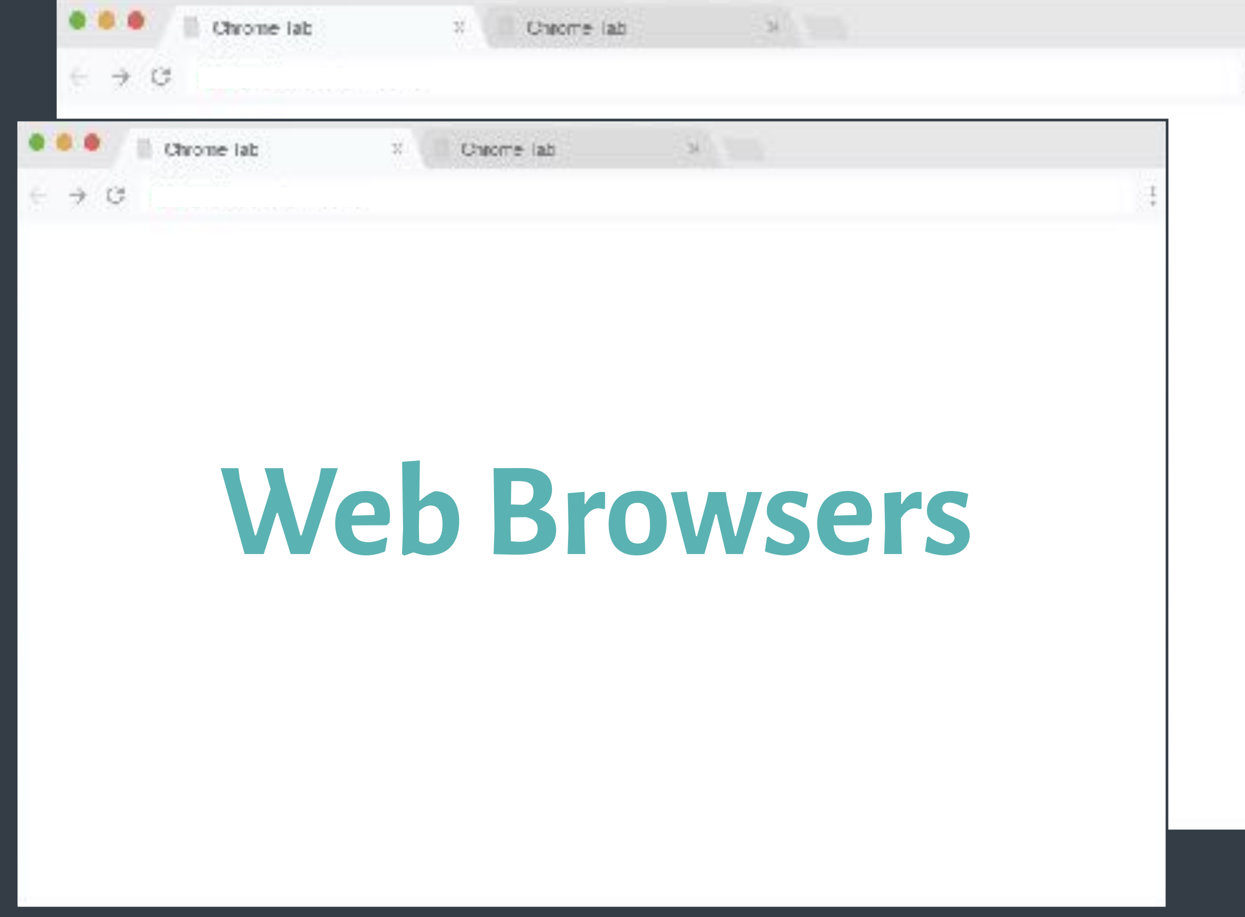
6.1040: Software Design

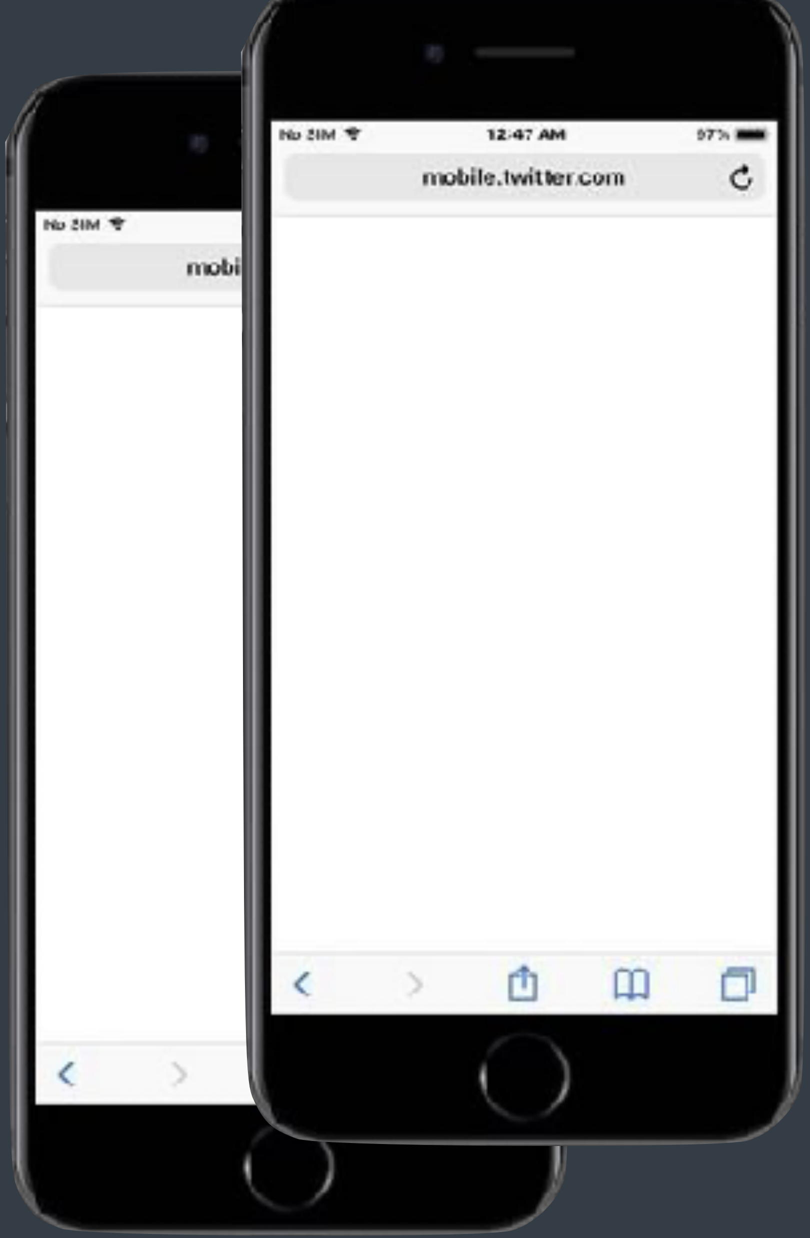
Service (API) Design

Arvind Satyanarayan & Daniel Jackson

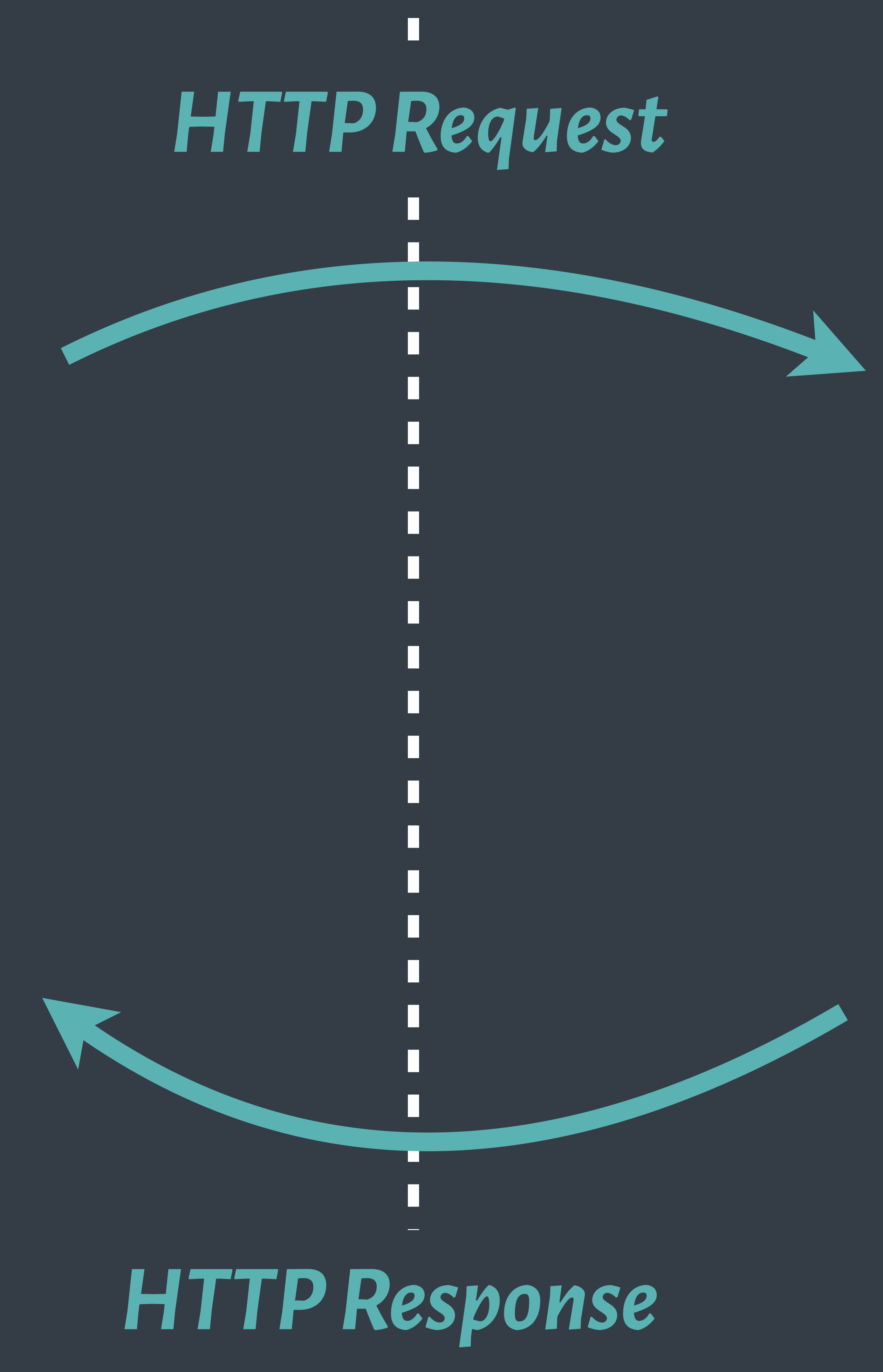
