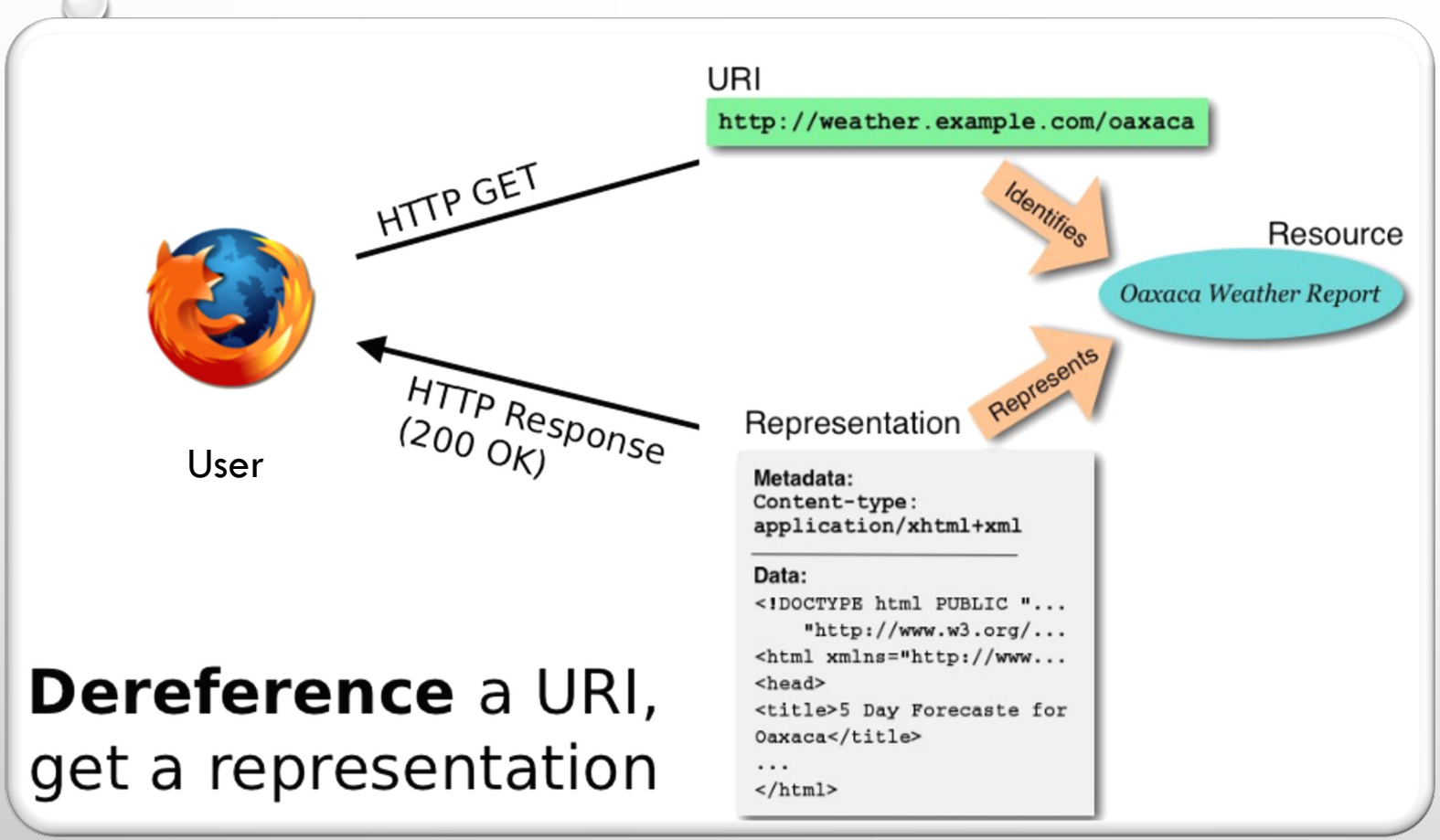
LECTURE01: WEB ARCHITECTURE, HTTP, AND GIT

CS418/518: Web programming

By Dr. Jian Wu

Courtesy: presentation slides from Dr. Justin Brunelle

WEB ARCHITECTURE



HTTP REQUEST

```
$ curl -i -v http://www.odu.edu
*   Trying 128.82.112.29:80...
* Connected to www.odu.edu (128.82.112.29) port 80 (#0)
> GET / HTTP/1.1
> Host: www.odu.edu
> User-Agent: curl/7.79.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 301 Moved Permanently
HTTP/1.0 301 Moved Permanently
< Location: https://www.odu.edu/
Location: https://www.odu.edu/
< Server: BigIP
Server: BigIP
* HTTP/1.0 connection set to keep alive!
< Connection: Keep-Alive
Connection: Keep-Alive
< Content-Length: 0
Content-Length: 0

<
* Connection #0 to host www.odu.edu left intact
```

This tell me where I should go.

HTTPS REQUEST

```
$ curl -i -v https://www.odu.edu
* Rebuilt URL to: https://www.odu.edu/
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0*   Trying 128.82.112.29...
* TCP_NODELAY set
* Connected to www.odu.edu (128.82.112.29) port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* Cipher selection: ALL:!EXPORT:!EXPORT40:!EXPORT56:!aNULL:!LOW:!RC4:@STRENGTH
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/cert.pem
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
} [217 bytes data]
* TLSv1.2 (IN), TLS handshake, Server hello (2):
{ [87 bytes data]
* TLSv1.2 (IN), TLS handshake, Certificate (11):
{ [5018 bytes data]
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
{ [333 bytes data]
* TLSv1.2 (IN), TLS handshake, Server finished (14):
{ [4 bytes data]
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
} [70 bytes data]
* TLSv1.2 (OUT), TLS change cipher, Client hello (1):
} [1 bytes data]
* TLSv1.2 (OUT), TLS handshake, Finished (20):
} [16 bytes data]
* TLSv1.2 (IN), TLS change cipher, Client hello (1):
{ [1 bytes data]
* TLSv1.2 (IN), TLS handshake, Finished (20):
{ [16 bytes data]
* SSL connection using TLSv1.2 / ECDHE-RSA-AES128-GCM-SHA256
* ALPN, server did not agree to a protocol
* Server certificate:
*  subject: serialNumber=232982; jurisdictionCountryName=US; jurisdictionStateOrProvinceName=Virginia; businessCategory=Government Entity; C=US; postalCode=23529; ST=VA; L=Norfolk; street=4600
Elkhorn Ave; O=Old Dominion University; OU=ITS; OU=COMODO EV Multi-Domain SSL; CN=www.odu.edu
*  start date: Jun  5 00:00:00 2019 GMT
*  expire date: Jun  4 23:59:59 2021 GMT
*  subjectAltName: host "www.odu.edu" matched cert's "www.odu.edu"
*  issuer: C=GB; ST=Greater Manchester; L=Salford; O=COMODO CA Limited; CN=COMODO RSA Extended Validation Secure Server CA
*  SSL certificate verify ok.
> GET / HTTP/1.1
> Host: www.odu.edu
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 19 Jan 2021 03:16:47 GMT
< Server: Apache/2.4.6 (Red Hat Enterprise Linux)
< Vary: Host, Accept-Encoding
< Accept-Ranges: bytes
< Connection: close
< Transfer-Encoding: chunked
< Content-Type: text/html; charset=UTF-8
< Set-Cookie: BIGipServerWEB_HTTPS_PROD.app~WEB_HTTPS_PROD_pool_int=rd741o00000000000000000000ffff8052619fo80; path=/; Httponly; Secure
<
{ [575 bytes data]
100 94661    0 94661    0     0   320k      0 --:--:-- --:--:-- --:--:--  320k
* Closing connection 0
* TLSv1.2 (OUT), TLS alert, Client hello (1):
} [2 bytes data]
```

This is too much!
Let's look at the most important stuff!

