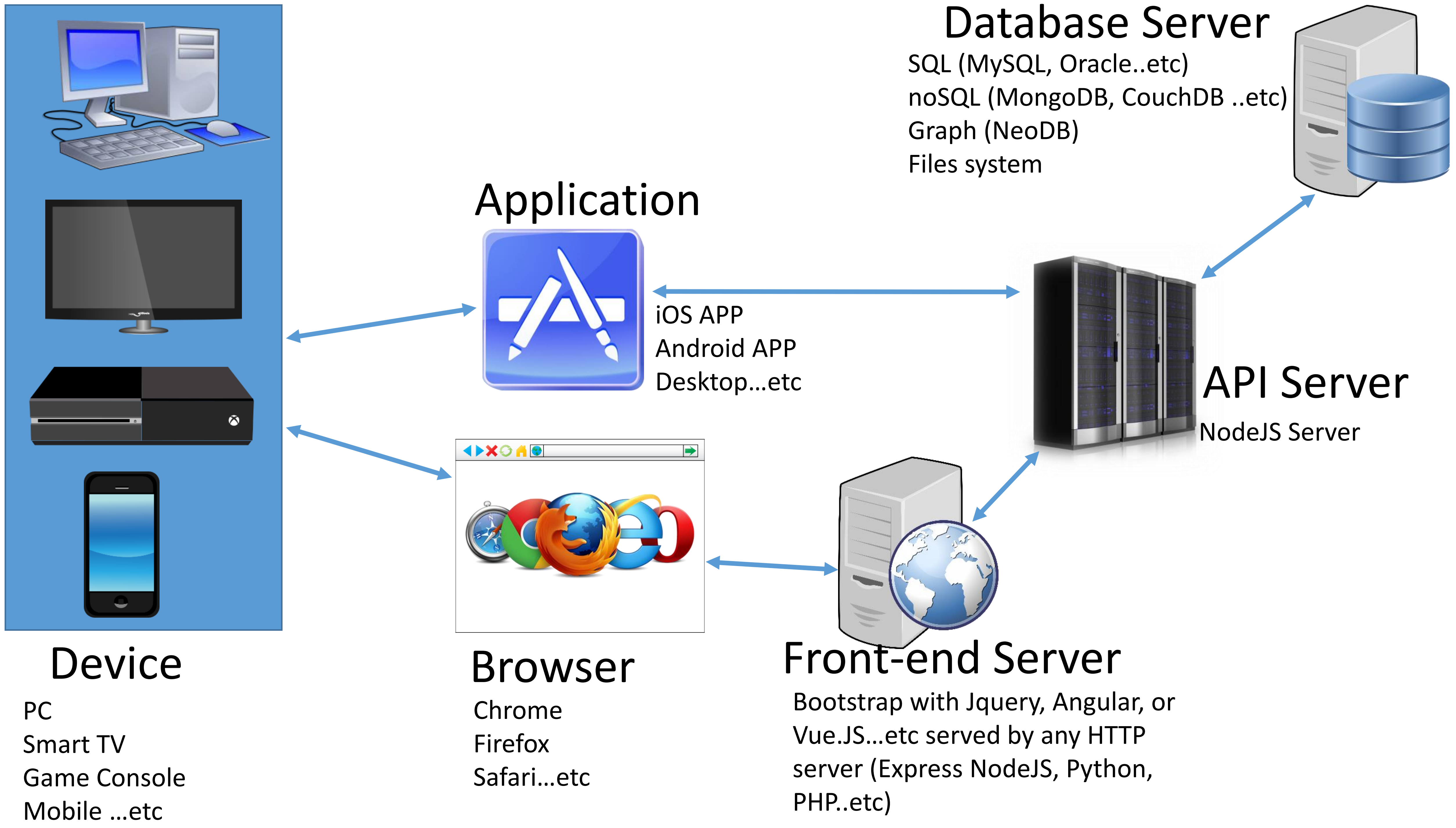