Client Side

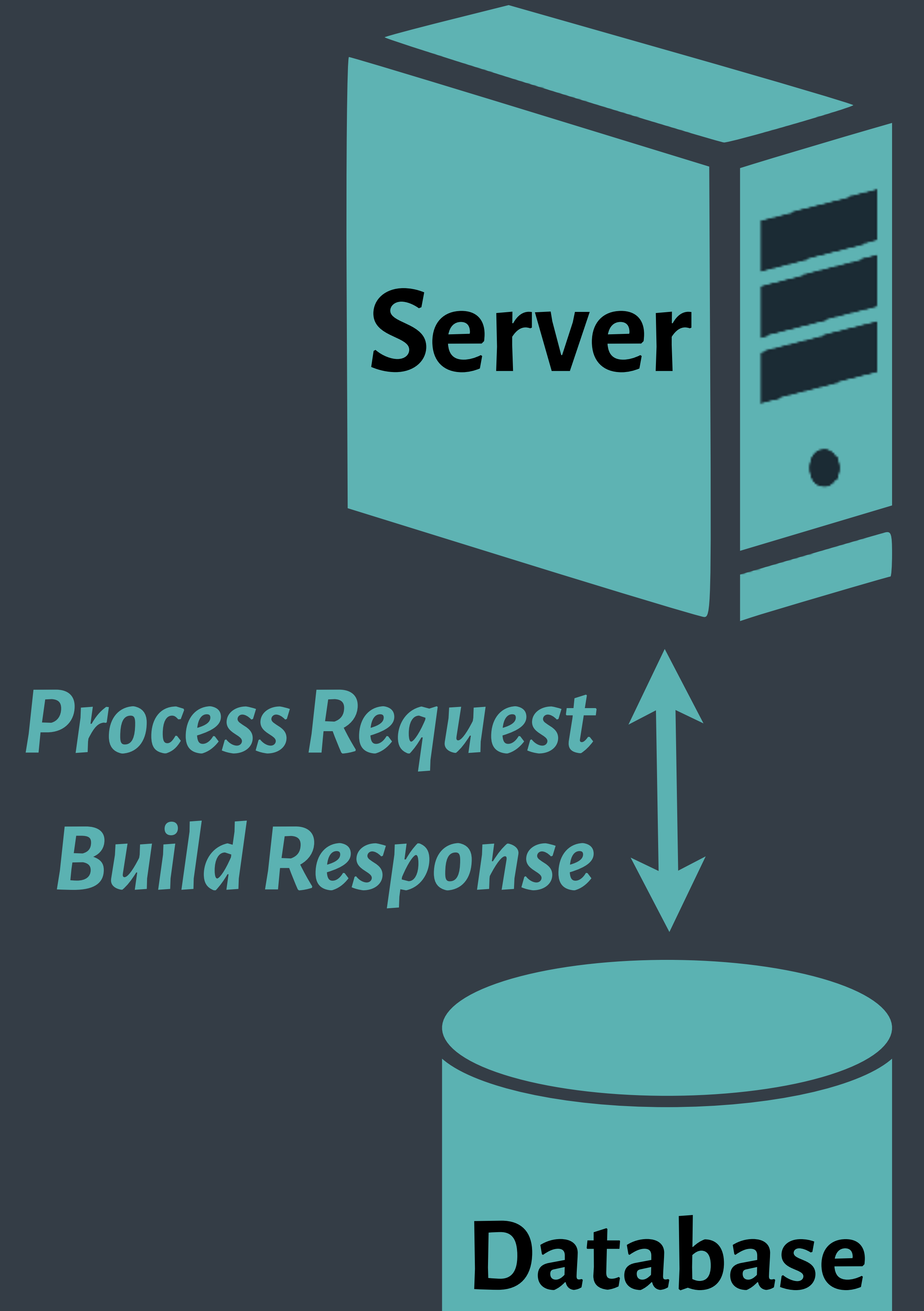
Server Side





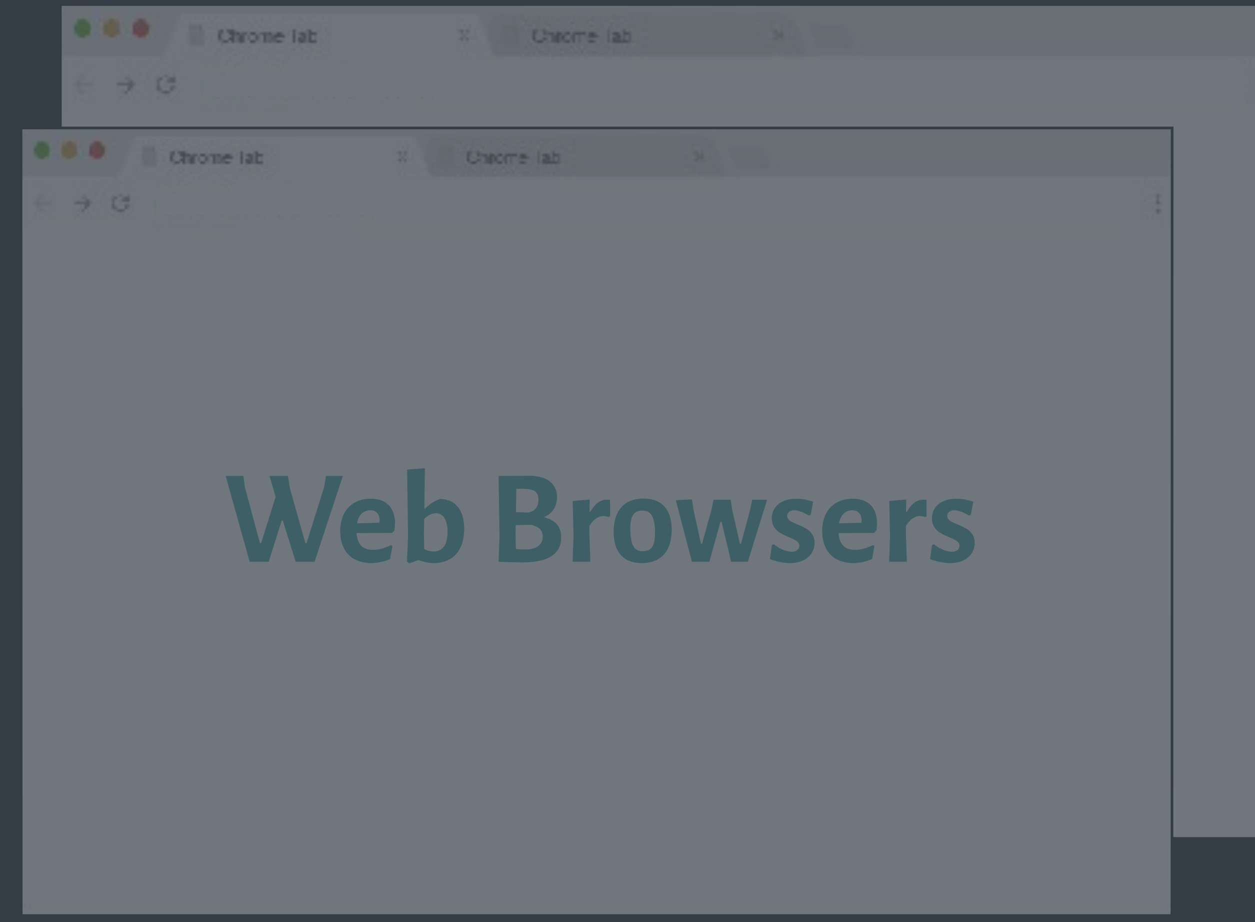
Mobile Apps

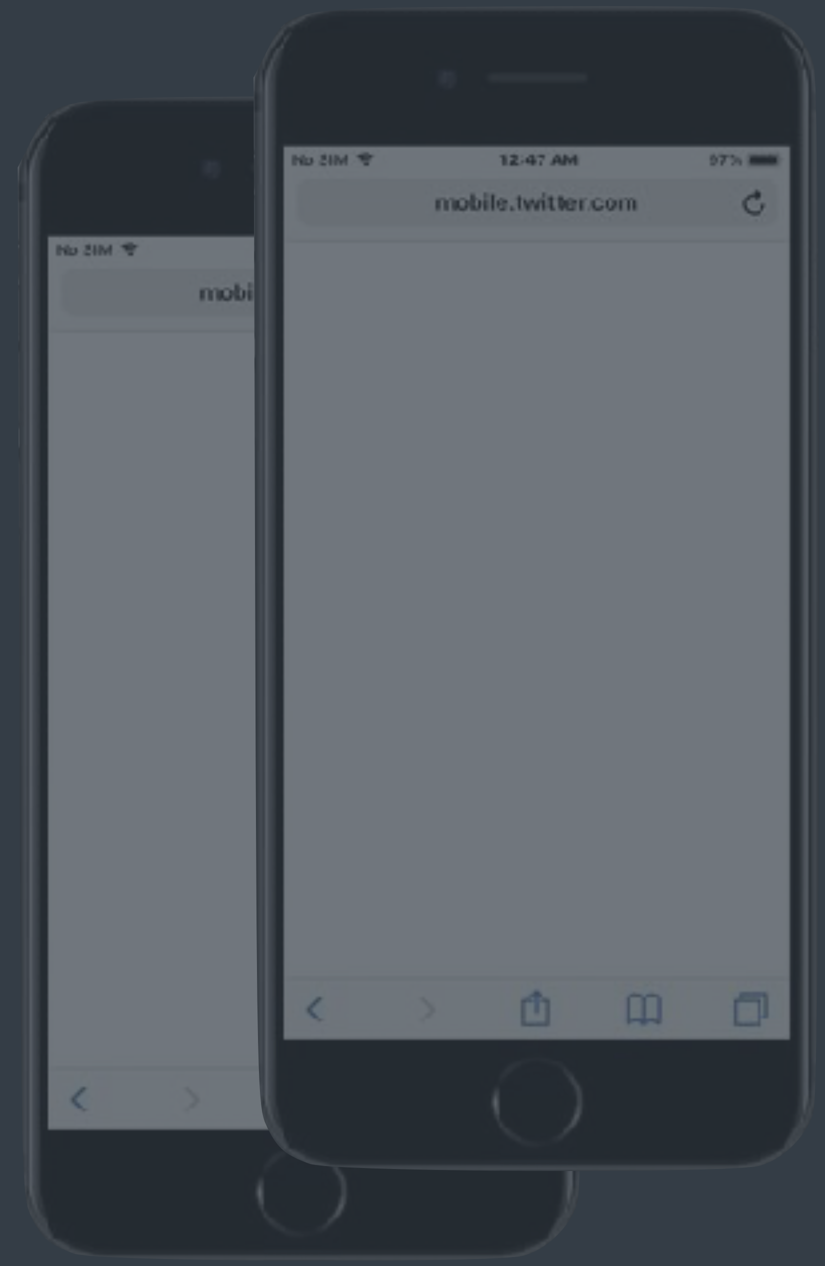