HTTPS REQUEST

```
$ curl -i -v https://www.odu.edu
* Rebuilt URL to: https://www.odu.edu/
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0*   Trying 128.82.112.29...
* TCP_NODELAY set
* Connected to www.odu.edu (128.82.112.29) port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
*  start date: Jun  5 00:00:00 2019 GMT
*  expire date: Jun  4 23:59:59 2021 GMT
*  subjectAltName: host "www.odu.edu" matched cert's "www.odu.edu"
*  issuer: C=GB; ST=Greater Manchester; L=Salford; O=COMODO CA Limited; CN=COMODO RSA Extended Validation Secure
Server CA
*  SSL certificate verify ok.
> GET / HTTP/1.1
> Host: www.odu.edu
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 19 Jan 2021 03:16:47 GMT
< Server: Apache/2.4.6 (Red Hat Enterprise Linux)
< Vary: Host, Accept-Encoding
< Accept-Ranges: bytes
< Connection: close
< Transfer-Encoding: chunked
< Content-Type: text/html; charset=UTF-8
< Set-Cookie: BIGipServerWEB_HTTPS_PROD.app~WEB_HTTPS_PROD_pool_int=rd741o00000000000000000000ffff8052619fo80;
path=/; Httponly; Secure
<
{ [575 bytes data]
100 94661    0 94661    0     0   320k      0 --:--:-- --:--:-- --:--:--  320k
* Closing connection 0
* TLSv1.2 (OUT), TLS alert, Client hello (1):
} [2 bytes data]
```

connection successful!

HTTP REQUEST AND RESPONSE

```
> GET / HTTP/1.1
> Host: www.odu.edu
> User-Agent: curl/7.79.1
> Accept: */*
```

request

```
< HTTP/1.1 200 OK
< Date: Mon, 29 Aug 2022 17:37:00 GMT
< Server: Apache/2.4.6 (Red Hat Enterprise Linux)
< Vary: Host, Accept-Encoding
< Accept-Ranges: bytes
< Connection: close
< Transfer-Encoding: chunked
< Content-Type: text/html; charset=UTF-8
```

Chunked transfer encoding (CTE) is a mechanism in which the encoder sends data to the player in series of chunks. The player doesn't have to wa until the complete segment is available. CTE is available in HTTP 1.1.

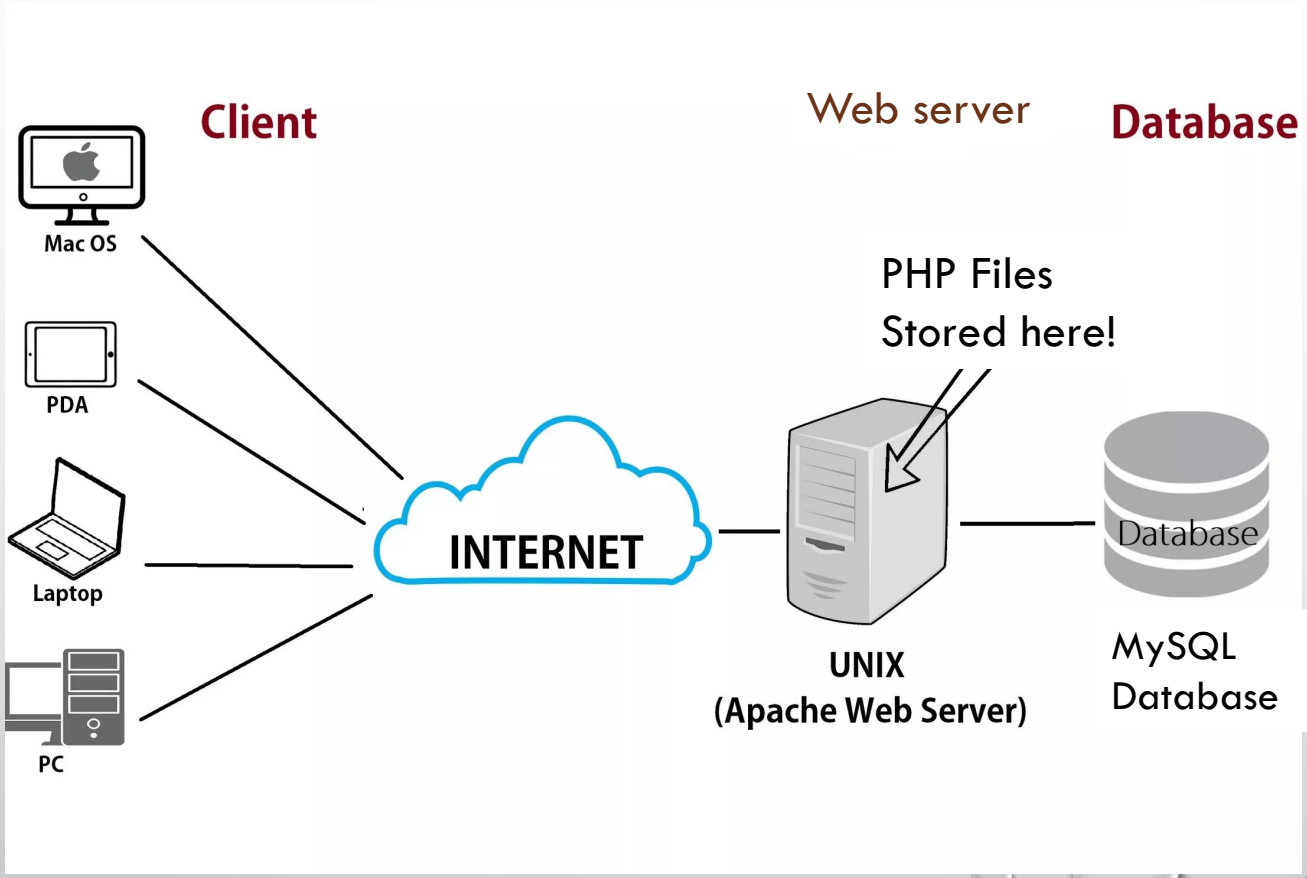
USER AGENT

- Web browser, e.g., Mozilla, Chrome
- Command line, e.g., curl, wget
- Anything used to navigate the web, e.g., Googlebot
- Refer to this page to see how to view user-agent on the Google chrome browser (tip: you can disguise yourself by changing it)