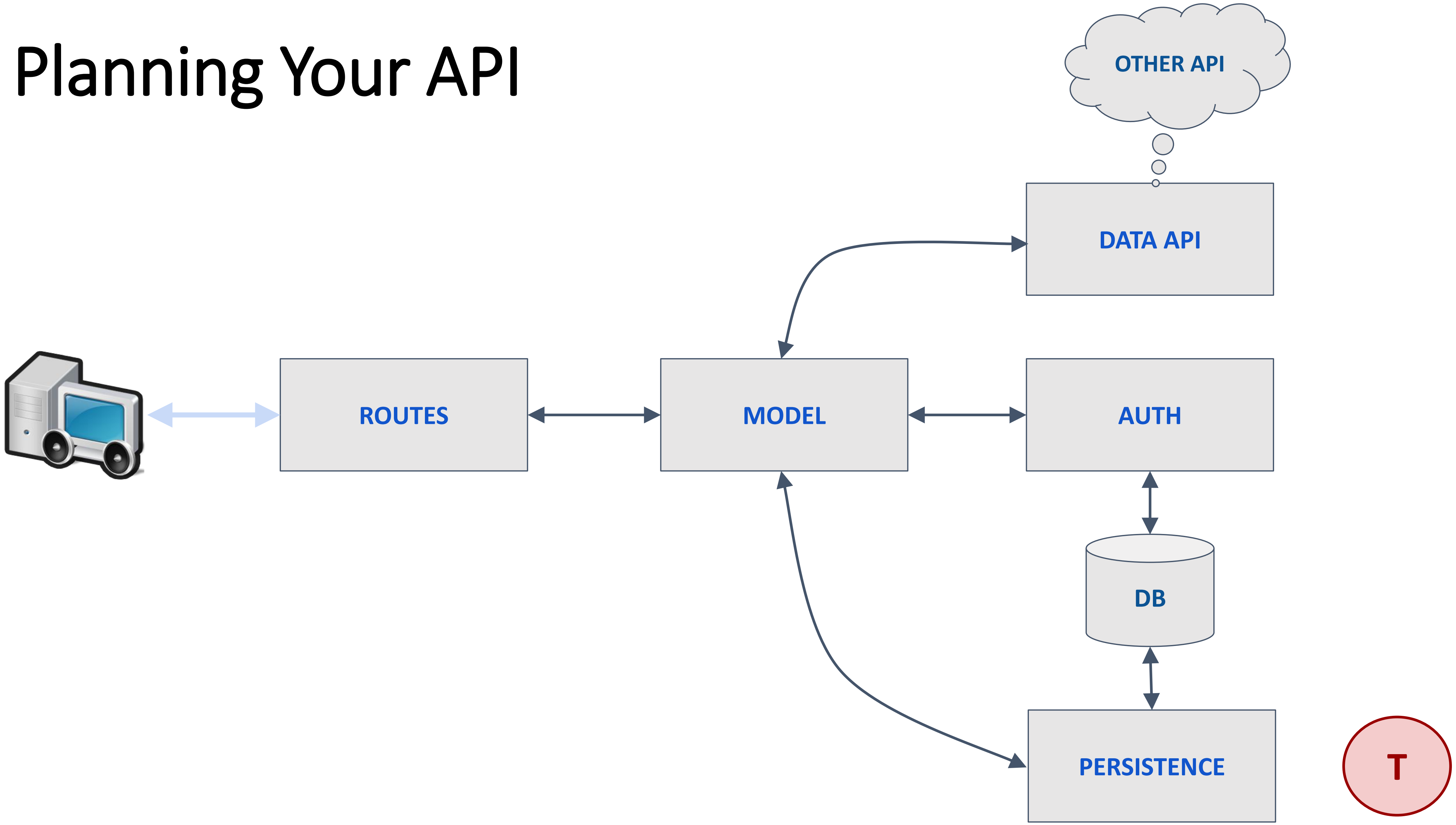
Web API Security