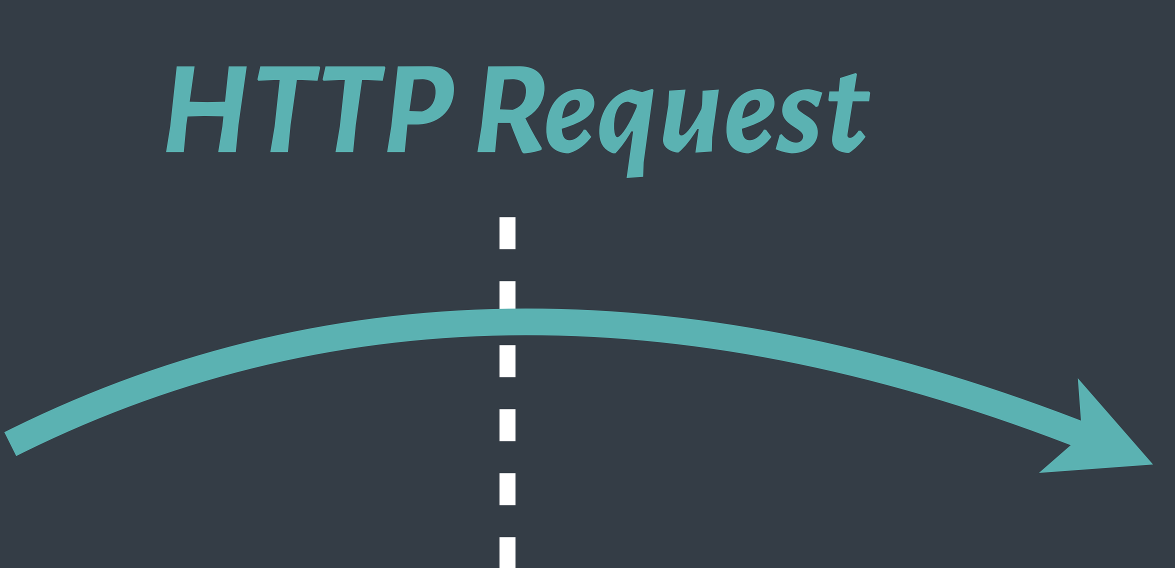


Client Side

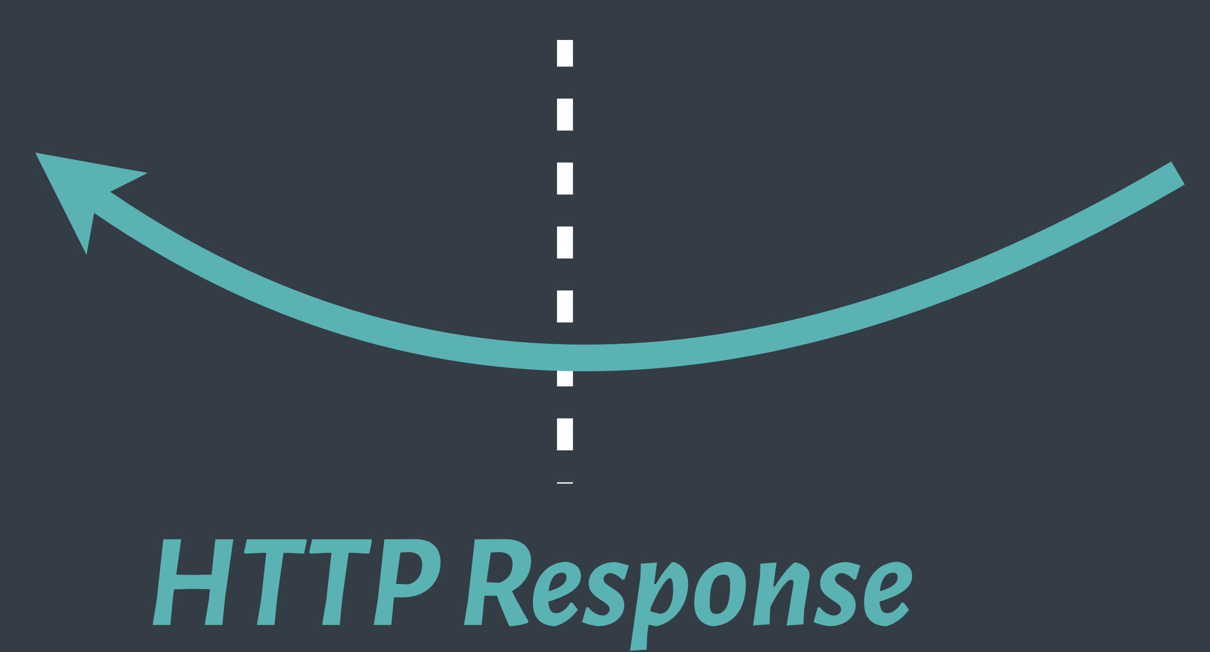
Server Side

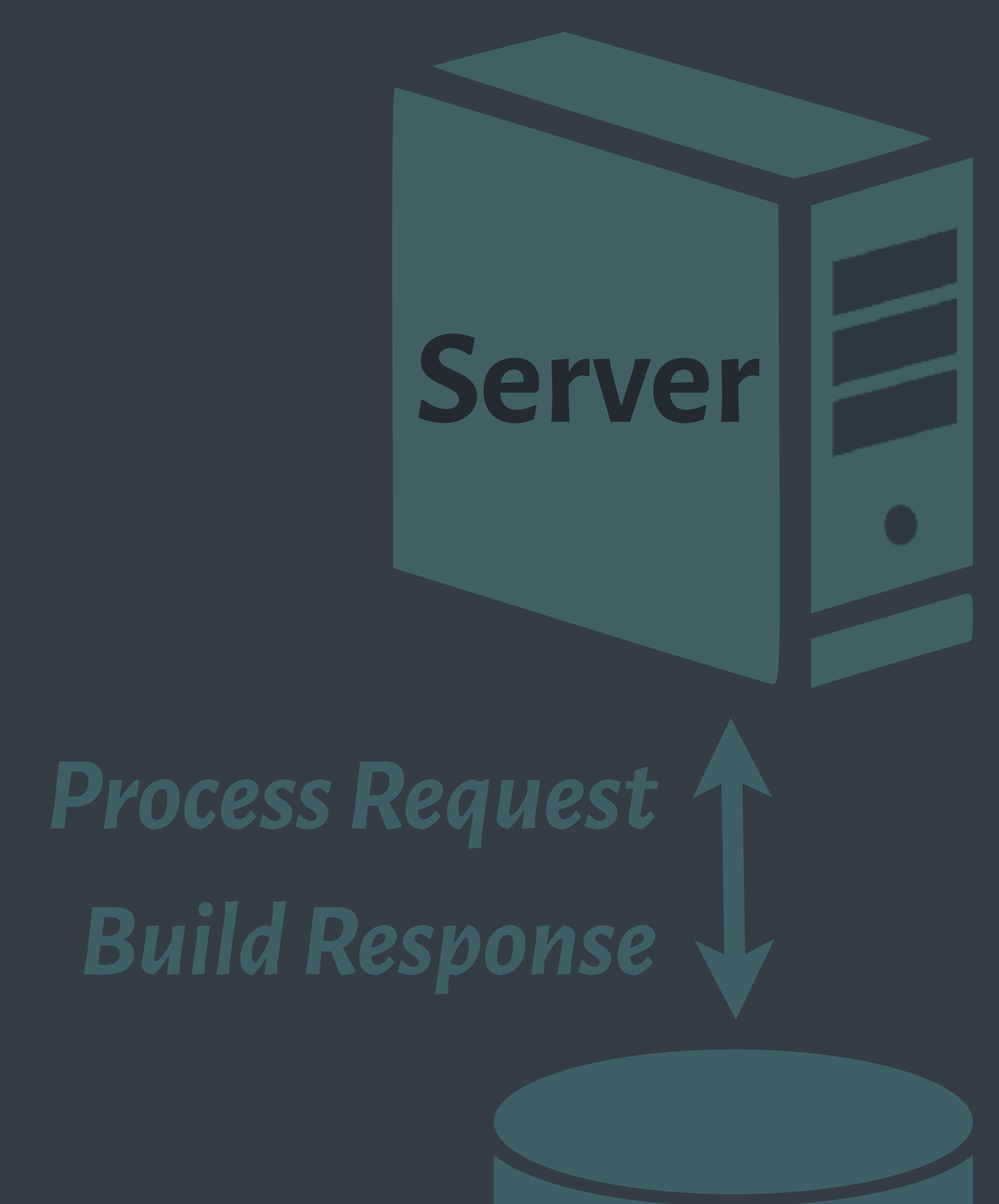


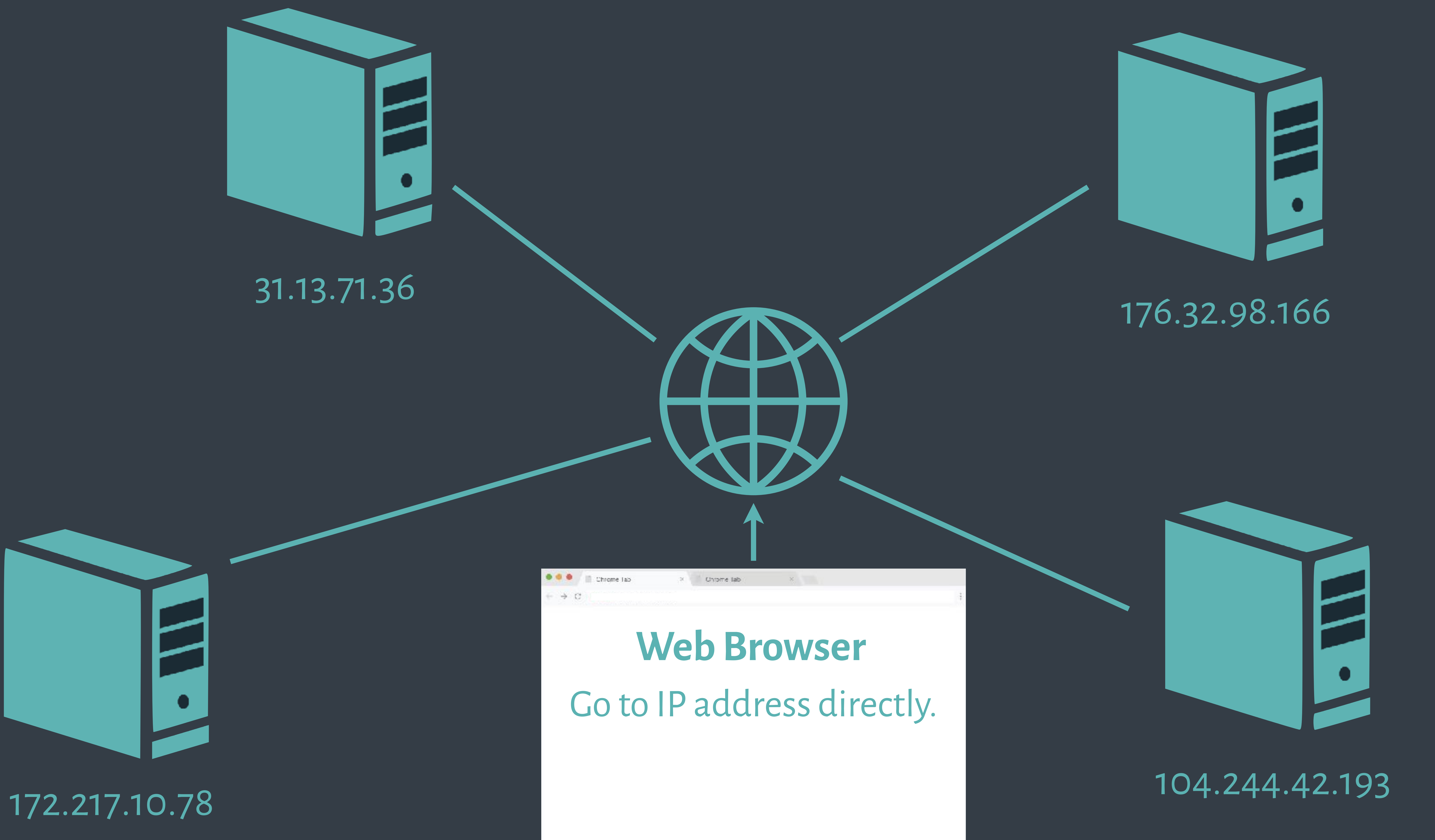