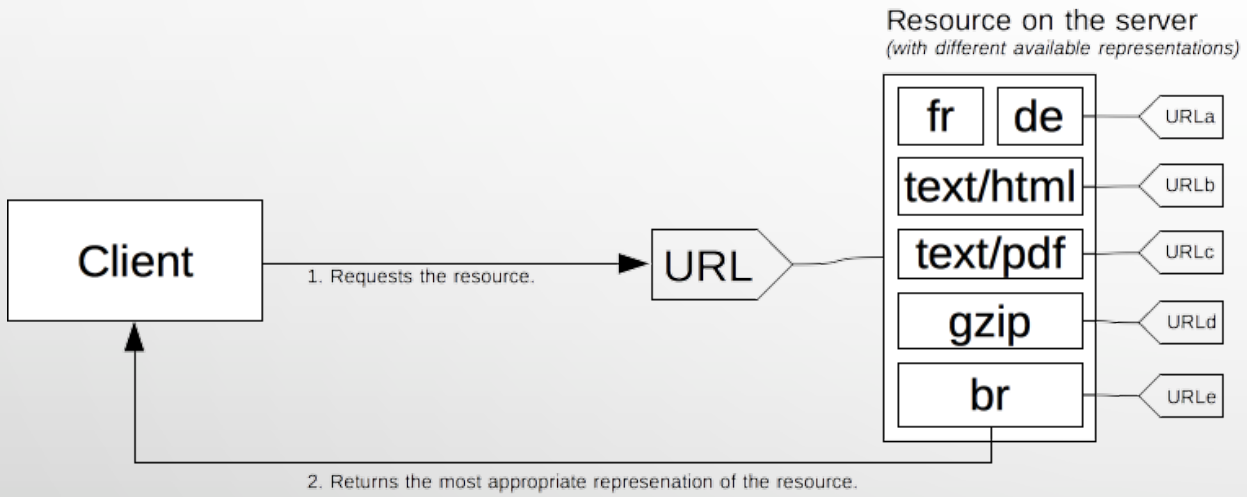
```
> GET / HTTP/1.1
> Host: www.odu.edu
> User-Agent: curl/7.79.1>
> Accept: */*
```

WEB SERVER

- Handle requests from clients



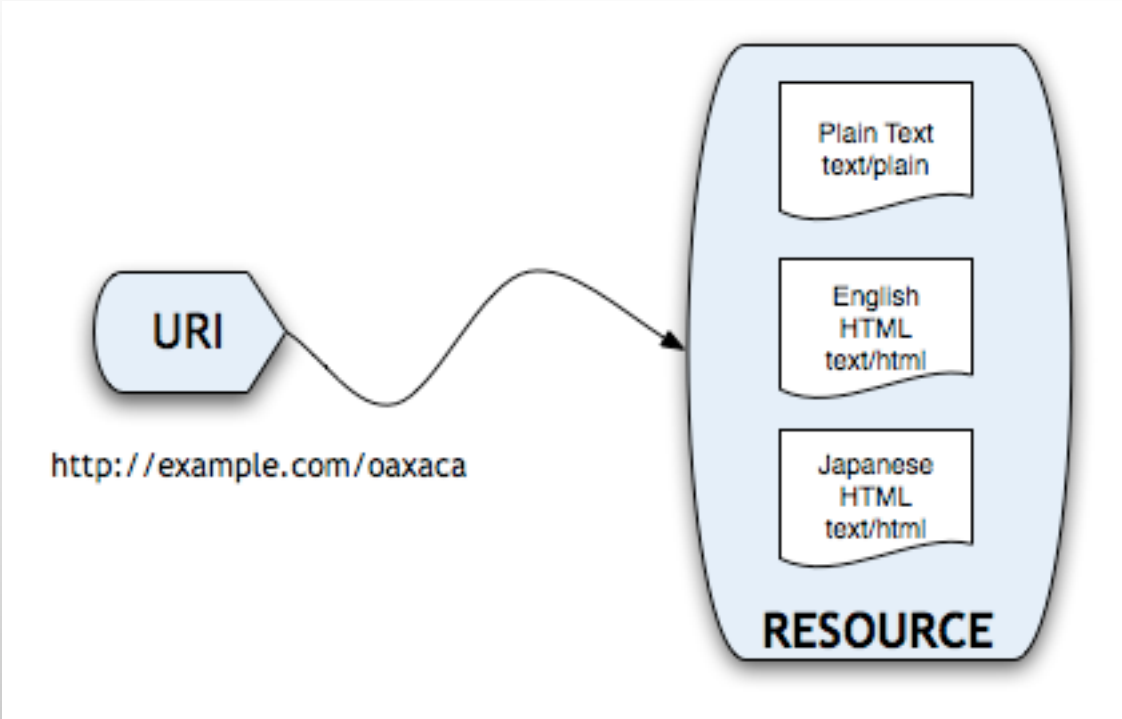
CONTENT NEGOTIATION



In HTTP, ***content negotiation*** is the mechanism that is used for serving different representations of a resource at the same URI, so that the user agent can specify which is best suited for the user (for example, which language of a document, which image format, or which content encoding).

The HTTP/1.1 standard defines list of the standard headers that start server-driven negotiation (Accept, Accept-Charset, Accept-Encoding, Accept-Language).

CONTENT NEGOTIATION EXAMPLES



Content Negotiation is a complex-sounding term for what is a rather simple mechanism.

Format Negotiation

Language Negotiation

Fore more information on how content negotiation works with PHP, here is an excellent blogpost: https://www.w3.org/blog/2006/02/content-negotiation/

IDENTIFIERS

- Uniform Resource Identifier (URI): it is an identifier, not the file name
  - http://www.ietf.org/rfc/rfc2396.txt: identifies a file on the web
  - news:comp.infosystems.www.servers.unix: identifies a news site
  - http://foo.com/page.html#section2: identifies a content block on a webpage
- Uniform Resource Locator (URL) refers to the **location** of a particular file.
  - http://foo.com/page.html
- Uniform Resource Name (URN): A Uniform Resource Name (URN) is a URI that identifies a resource by name in a particular namespace.

URN

The **International Standard Book Number** (**ISBN**) is a numeric commercial book identifier which is intended to be unique. Publishers purchase ISBNs from an affiliate of the International ISBN Agency.

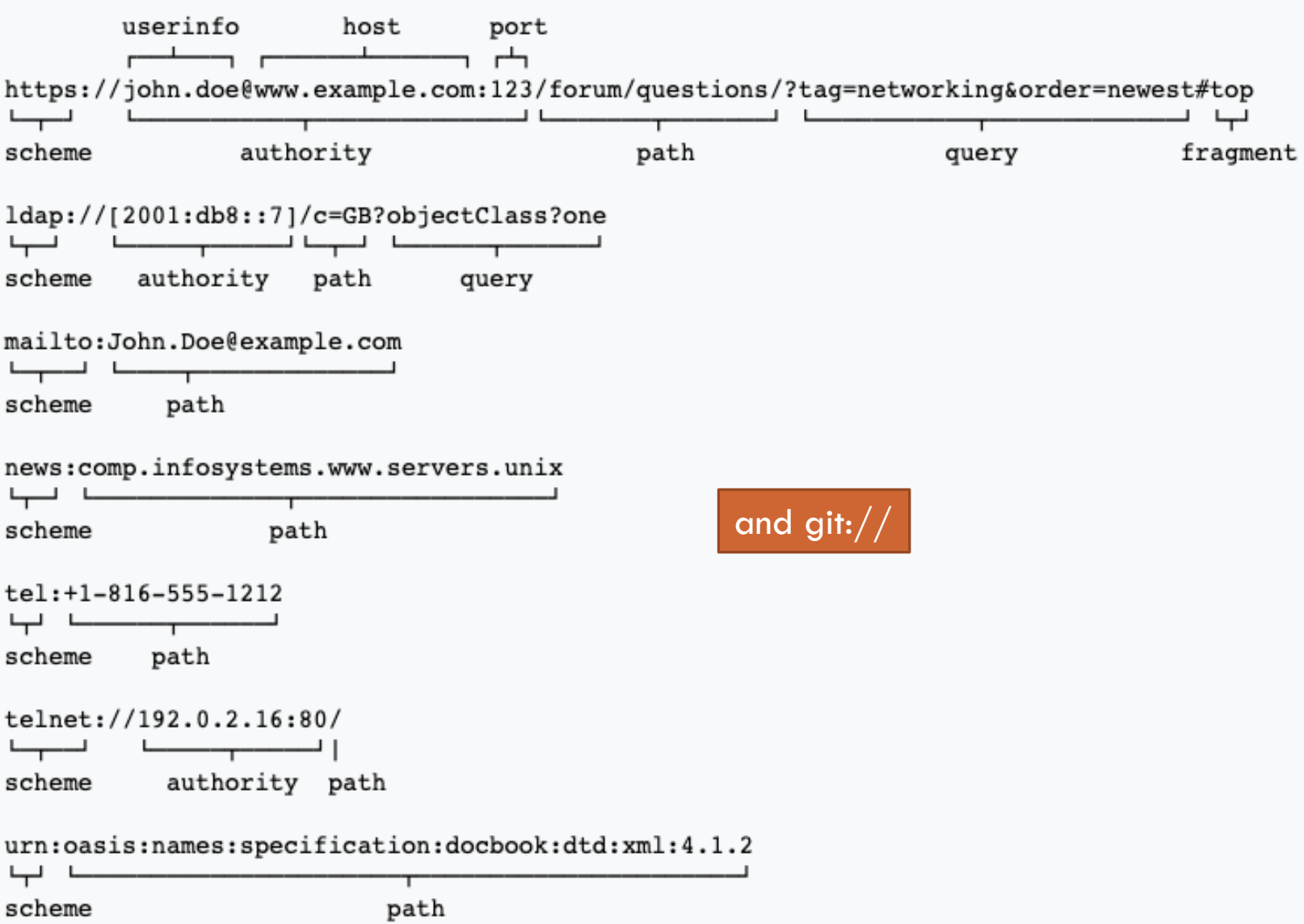
International Standard Book Number (ISBN)