Mahmoud Awad

Web Architecture


Database Server
SQL (MySQL, Oracle..etc)
noSQL (MongoDB, CouchDB ..etc)
Graph (NeoDB)
Files system
Application
iOS APP
Android APP
Desktop…etc
API Server
NodeJS Server
Device
PC
Smart TV
Game Console
Mobile …etc
Browser
Chrome
Firefox
Safari…etc
Front-end Server
Bootstrap with Jquery, Angular, or Vue.JS…etc served by any HTTP server (Express NodeJS, Python, PHP..etc)

Planning Your API



Module Plan update

- Lectures & Lab plans were updated
- Lab 5.1 Solution is now live on Moodle

Assignment Assessment

- 45% Front-end
- 40% API
- 10% Persistent Storage
- 5% Video Demo **OR** Live App & API

304CEM Grading Rubric for Front-end (45%)

	0	1	2	3	4	5
Version Control 5%	No access given to the remote repository on GitLab.	Code access provided but does not match the code demonstrated.	Access provided to GitLab however there is no evidence of regular commits.	Access has been given to the code on GitLab with evidence of regular commits.	Evidence of regular commits over an extended period of time.	Commits over an extended period of time, demonstrating the use o branching and merging.
Responsive 5%	No responsive design	An attempt for a responsive design but it is not working properly.	Around half of the pages are responsive, or there are all responsive with major issues in each page	Most of the pages are responsive, with very few major issue with design such as wrong placement of large components	Fully responsive design, with minor issues, such as wrong placement of some small elements or extra shifts in some components	Fully responsive design UI
CSS 10%	No CSS used at all	Most of the CSS are working from the template. No real modification or use of its customised CSS	Half of the website is based on the original and customised CSS which heavily dependent on the template CSS	Most of the pages used original CSS, with major components throughout the pages used the template CSS	Well designed and original CSS, with some elements used from the template	Well designed, original CS creation, completely different from the standard template
Front-end code Architecture 5%	No architecture	An attempt for a front-end code architecture was made, but it is not working properly or it is mostly incomplete	Code architecture is not complete, with major flaws in some pages	Generally good architecture that needs improving in some places.	Good architecture with some minor flow in the design of the code	Clear architecture, clear separation of JS, CSS and contents
Front-end Coding (HTML & JS) 10%	No code supplied through GitLab	An attempt has been made to write code to implement some of the basic functionality although this may not be successful. No attempt at documentation.	Working code base showing the application of basic programming principles. Code may contain linting errors and warnings.	Demonstration of the usage of modularity to organise the code. Code contains linting warnings but no errors.	The code is modular. Code contains no linting errors or warnings with minor errors in the code	Code contains no linting errors or warnings.
Completeness 5%	Zero achievement	At attempt has been made to create front-end for the requested API, however they still need major work and improvements	Have of the back-end services have been utilised in the front-end	Most of the requested back-end APIs were used in the front-end, with some having minor or major issue	Most of the requested back-end APIs have a front-end pages ready to serve them	All requested backend APIs have been utilised in the front-end
Connecting to APIs 5%	No connecting to APIs from the front-end	An attempt for connecting and retrieving data from the APIs was done but it is mostly having issues in connecting	half of the pages connecting and retrieving data from the APIs with few having minor or major issues in connecting	Most of the pages connecting and retrieving data from the APIs with few having minor or major issues in connecting	All pages connecting and retrieving data from the API with few having minro or major issues in connecting	All pages connecting an retrieving data from the API

Example of Bad Frontend

- Excessive use of Labs source code templates
- No attempt to design properly
- Random layout of components
- No proper styles or colours
- Broken navigation
- No dynamic alternation of page
- Pure static pages
- Pages do not match in style
- No heading or footers
- Badly designed Home page
- No proper use of GIT

304CEM Grading Rubric for Backend API (40%)

	0	1	2	3	4	5
Version Control 5%	No access given to the remote repository on GitLab.	Code access provided but does not match the code demonstrated.	Access provided to GitLab however there is no evidence of regular commits.	Access has been given to the code on GitLab with evidence of regular commits.	Evidence of regular commits over an extended period of time.	Commits over an extended period of time, demonstrating the use of branching and merging.
Automated Testing 5%	No tests have been written or run.	Limited attempt at flawed tests.	An attempt has been made to write simple tests	Evidence of a limited number of tests written and run.	A range of tests showing how they contribute to code quality.	A full suite of automated tests ensuring full code coverage
API Design 10%	No API demonstrated in the screencast	An attempt has been made to implement a basic API however this does not work as expected	Simple functional API demonstrating a basic understanding of REST principles (resources, collections, methods and headers)	The API is fully functional and includes an authentication mechanism. The API demonstrates a good understanding of REST principles.	The API demonstrates user registration and authentication. It provides feedback for invalid requests through appropriate response codes and messages. Limited HATEOAS support.	Fully REST-compliant API that includes filtering and sorting and conditional GET requests. makes use of the full range of request and response headers Full HATEOAS implementation
Architecture 5%	No code supplied through GitLab	An attempt to write the API however it fails to work correctly.	All code for routing and business logic maintained in a single file	Code split into several files but overlap between routing and business logic.	Clear separation between routing and business logic code.	Clear separation between routing and business logic cod with no code duplication.
Coding 10%	No code supplied through GitLab	An attempt has been made to write code to implement some of the basic functionality although this may not be successful. No attempt at documentation.	Working code base showing the application of basic programming principles. Code may contain linting errors and warnings. No attempt at code documentation	Demonstration of the usage of modularity to organise the code. Code documentation is incomplete. Code contains linting warnings but no errors.	The code is modular and includes full exception-handling. Code is fully annotated and explained. Code contains no linting errors or warnings and is fully documented.	The API demonstrates a wide range of appropriate language constructs including clear modular structure. Code contains no linting errors or warnings and is fully documented.
Completeness 5%	No APIs were developed at all, or all of the created APIs are not working	Only few of the requested APIs were created, the exiting APIs may lack functionality	Half of the requested APIs were created and working properly, the existing APIs may not work or lack functionality	Most of the requested APIs were created, the existing APIs may not work or lack functionality	All requested API were created, with some of them may not working or lack functionality	All requested APIs were create and all working correctly

Example of bad Backend

- Rely entirely on labs code
- No authentications
- No unit testing
- Incorrect results
- Incomplete API
- Server crashes
- No server validation
- Bad Modules and code structures
- No proper documentation
- No proper use of GIT
- No status codes are sent or incorrect ones are sent

Assignment Update

- ~~GIT repositories~~
- ~~Plan DB, API and Frontend~~
- ~~Persistent Storage~~
- ~~Basic API with Node.JS~~
- ~~Layout contents with Bootstrap~~
- ~~Create Bootstrap forms~~
- Connecting Frontend with Backend (read/write)
- Dynamically change contents of frontend
- Full API methods
- API security & Authentications
- Unit Testing
- Documenting API

Connecting Front-end to API

- Frontend page purpose is to provide a user interface to interact with your application, this interaction will result or need data from persistent storage, only API server can communicate with DB server
- So front-end sends a request in the back ground using AJAx to ask API server to add a new user data
- But!!..