URLs are an **interface** that **require design**





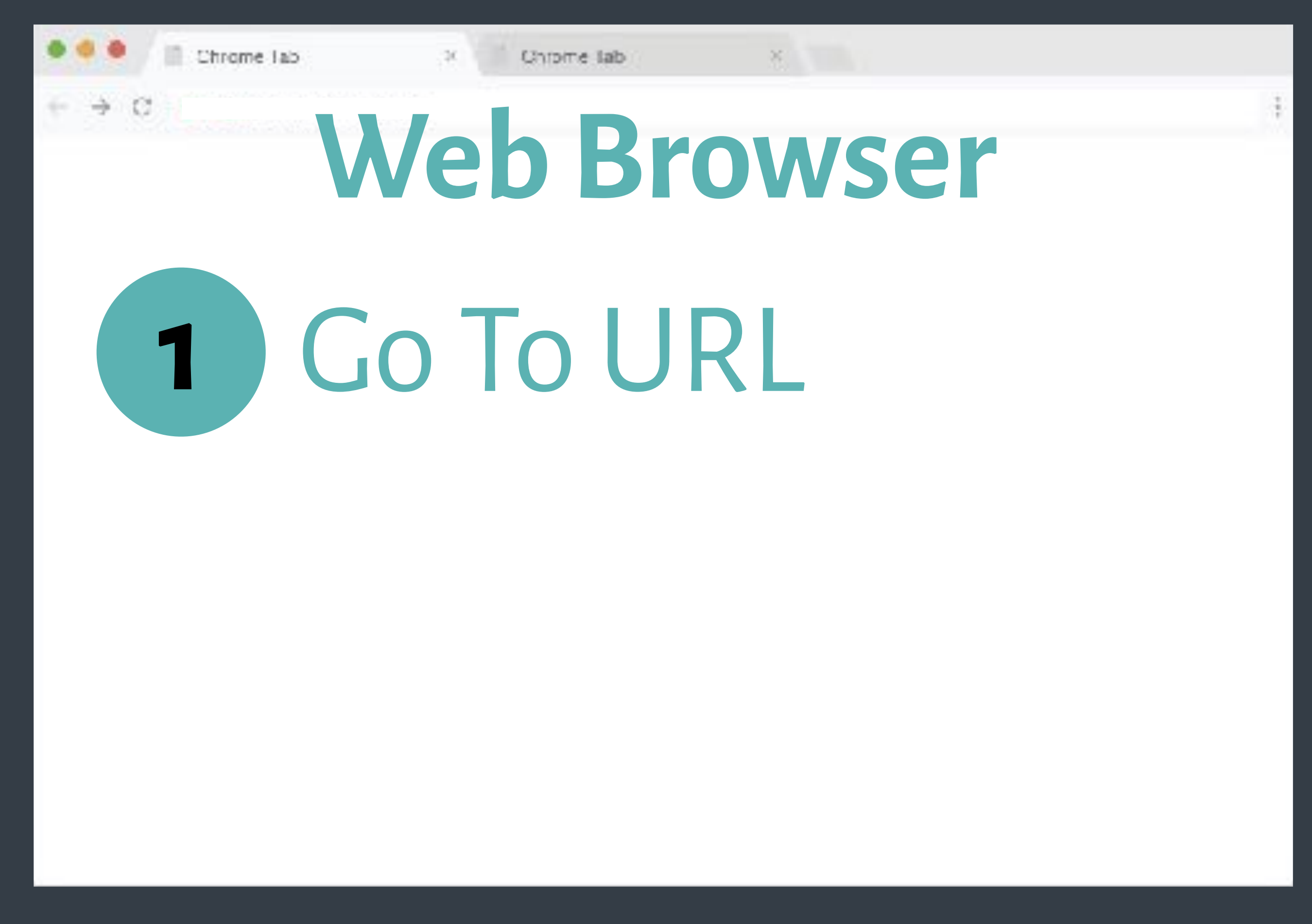

31.13.71.36
176.32.98.166
172.217.10.78
104.244.42.193
Web Browser
Go to IP address directly.