SBN 0-486-27557-4

Romeo and Juliet is a tragedy written by William Shakespeare early in his career about two young star-crossed lovers whose deaths ultimately reconcile their feuding families. It was among Shakespeare's most popular plays during his lifetime and along with *Hamlet*, is one of his most

URN: urn:isbn:0-486-27557-4
However, it does not tell where to find it.

URI VS. URL



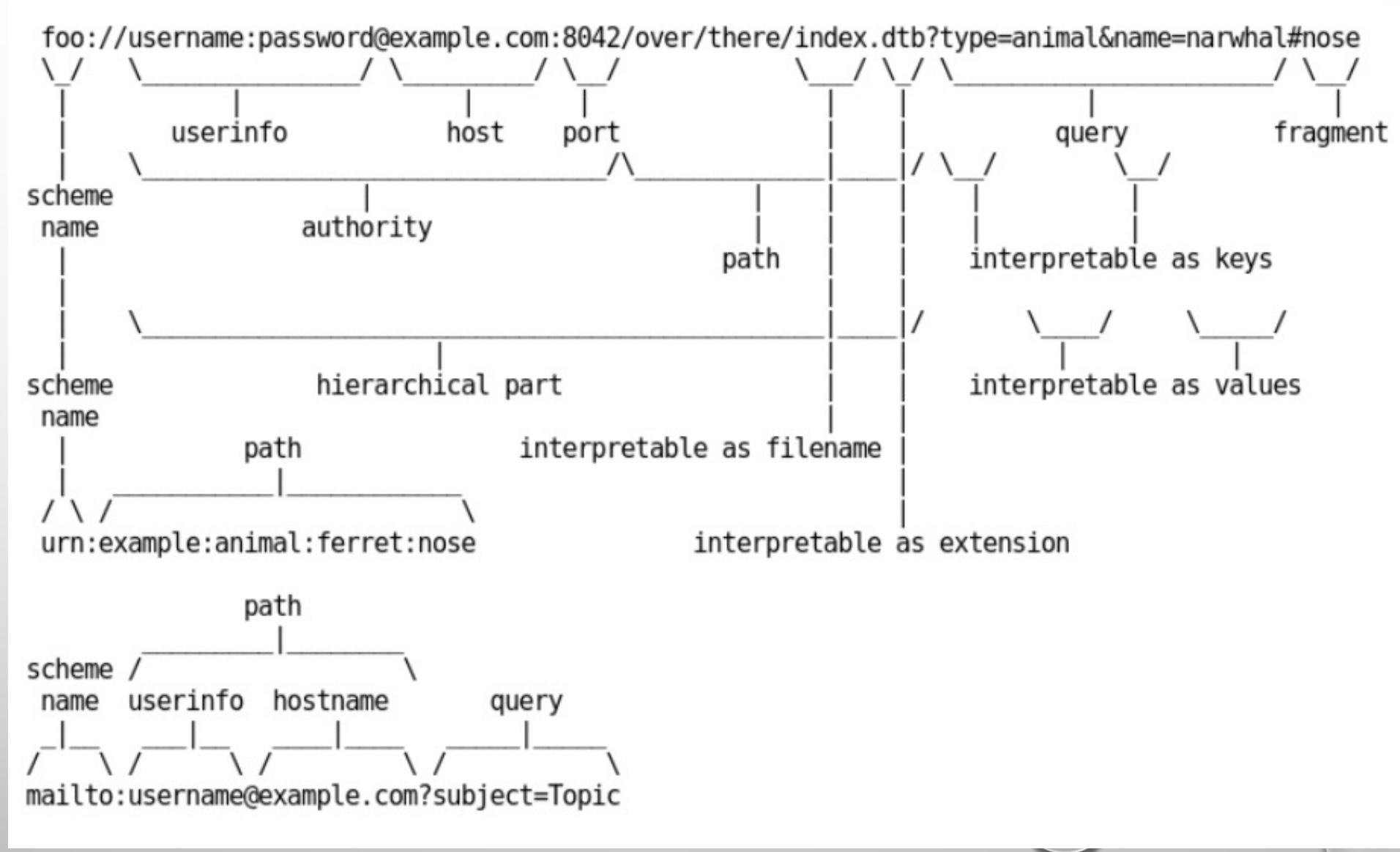
and git://

In technical publications, a URL is simply a URI that happens to point to a resource over a network.

In non-technical context and in software for the WWW, the word “URL” (a.k.a. web address) remains widely used.

They are often used interchangeably.