Error!!

- Failed to load http://localhost:8080/user/add: Response to preflight request doesn't pass
  access control check: No 'Access-Control-Allow-Origin' header is present on the requested
  resource. Origin 'http://localhost' is therefore not allowed access. The response had HTTP status code 405.

- 405 status code is Method not allowed
- So what happened?

HTTP Methods

HTTP Methods are the verbs that act on collections and elements.

CRUD operations require 4 methods.

GET POST PUT DELETE

Additional Methods

OPTIONS
HEAD
PATCH

Cross Origin Resource Sharing (CORS)

- By default API server only allows requests from same domain to post data to it.
- Your frontend page sends an OPTIONS method to the API server to sniff if it can post to it
- API server replies with 405 Method not allowed.
- We need to allow SPI server to accepts connections from other web domains by leverage the Cross Origin Resource Sharing settings

- This can be done in the header

```
'Access-Control-Allow-Origin : http://whatotherdomain.com’
'Access-Control-Allow-Methods: GET, PUT, POST, DELETE, OPTIONS'
'Access-Control-Max-Age: 1000'
'Access-Control-Allow-Headers: Content-Type, Authorization, X-Requested-With'
```

Allows CORS in Restify Package

```
const corsMiddleware = require('restify-cors-middleware')

const cors = corsMiddleware({
  preflightMaxAge: 5, //Optional
  origins: ['*'],
  allowHeaders: ['API-Token'],
  exposeHeaders: ['API-Token-Expiry']
})

//create the restify module
const server = restify.createServer()

server.pre(cors.preflight)
server.use(cors.actual)
```

HTTP Authentication

- Remember that iF you set CORS to *, that means any device or domain will be able to perform any methods on your API.
- In all cases you should protect your API by authenticating your users.
- There are different methods for HTTP authentication, we will discuss:
  - Basic Auth
  - Cookies
  - JWT
  - HMAC
  - OAuth 2.0

REST Principle

- All REST interactions are ***stateless***. That is, each ***request contains all of the information necessary for a connector to understand the request, independent of any requests that may have preceded it***.

Basic Access Authentication

Client includes an `Authorization` header

Username and password combined with a colon

```
testuser:p455w0rd
```

This string is encoded using the RFC2045-MIME variant of base64

```
Printf testuser:p455w0rd | base64
  dGVzdHVzZXI6cDQ1NXcwcmQ=
```

The result is sent in the header prepended with `Basic`

```
Authorization: Basic dGVzdHVzZXI6cDQ1NXcwcmQ=
```

Basic Access Authentication

- it doesn't require cookies, sessions or anything else
- the client has to send the Authorization header along with every request it makes
- the username and password are sent with every request, potentially exposing them - even if sent via a secure connection
- If a website uses weak encryption, or an attacker can break it, the usernames and passwords will be exposed immediately
- there is no way to log out the user using Basic auth
- expiration of credentials is not trivial - you have to ask the user to change password to do so

Cookies

- When a server receives an HTTP request in the response, it can send a Set-Cookie header.
- The browser puts it into a cookie jar, and the cookie will be sent along with every request made to the same origin in the Cookie HTTP header.

Cookies

- Always use the HttpOnly flag when setting cookies. This way they won't show up in document.cookies
- Use signed cookies, so that a server can tell if a cookie was modified by the client.
- Incompatibile with REST Principle

Hash-based Message Authentication Code (HMAC)

- A hash code of current message is added to every request.
- Hash code calculated by *cryptographic hash function* in combination with a *secret cryptographic key*.
- *Secret cryptographic key* should be provided by server to client as resource, and client uses it to calculate hash code for every request.

Hash-based Message Authentication Code (HMAC)

- If client knows the secret key, then it's ready to operate with resources.
- Otherwise he will be temporarily redirected (status code 307 Temporary Redirect) to authorize and to get secret key, and then redirected back to the original resource.