ANATOMY OF A URL

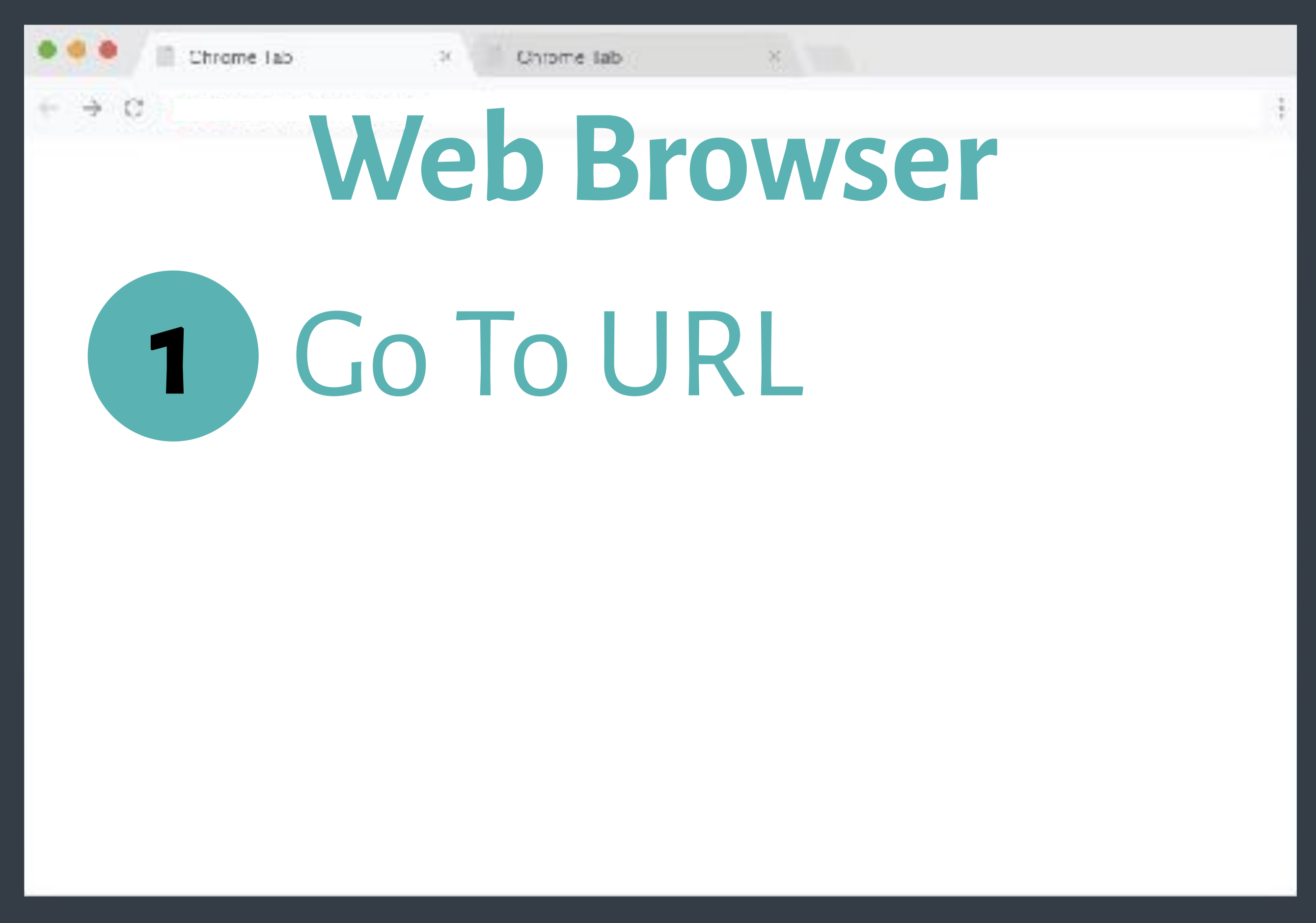
Protocol　Host　Path

http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya#topic-footer-buttons

Query Parameters　Fragment



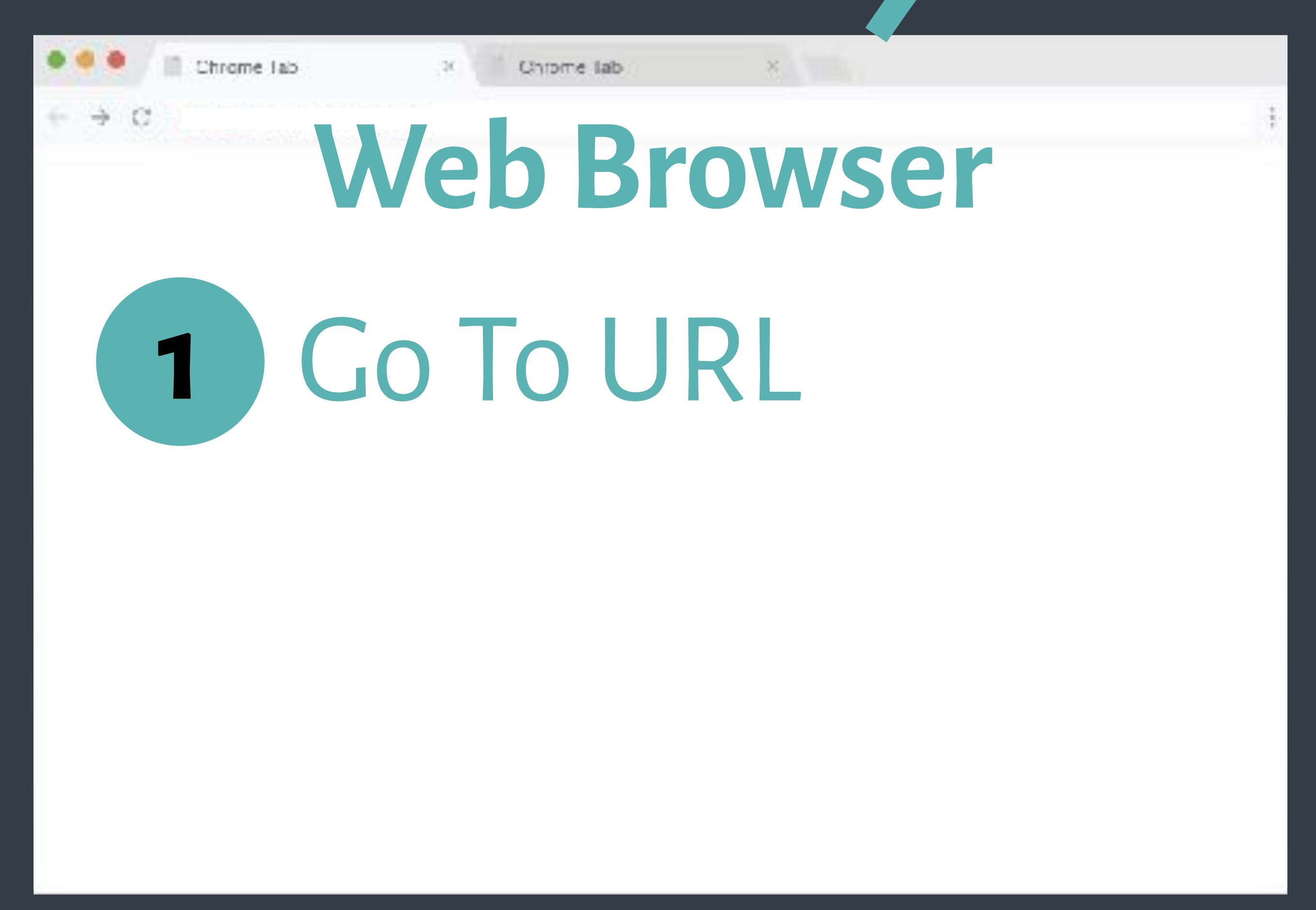
http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya



