

```
from flask_login import login_required, current_user
...
@main.route('/profile')
@login_required  # also checks role?
def profile():
    return render_template('profile.html', name=current_user.name)
```

An example is there is a module called flask login, a separate item module. Now, this is not the only way that you can implement logins. So, I am just using this as an example. I am not sort of recommending this as necessarily the only way of doing logins. But in this last login module, there are a few functions. One of them is called login required. It also has information which is the current user. So, these are functions.

So, what you do is you have the @main.route(/profile) or @app.route(/profile), and basically say /profile. And what it says is that in order to see the /profile path, I need to be logged in. How do I specify that? I put one more decorator, so see how decorators are being changed over here. So, there is an @main.route(). And inside that there is an app login required, (( ))(13:33) @login\_required to match what we have out here.

And what it is saying is that I will then go run this functionality first, for login required, which in turn will check whether the correct authentication tokens and so on are present. If not, this login required will itself redirect you to the login page. But if it is present, it will come back over here and render the template for profile.html, take the current users and pass that in and so on. So, this decorator basically allowed me to change what access is there.

I could have done something more, I could have had a decorator, which also checks the role perhaps. And say that, either this particular user or if I am an admin, then allow the admin access to any user's information. So, that is also a possibility that can be done.

(Refer Slide Time: 14:37)



### Example - flask\_login

```
from flask_login import login_user, logout_user, login_required
...
@auth.route('/logout')
@login required
def logout():
    logout_user()
    return redirect(url_for('main.index'))
```

Now, how do you log out? Once again, the interesting thing is you still need to have this decorator login required. Because otherwise, you cannot log out unless you are logged in to start with. But all that the logout needs to do is essentially call the logout user functionality. What would that do? This has to you need the session information on the server side. So, this is after all your flask code?

On the server side, the moment that logout user function is called, you just has to locally delete the session information. And then go back to the main index. So, by using decorators, you can basically enforce authentication on specific controller.

(Refer Slide Time: 15:17)



### Transmitted data security

- Assume connection can be "tapped"
- Attacker should not be able to read data
- HTTP GET URLs not good:
  - logged on firewalls, proxies etc
- HTTP POST, Cookies etc:
  - if wire can be made safe, then good enough

How to make the wire safe?

Now, all of this is fine. But the one question that comes up in relation to all of this is how do we make the wire itself safe? That is to say I am transmitting information from client to server and from server back to the client. I already know that, the main problem is that the attacker should not be able to read the data. How do I go about making this wire safe and ensuring that the attacker by themselves cannot see what is being transferred?