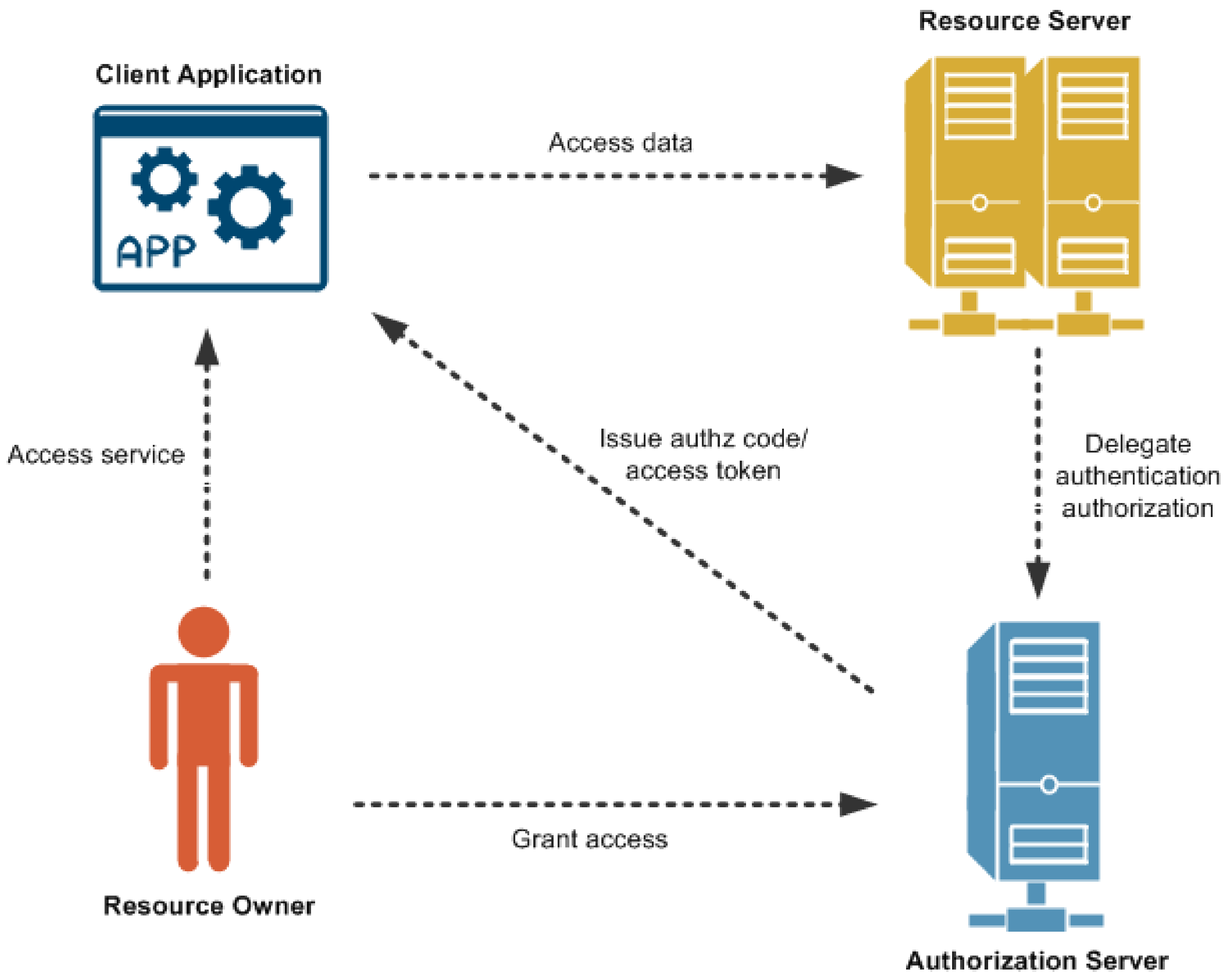
JSON Web Tokens (JWT)

- Consists of three parts:
  - Header, containing the type of the token and the hashing algorithm
  - Payload, containing the claims
  - Signature
- If you are writing APIs for native mobile applications or Single Page Applications (SPAs), then JWT can be a good fit.
- to use JWT in the browser you have to store it in either LocalStorage or SessionStorage, which can lead to XSS attacks
- Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites

OAuth 2.0

- Industry Standard protocol for authorization
- In the traditional client-server authentication model, the client requests an access-restricted resource on the server by authenticating with the server using the resource owner’s credentials
- In order to provide third-party applications access to restricted resources, the resource owner shares its credentials with the third party.
- Instead of using the resource owner's credentials to access protected resources, the client obtains an access token
- Access tokens are issued to third-party clients by an authorization server with the approval of the resource owner.

OAuth 2.0


Resource Server
Client Application
APP
Access data
Access service
Issue authz code/ access token
Delegate authentication authorization
Grant access
Resource Owner
Authorization Server

How to do Logins?

1. Create a Login/logout API like: /api/v1/login and api/v1/logout
2. In these Login and Logout APIs, user needs to provide username/password and get verified.
3. If successfully, then The outcome is a token that is sent back to the client (web, mobile, whatever)
4. From this point onwards, all subsequent calls made by your client will include this token
5. For example user called /api/v1/findUser then the first thing this API code will do is to check for the token ("is this user authenticated?")
6. If the answer comes back as NO, then you throw a HTTP 401 Status back at the client. Let them handle it.
7. If the answer is YES, then proceed to return the requested User

Storing Login Password

- Remember, never store the user password in your persistent storage
- Store a one-way encrypted hash of the user password
- To authenticate user, compare the user sent password to the hash is stored in the database