Host

http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya

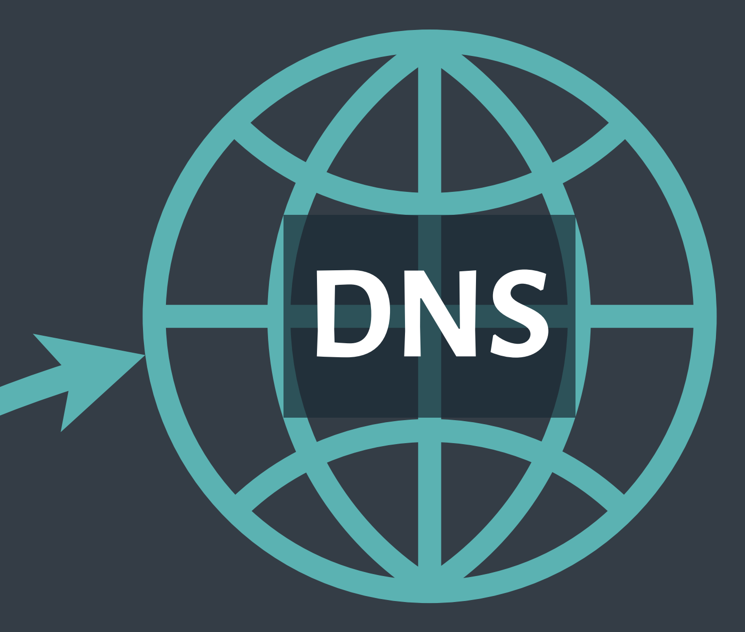
2 Lookup
Domain Name

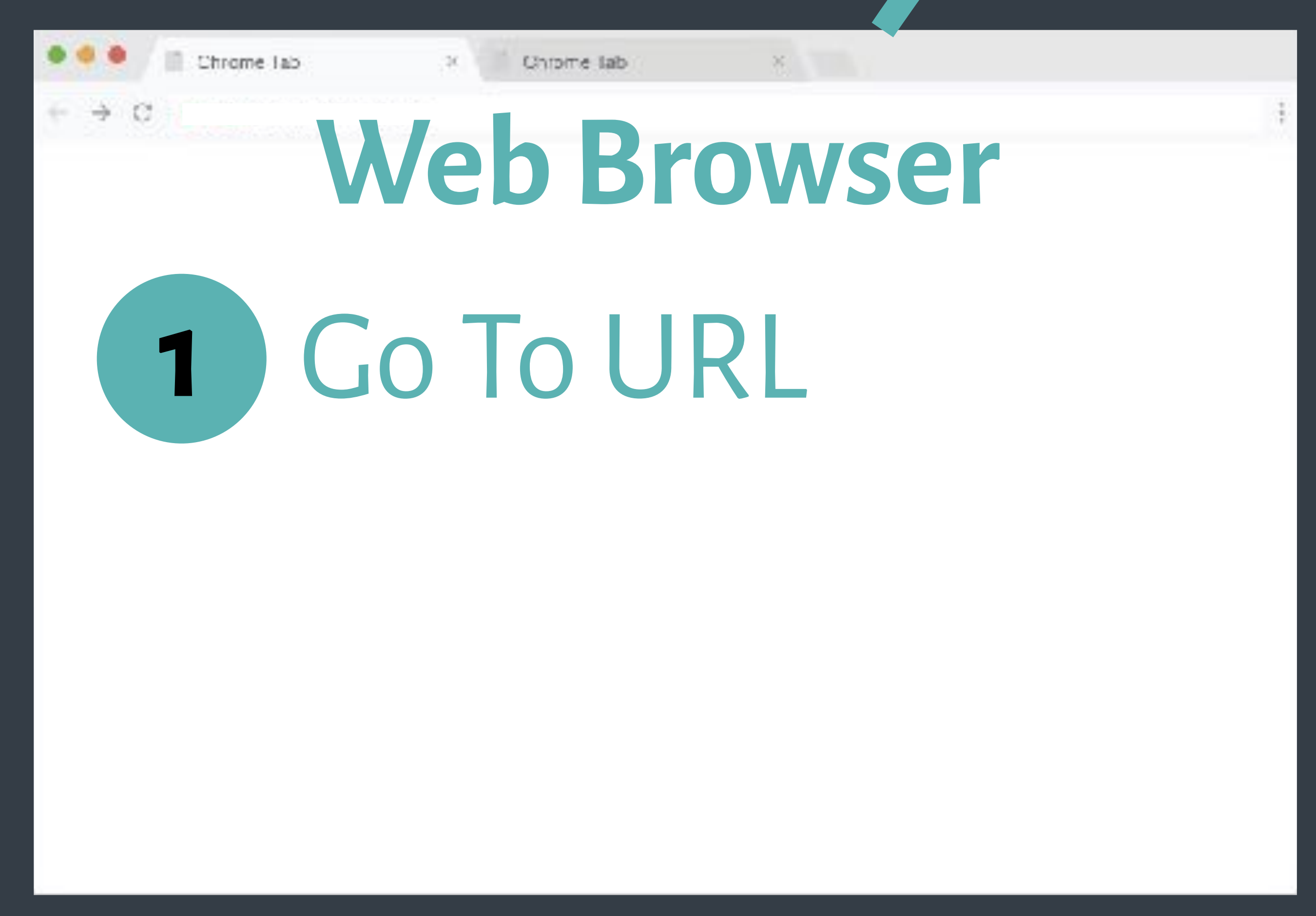


Host

http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya

2 Lookup Domain Name

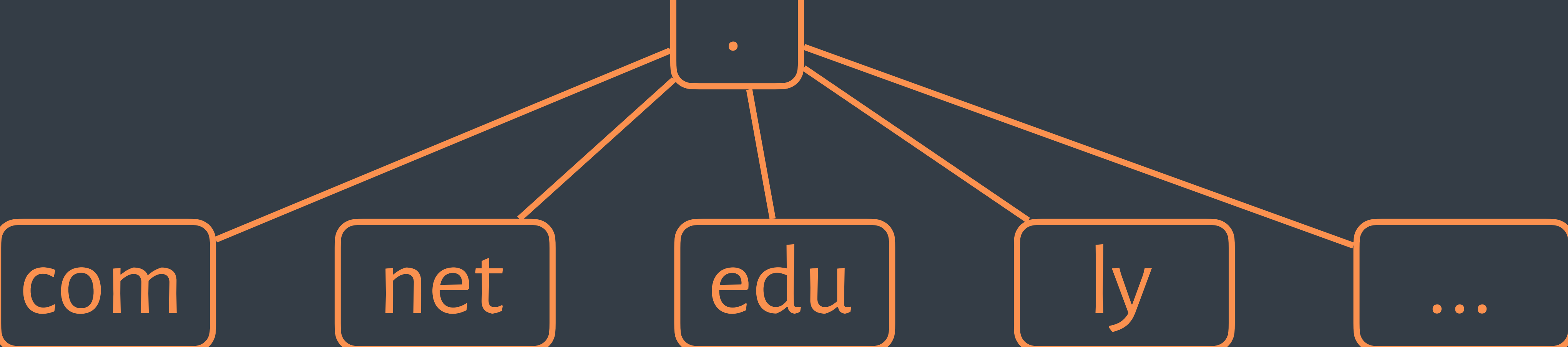


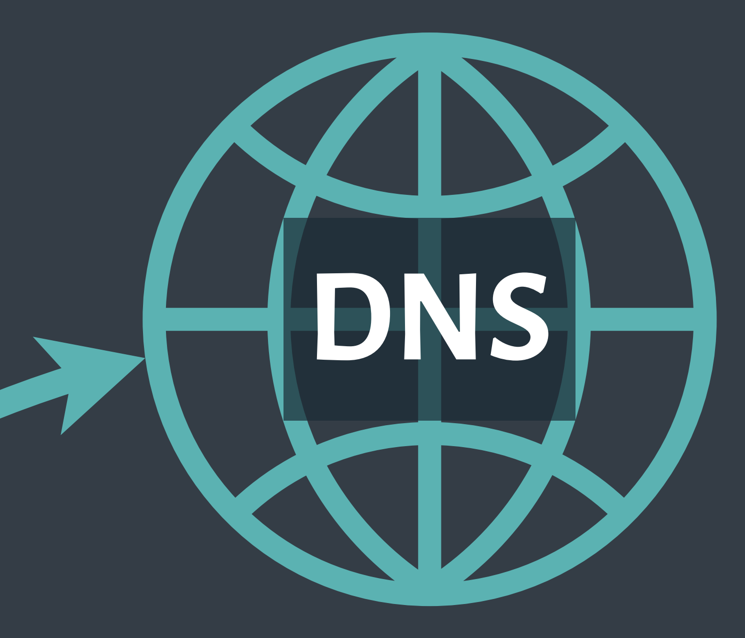


Host

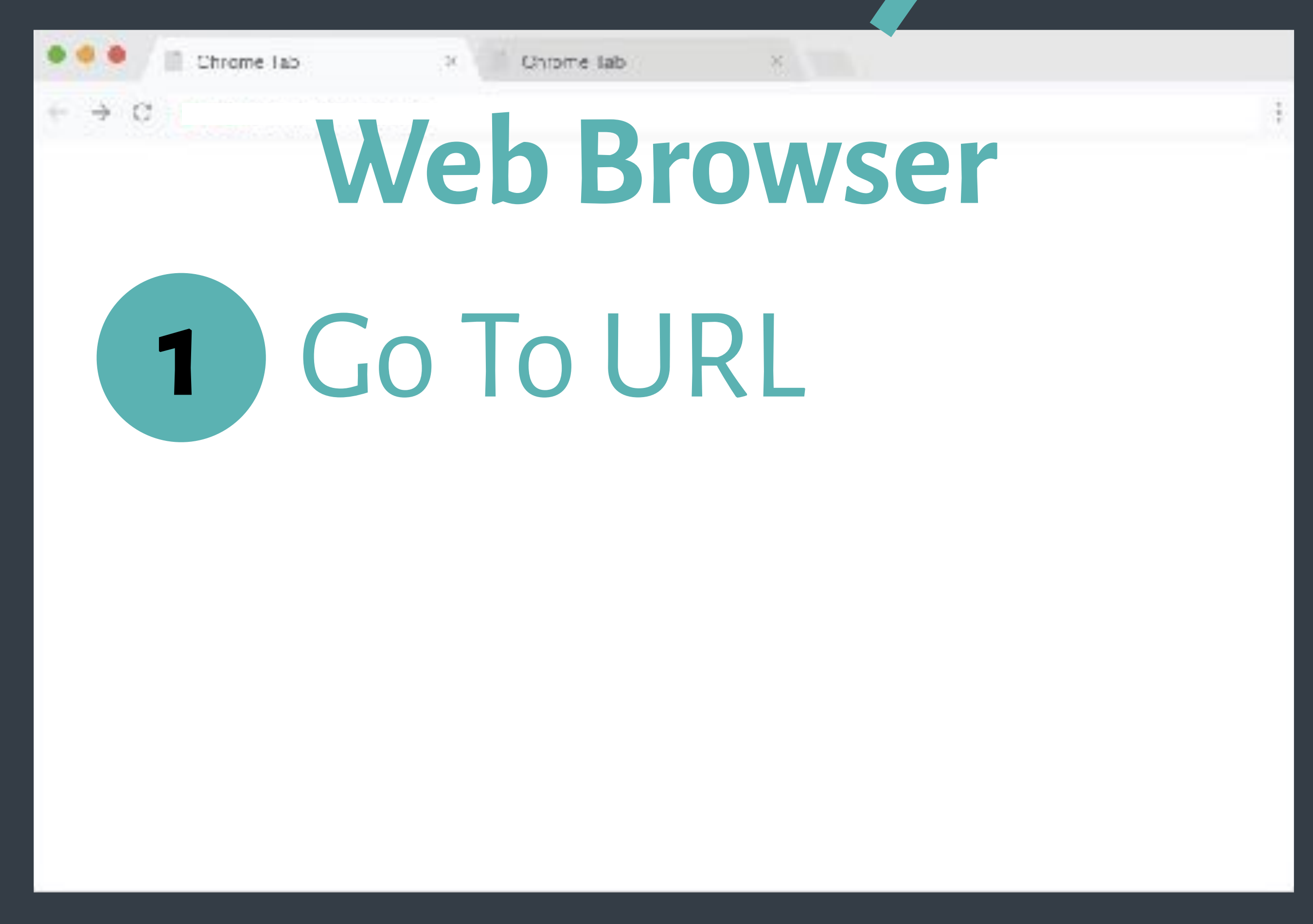
http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya

top-level domain (TLD)