URI SCHEME IN FINE GRANULARITY



COMMON RESPONSE CODE

- 200 OK -- The request has succeeded. The information returned with the response is dependent on the method used in the request
- 301 Moved Permanently -- The requested resource has been assigned a new permanent URI and any future references to this resource SHOULD use one of the returned URIs
- 404 Not Found -- The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent. The 410 (Gone) status code SHOULD be used if the server knows...that an old resource is permanently unavailable and has no forwarding address
- 405 Method Not Allowed -- The method specified in the Request-Line is not allowed for the resource identified by the Request-URI.
- For a complete list: https://www.rfc-editor.org/rfc/rfc9110.html#name-status-codes

TYPES OF HTTP RESPONSES

1. Informational responses (100–199)
2. Successful responses (200–299)
3. Redirects (300–399)
4. Client errors (400–499)
5. Server errors (500–599)

```
< HTTP/1.1 200 OK
```

For more examples please refer to :
https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

and

https://www.tutorialspoint.com/http/http_responses.htm

STRUCTURES OF HTTP RESPONSES

- A Status-line:
  - Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
- Zero or more header (General | Response | Entity) fields followed by CRLF
- An empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the header fields
- Optionally a message-body

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
```

```
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

HTTP HEADER FIELDS

- There are at least 39 common standard request fields and 15 common non-standard request fields
- There are at least 41 common standard response fields and 10 common non-standard response fields
- Back to the example we had

```
< Date: Mon, 29 Aug 2022 17:37:00 GMT
< Server: Apache/2.4.6 (Red Hat Enterprise Linux)
< Vary: Host, Accept-Encoding
< Accept-Ranges: bytes
< Connection: close
< Transfer-Encoding: chunked
< Content-Type: text/html; charset=UTF-8
```

For a complete list, see
https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

HTTP METHODS

```
> GET / HTTP/1.1
```

- There are at least 9 HTTP methods
- GET, HEAD (covered in detail later)
- POST: Frequently used for passing credentials
- TRACE: What methods are defined on this URI?
- DELETE: Rarely supported for most URIs
- PUT: Rarely supported. Equivalent to Unix
  $ echo "hello world" > temp.txt

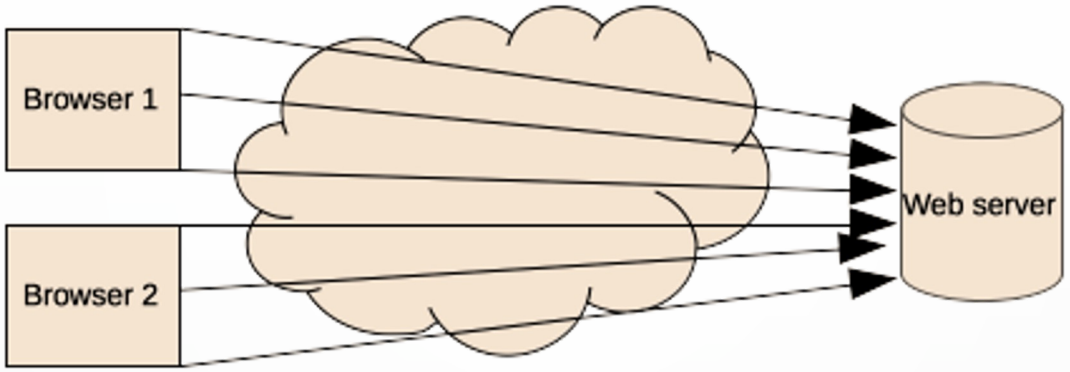
1. CONNECT
2. DELETE
3. GET
4. HEAD
5. OPTIONS
6. PATCH
7. POST
8. PUT
9. TRACE

GET

- **GET is used to request data from a specified resource.**
- **GET is one of the most common HTTP methods. In the example below, the user is requesting data satisfying two conditions: name1=value1 and name2=value2.**
  `/test/demo_form.php?name1=value1&name2=value2`
  - GET requests can be cached (different types of caching: https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching, https://developers.google.com/web/fundamentals/performance/get-started/httpcaching-6)
  - GET requests remain in the browser history
  - GET requests can be bookmarked
  - GET requests should never be used when dealing with sensitive data
  - GET requests have length restrictions (2048 characters, see this answer on Stackoverflow)
  - GET requests is only used to request data (not modify)

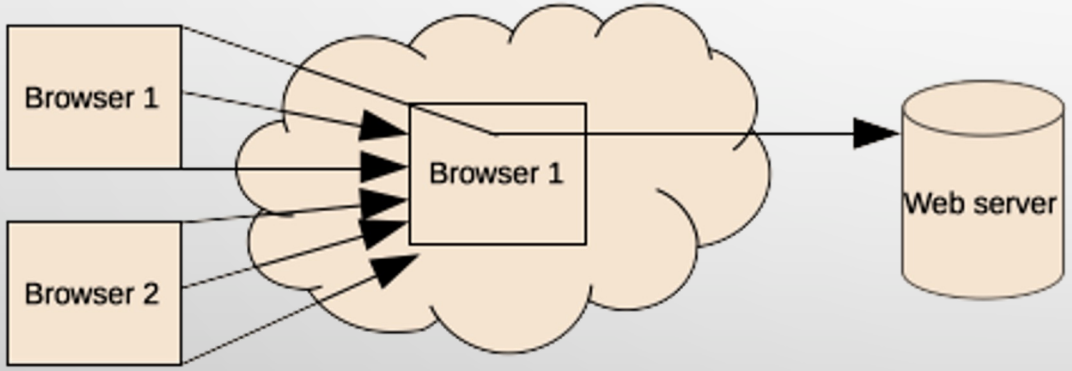
http caching

No cache



All identical requests are
going through to the server.

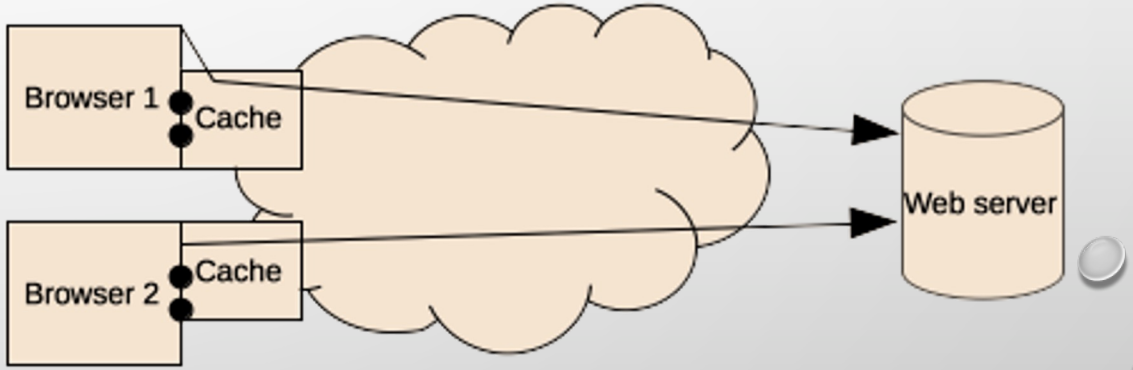
Shared cache



The first request is going through.

Subsequent identical requests are served
by the shared cache.
(more efficient)

Local (private) cache



The first request of each client is going through.

Subsequent identical requests are not even sent, but
served by the local cach.
(most efficient, except for first requests)

HEAD

The HEAD method asks for a response identical to that of a GET request, but without the response body.

You may try different methods at: https://reqbin.com/
ReqBin is a free, online HTTP/REST/SOAP API client.

POST

POST is used to send data to a server to create/update a resource.

-d, --data <data> HTTP POST data
-F, --form <name=content> Specify multipart MIME data
-H, --header <header/@file> Pass custom header(s) to server
-X, --request <command> Specify request command to use
For more options of curl, check the curl man page.

- POST requests are never cached
- POST requests do not remain in the browser history
- POST requests cannot be bookmarked
- POST requests have no restrictions on data length

Examples:

To Post with data

```
$ curl -d "username=mkyong&password=abc" http://localhost:8080/api/login/
```

To post with a file

```
$ curl -F file=@"path/to/data.txt" http://localhost:8080/api/upload/
```

To post with JSON data

```
$ curl -H "Content-Type: application/json" -X POST -d
'{"username":"mkyong","password":"abc"}' http://localhost:8080/api/login/Copy
```

GET VS POST

	GET	POST
How data is applied	In the URL	In the message body
Data type	Only ASCII characters	No restrictions, even binary
Security	Less secure	Safer as parameters are not stored in browser history of web server logs
Restrictions on data length	Limited to URL length: usually <2048 characters	No restrictions
Usability	Should not be used when sending sensitive information (e.g., password)	POST should be used
Visability	Displayed in the address bar	Not displayed on in the address bar

HOW TO SEND HTTP REQUESTS ON A COMMAND LINE

On Linux, you may use wget or curl commands to send HTTP requests.