Hashing Passwords in Node.JS

```
const bcrypt = require('bcrypt’);

bcrypt.hash('myPassword', 10, function(err, hash) {
        // Store hash in database
});

bcrypt.compare('somePassword', hash, function(err, res) {
        if(res) {
                // Passwords match
         }
         else {
                // Passwords don't match
         }
});
```

Logging API Requests

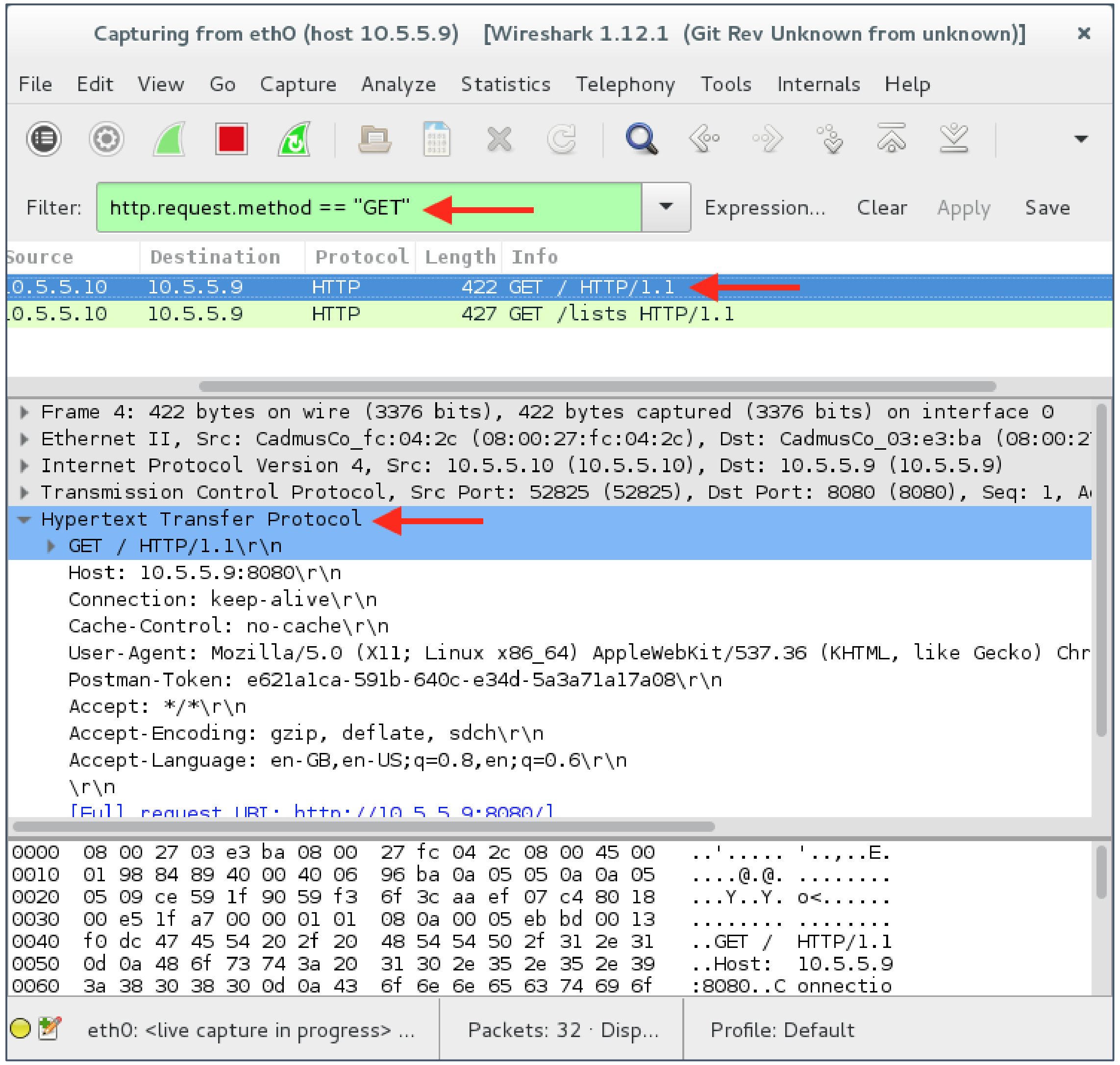
Need to keep logs of all API requests:

- Why is this important?
- What useful data can be found in the request?