2 Lookup Domain Name

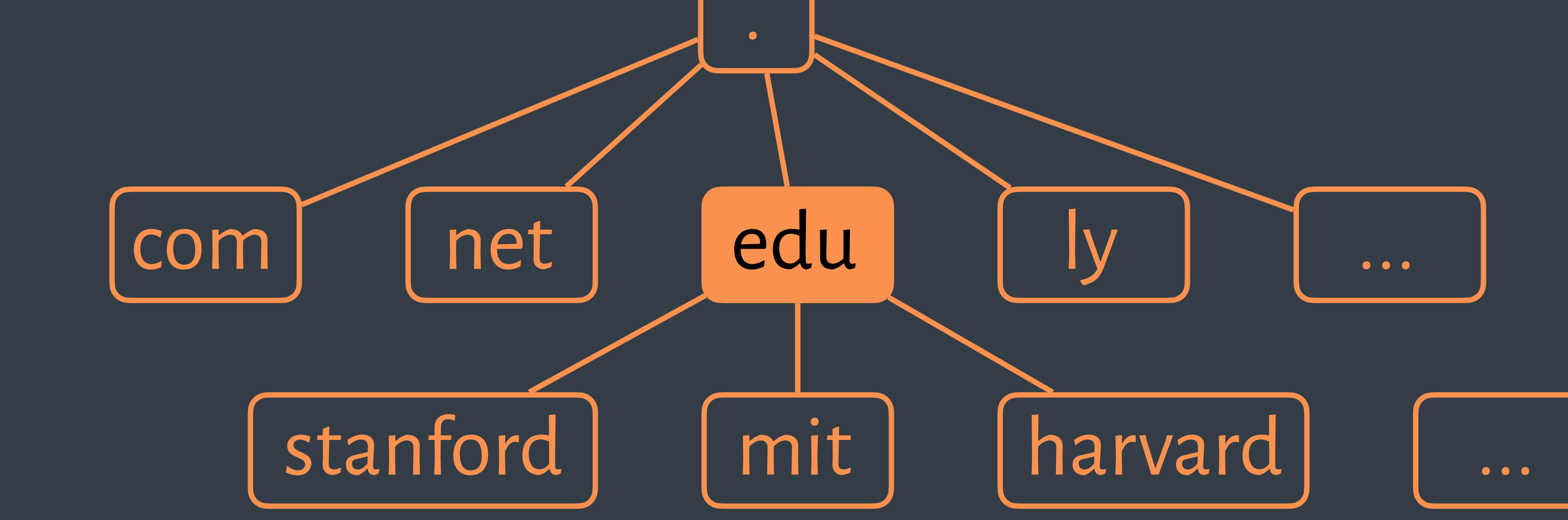


Host

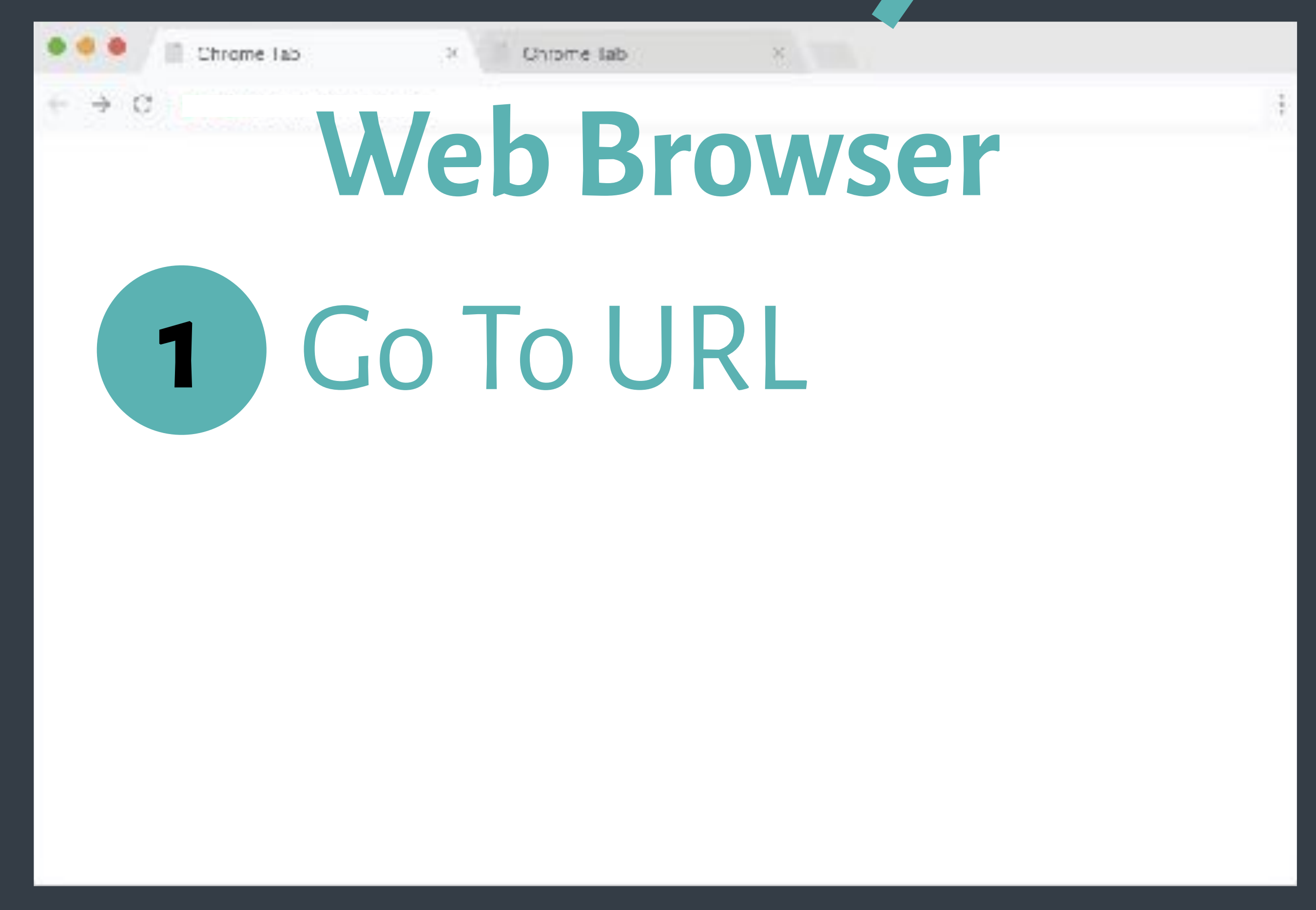
http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya

top-level domain (TLD)