Examples:

$ wget -S -O - http://www.google.com

$ curl -i http://www.google.com

How to use wget:

https://www.digitalocean.com/community/tutorials/how-to-use-wget-to-download-files-and-interact-with-rest-apis

How to use curl:

https://curl.se/docs/httpscripting.html

A MORE COMPLICATED EXAMPLE

curl -d "param1=value1&param2=value2" -H "Content-Type: application/x-www-form-urlencoded" -X POST http://localhost:3000/data

-d "param1=value1&param2=value2": Send specified data in POST request.

-H "Content-Type: application/x-www-form-urlencoded”: Content type header

- -H "Content-Type: application/json"

-X POST: The request method to use.

- -X POST
- -X PUT

SUMMARY OF WEB ARCHITECTURE

- Principles: Internet Standard documented by a Request for Comments (RFC) IETF website: https://ietf.org/standards/rfcs/
- Identifiers: URI, URL, etc.
- Protocols: HTTP, SOAP, etc.
- Meta formats: XML, etc.
- Internationalization: IRI (Internationalization of identifiers)

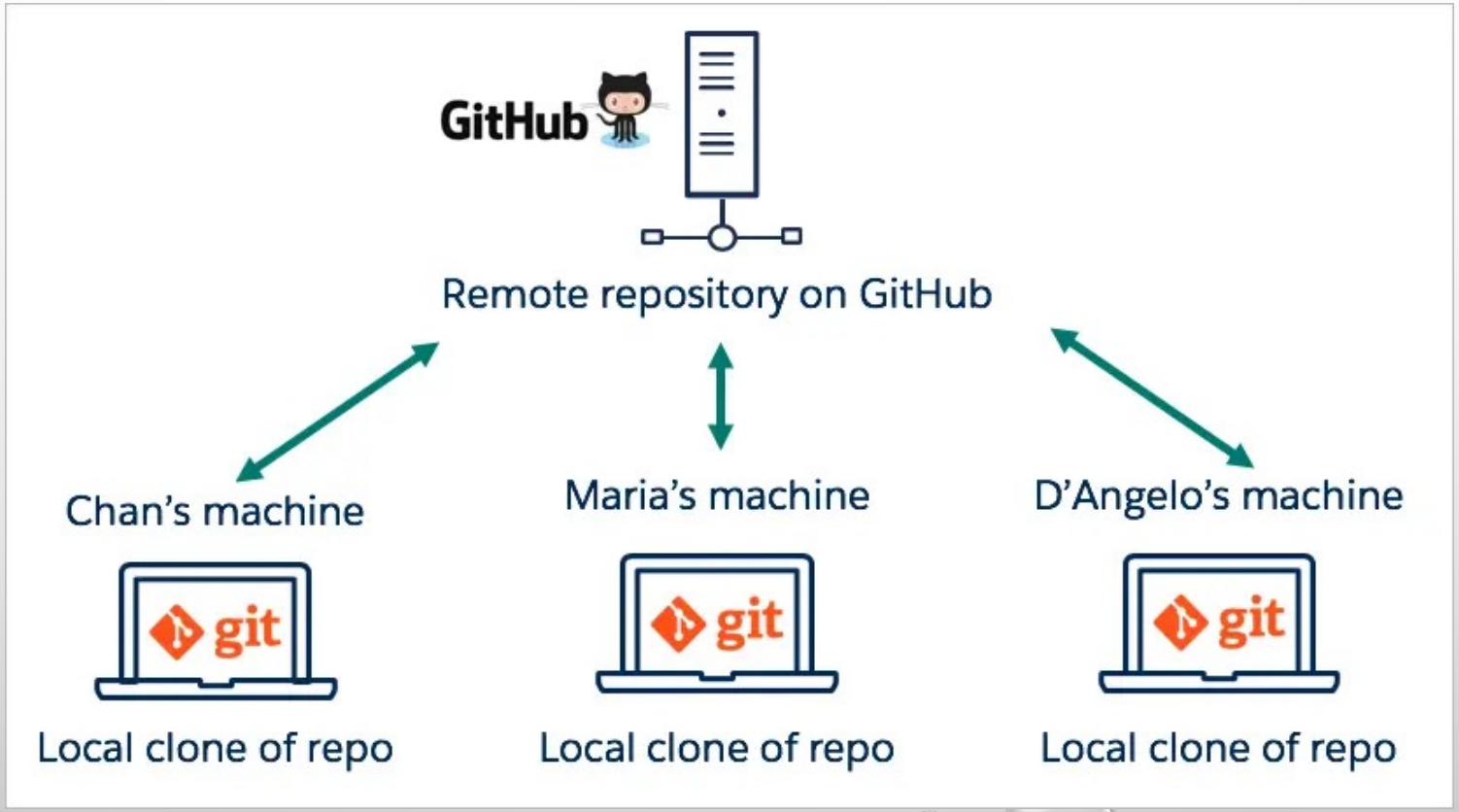
INTRODUCTION TO GIT

CS418/518

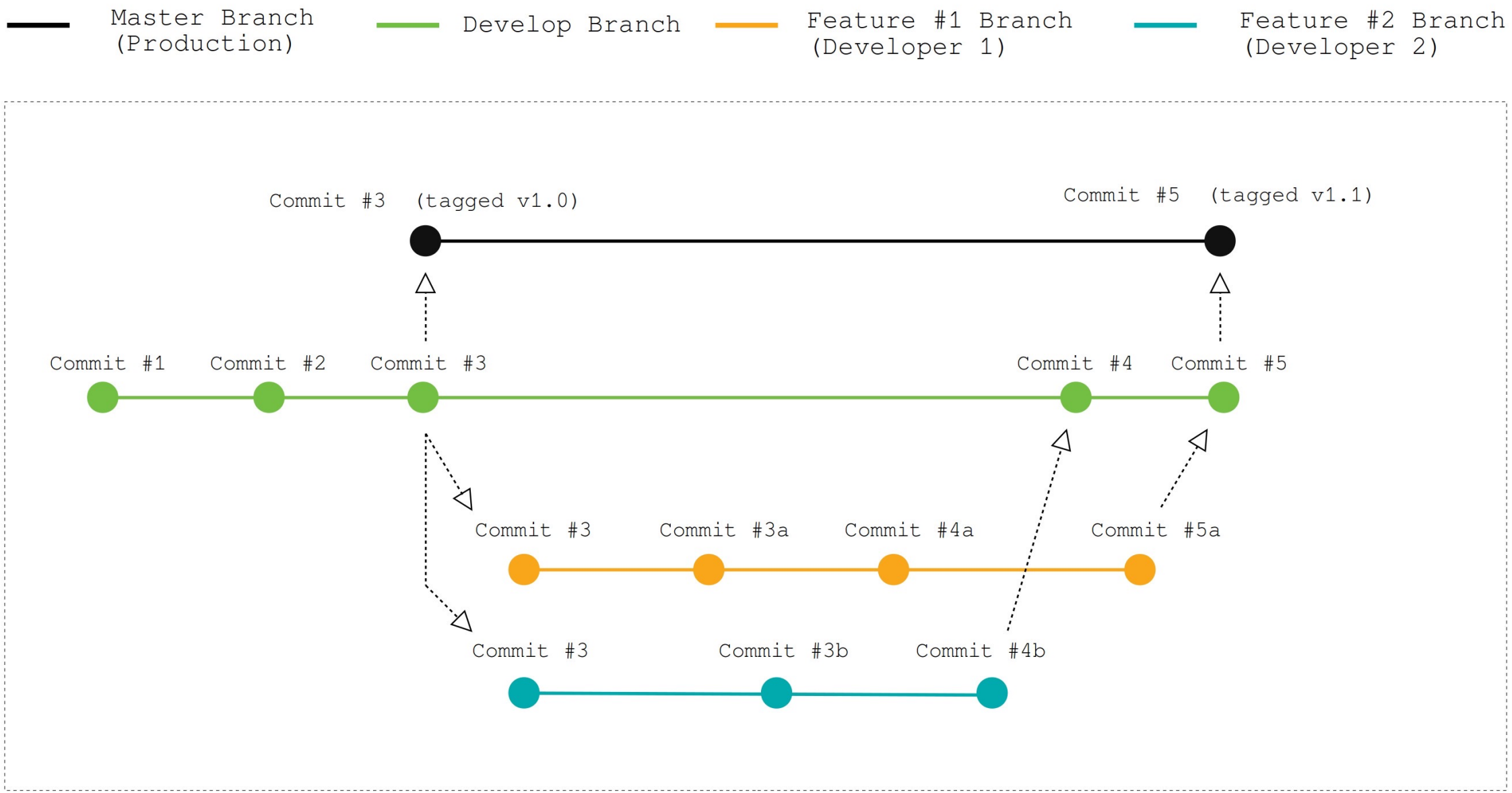
Jian Wu

OUR GOAL

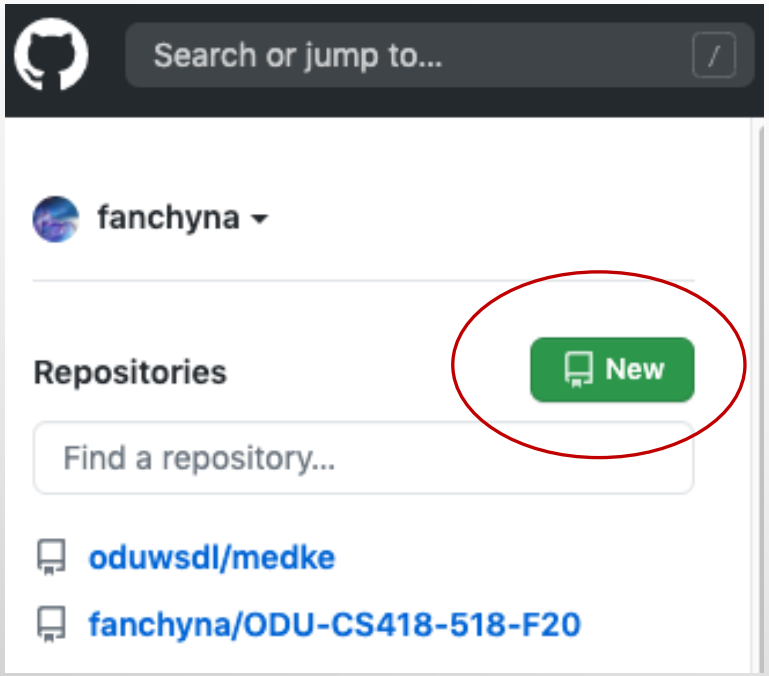
- Create a code repo on github and sync it with the repo on your local machine



A Simple, Effective Git Workflow



First create a new repository on GitHub



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * **Repository name** *

fanchyna /

Great repository names are short and memorable. Need inspiration? How about improved-barnacle?

Description (optional)

Public
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

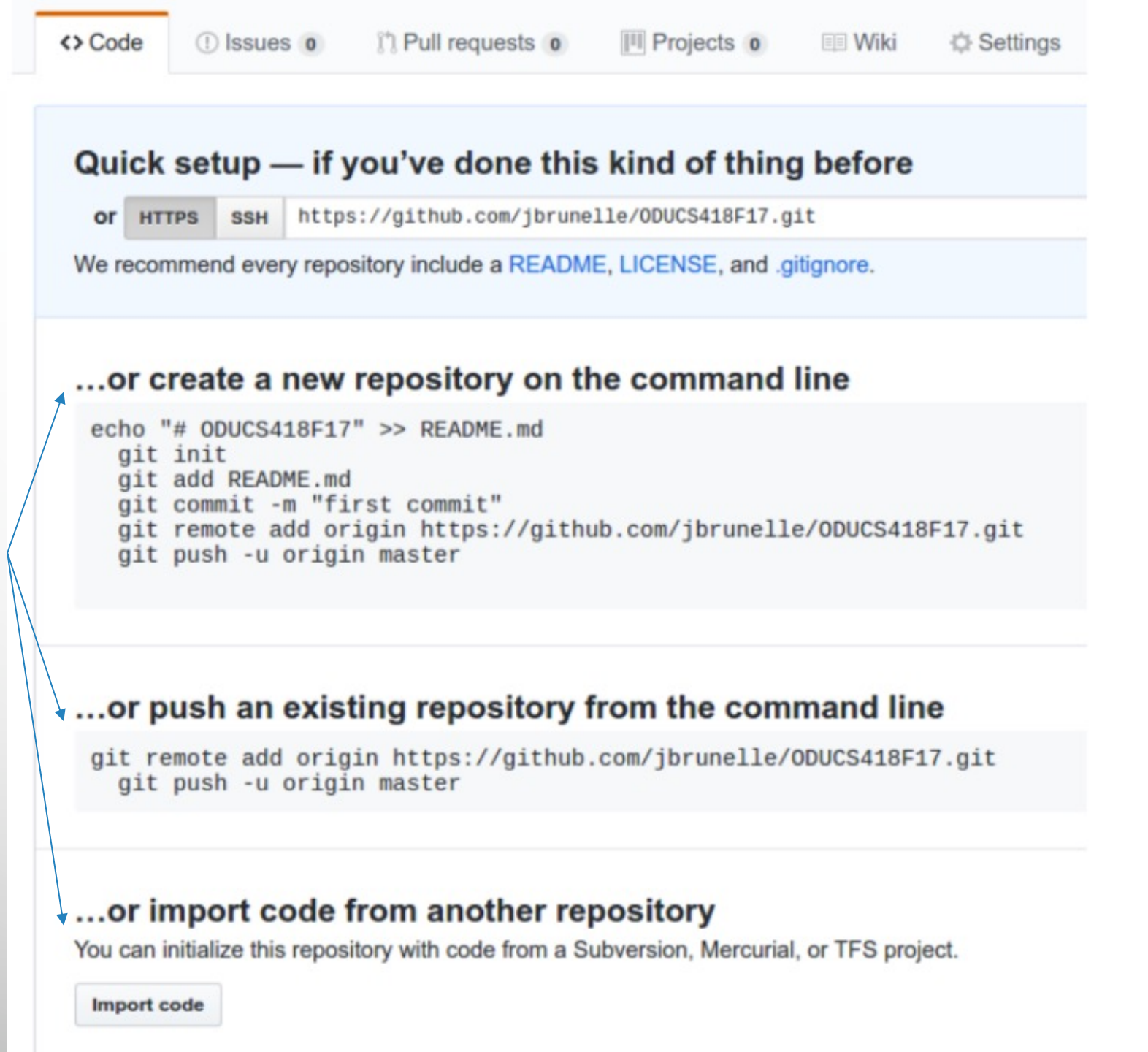
Initialize this repository with:
Skip this step if you're importing an existing repository.