The Problem with HTTP

- All the APIs built so far run over HTTP
- This is an unencrypted connection
- Packets can be intercepted and read using a packet sniffer

Wireshark


Capturing from eth0 (host 10.5.5.9) [Wireshark 1.12.1 (Git Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

Filter: http.request.method == "GET"

```
Source      Destination  Protocol  Length  Info
.0.5.5.10   10.5.5.9     HTTP       422  GET / HTTP/1.1
.0.5.5.10   10.5.5.9     HTTP       427  GET /lists HTTP/1.1
```

```
▸ Frame 4: 422 bytes on wire (3376 bits), 422 bytes captured (3376 bits) on interface 0
▸ Ethernet II, Src: CadmusCo_fc:04:2c (08:00:27:fc:04:2c), Dst: CadmusCo_03:e3:ba (08:00:2
▸ Internet Protocol Version 4, Src: 10.5.5.10 (10.5.5.10), Dst: 10.5.5.9 (10.5.5.9)
▸ Transmission Control Protocol, Src Port: 52825 (52825), Dst Port: 8080 (8080), Seq: 1, A
▾ Hypertext Transfer Protocol
  ▸ GET / HTTP/1.1\r\n
    Host: 10.5.5.9:8080\r\n
    Connection: keep-alive\r\n
    Cache-Control: no-cache\r\n
    User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chr
    Postman-Token: e621a1ca-591b-640c-e34d-5a3a71a17a08\r\n
    Accept: */*\r\n
    Accept-Encoding: gzip, deflate, sdch\r\n
    Accept-Language: en-GB,en-US;q=0.8,en;q=0.6\r\n
    \r\n
    [Full request URI: http://10.5.5.9:8080/]
```

```
0000  08 00 27 03 e3 ba 08 00  27 fc 04 2c 08 00 45 00   ..'..... '..,..E.
0010  01 98 84 89 40 00 40 06  96 ba 0a 05 05 0a 0a 05   ....@.@. ........
0020  05 09 ce 59 1f 90 59 f3  6f 3c aa ef 07 c4 80 18   ...Y..Y. o<......
0030  00 e5 1f a7 00 00 01 01  08 0a 00 05 eb bd 00 13   ........ ........
0040  f0 dc 47 45 54 20 2f 20  48 54 54 50 2f 31 2e 31   ..GET /  HTTP/1.1
0050  0d 0a 48 6f 73 74 3a 20  31 30 2e 35 2e 35 2e 39   ..Host:  10.5.5.9
0060  3a 38 30 38 30 0d 0a 43  6f 6e 6e 65 63 74 69 6f   :8080..C onnectio
```

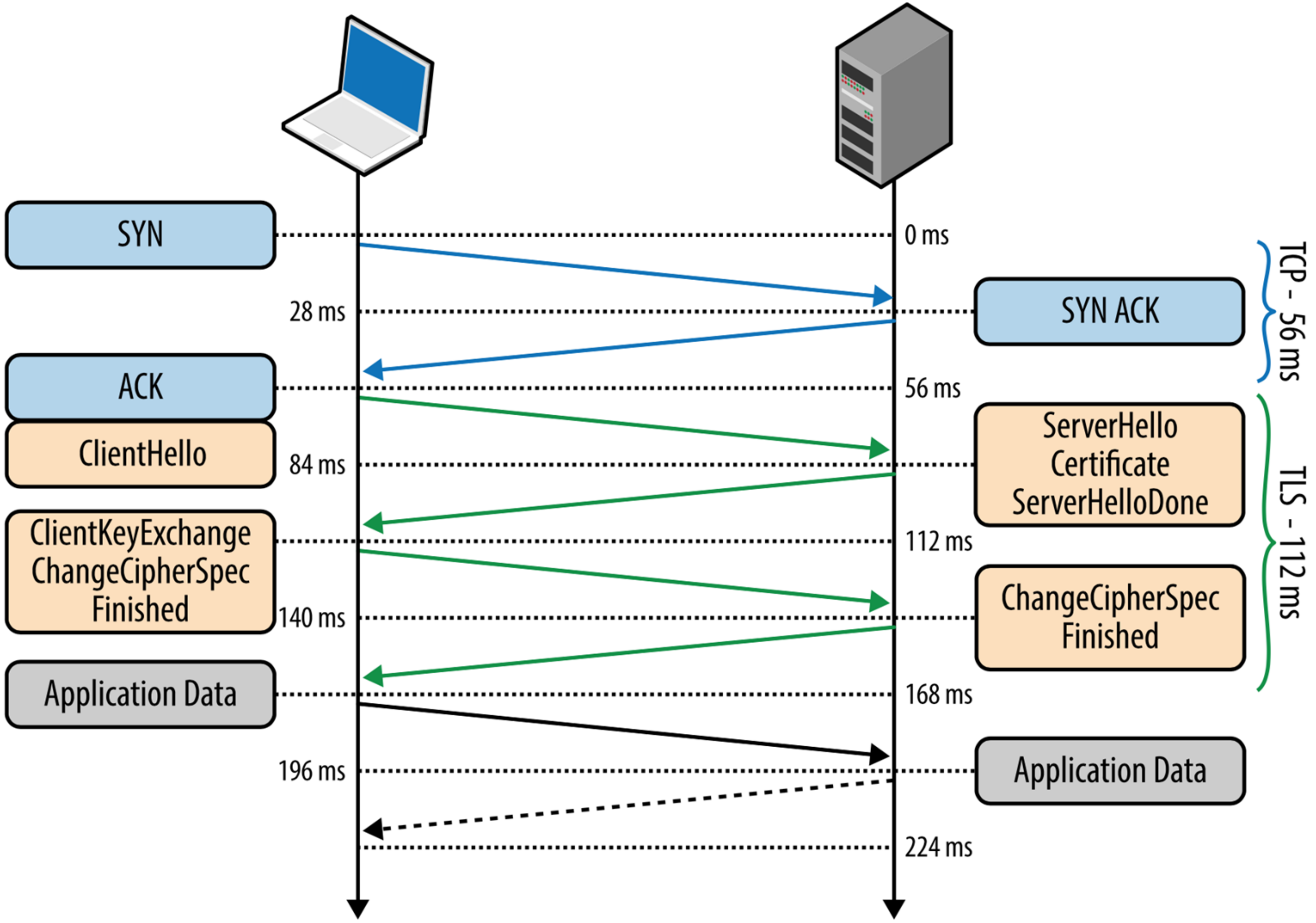
eth0: <live capture in progress> ... Packets: 32 · Disp... Profile: Default

The Solution

- All data packets need to be encrypted
- Use an asymmetric encryption protocol called (Transport Layer Security) TLS 1.2
- TLS is responsible for the authentication and key exchange necessary to establish or resume secure sessions.
- TLS predecessor is SSL (Secured Socket Layer)

The Process

- https means client will connect to port 443 unless specified
- First byte out of our browser makes a handshake request
- Server sends a response and agrees to the request to use TLS
- The client checks server certificate expiry and makes sure its public key is authorized for exchanging secret keys
- TLS is used to exchange a shared (symmetric) one time encryption key that is then used to encrypt and decrypt data packets


SYN
0 ms
28 ms
SYN ACK
TCP - 56 ms
ACK
56 ms
ClientHello
84 ms
ServerHello
Certificate
ServerHelloDone
TLS - 112 ms
ClientKeyExchange
ChangeCipherSpec
Finished
112 ms
140 ms
ChangeCipherSpec
Finished
Application Data
168 ms
196 ms
Application Data
224 ms

Generating a TLS 1.2 Certificate

- Normally certificated issued by a 'trusted authority'
- Tied to a specific domain name
- Requires annual payment
- For development purposes we can generate our own

Known as a 'self-signed' certificate

- Need to generate a self-signed certificate for our Certificate Authority
- Means that this CA is totally trusted
- Its certificate will serve as the root certificate
- Run the following command to generate the self-signed certificate for the CA

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out
  cert.pem -days 365 -nodes
```

Key and Certificate

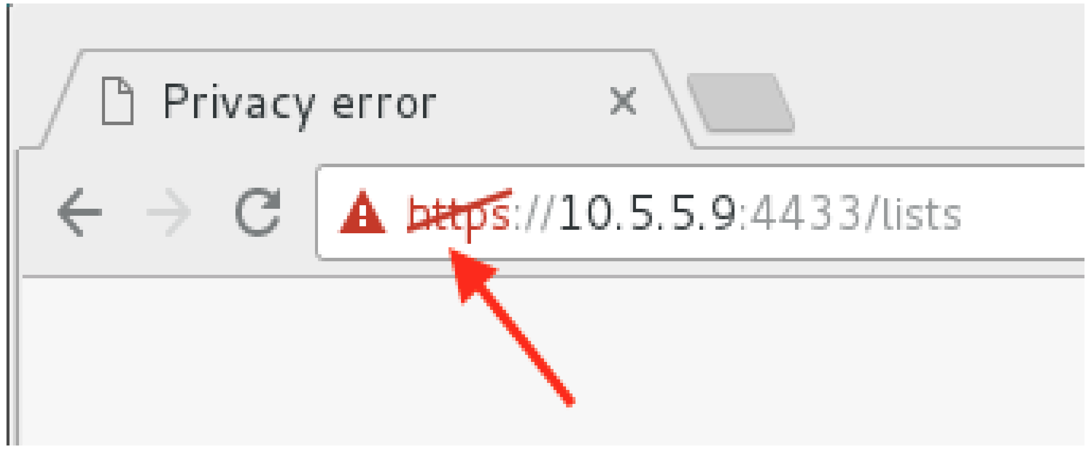
- The Key
    - Contains the Certificate Authority's private key
        - `key.pem`
- The Certificate
    - Contains the public-key certificate
        - `cert.pem`

Loading the Key and Cert

```
const fs = require('fs')
const httpsOptions = {
 key: fs.readFileSync('./key.pem'),
 certificate: fs.readFileSync('./cert.pem')
}
const server = restify.createServer(httpsOptions)
```

Browser Warning

- Browser will warn user if pages that contains passwords or payment are not using TLS or SSL
- To ensure that the Not Secure warning is not displayed for your pages, you must ensure that all forms containing <input type=password> elements and any inputs detected as credit card fields are present only on secure origins.


Privacy error
~~https~~://10.5.5.9:4433/lists