Add a README file
This is where you can write a long description for your project. Learn more.

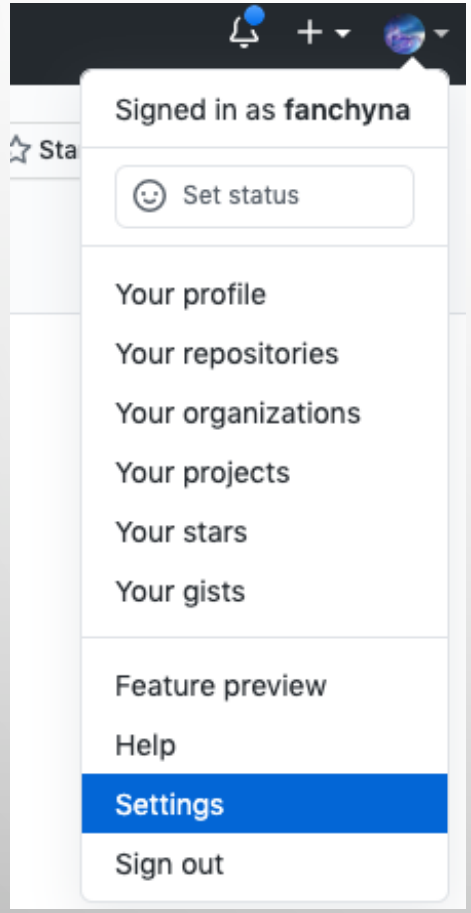
Add .gitignore
Choose which files not to track from a list of templates. Learn more.

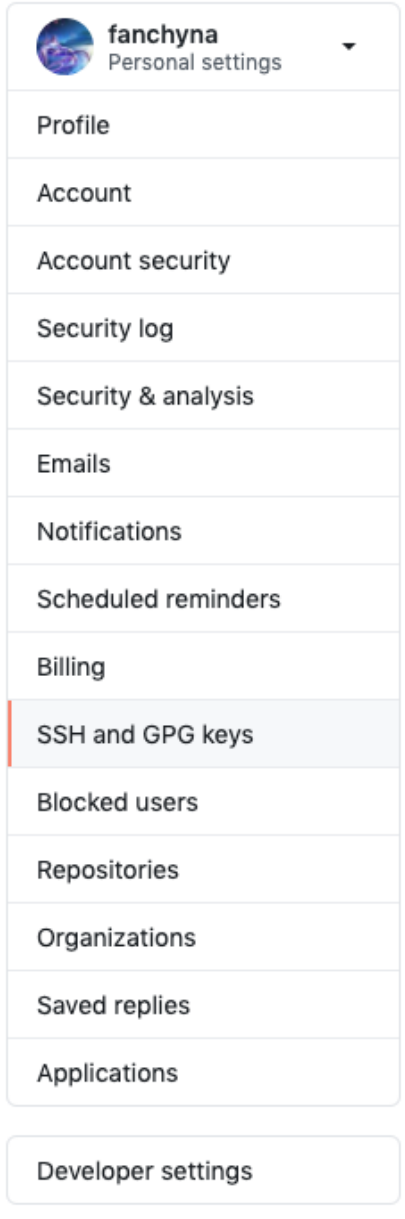
Choose a license
A license tells others what they can and can't do with your code. Learn more.

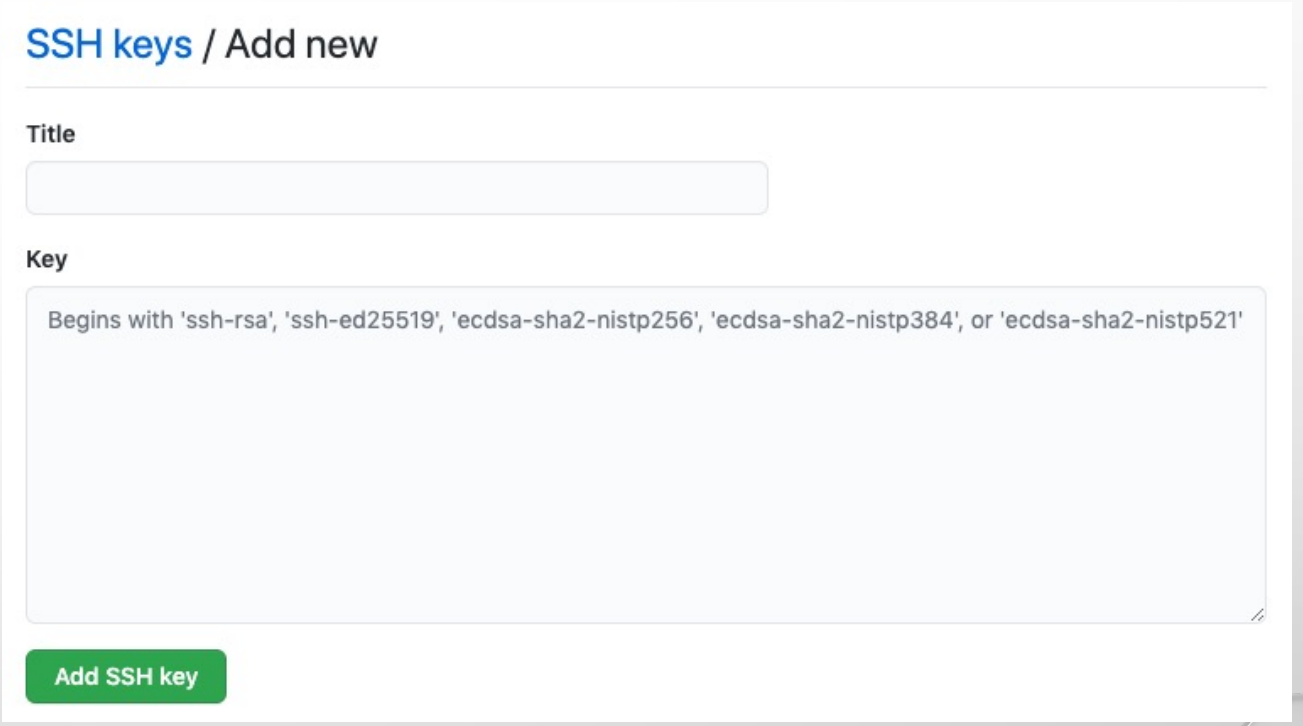
3 ways to create repositories



RSA_PUB KEY







[How to generate an RSA key in Windows 10]
[How to generate SSH keys on Mac OSX?]

WHY USING COMMAND LINES?

1. To show off
2. Build a professional understanding of what is going on behind drag-and-drop
3. Easy to fix problems

BRANCHES

- Purpose: work on a new feature with impacting the stable code
- Command:
  - `$ git checkout –b [branchid]`
- Merge:
  - `$ git checkout master` Switched to branch ‘master’
  - `$ git merge [branchid]`
  - `$ git status`

GIT BRANCHING GUIDES AND TUTORIALS

- https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell
- http://rogerdudler.github.io/git-guide/

ASSIGNMENT

- See the assignment document
- Due Tuesday September 6

BACKUP SLIDES BEYOND THIS POINT