domain



2 Lookup Domain Name



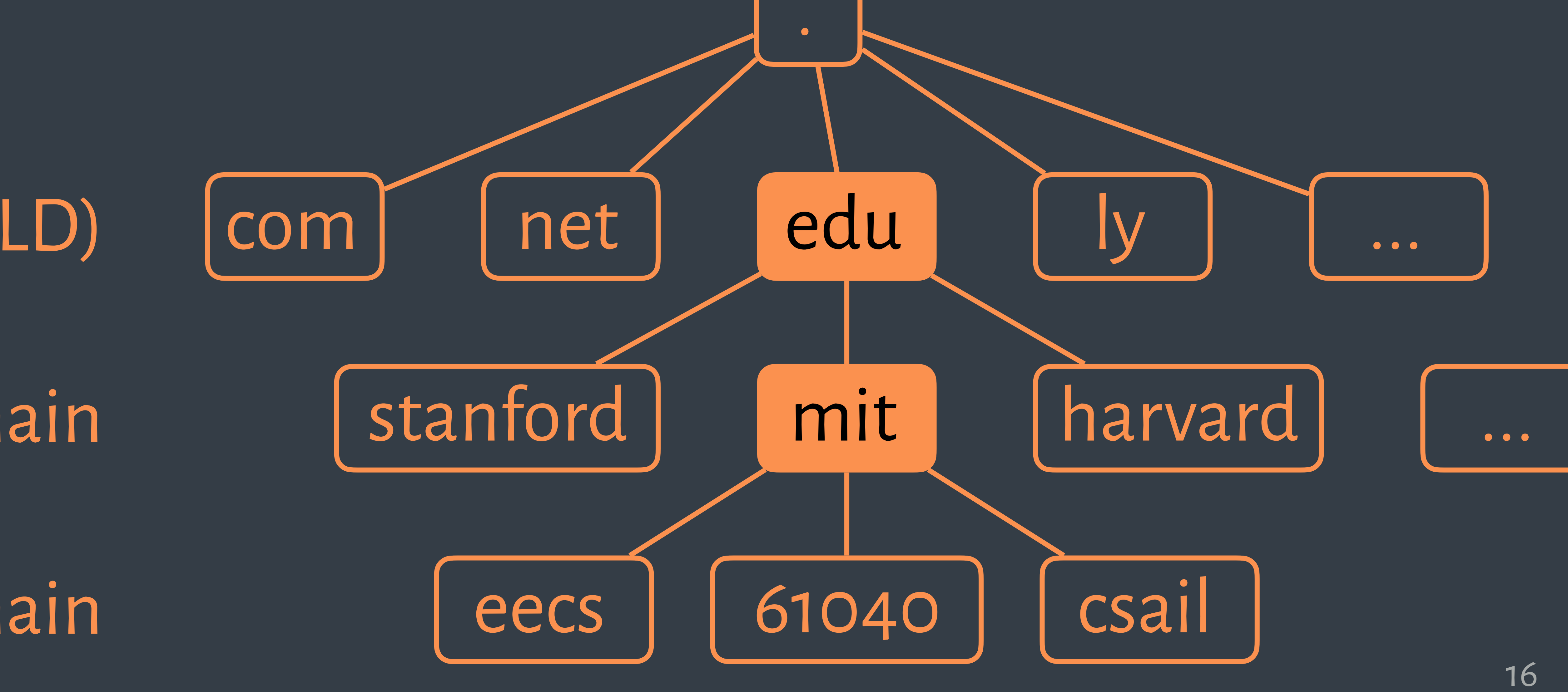
Host

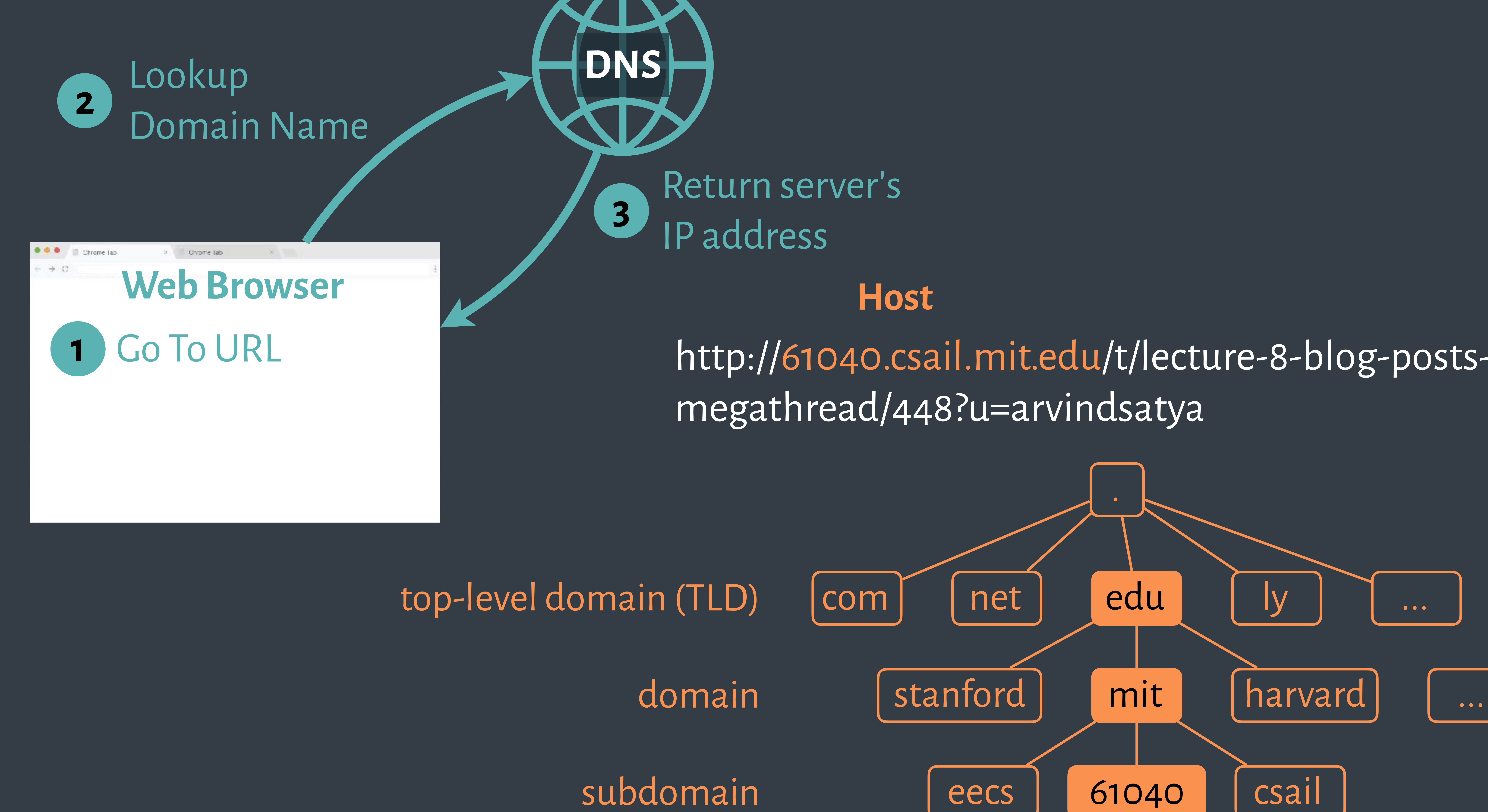
http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya

top-level domain (TLD)

domain

subdomain

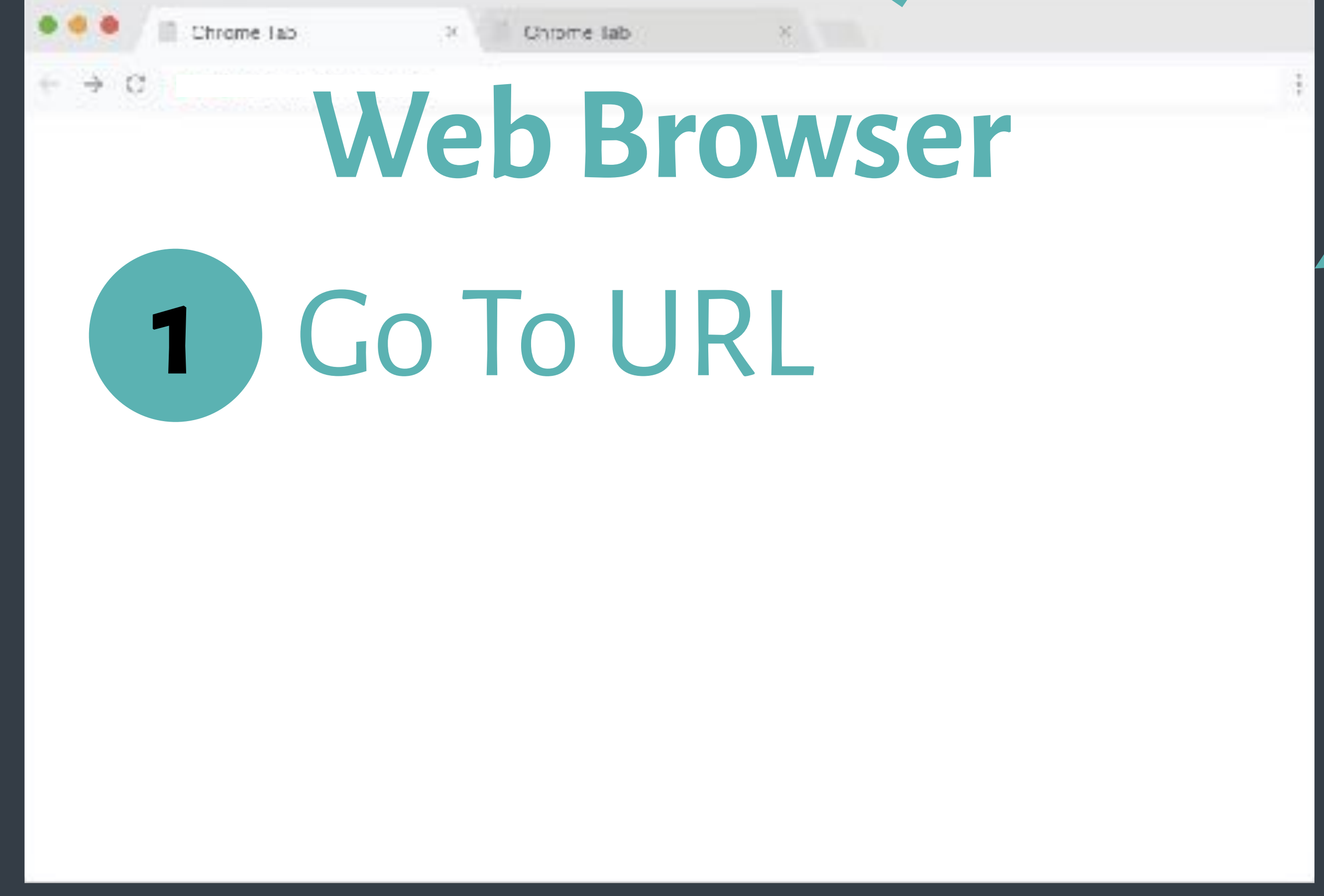



DNS
2
Lookup
Domain Name
3
Return server's
IP address
Web Browser
1
Go To URL
Host
http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya
.
top-level domain (TLD)
com
net
edu
ly
...
domain
stanford
mit
harvard
...
subdomain
eecs
61040
csail

DNS

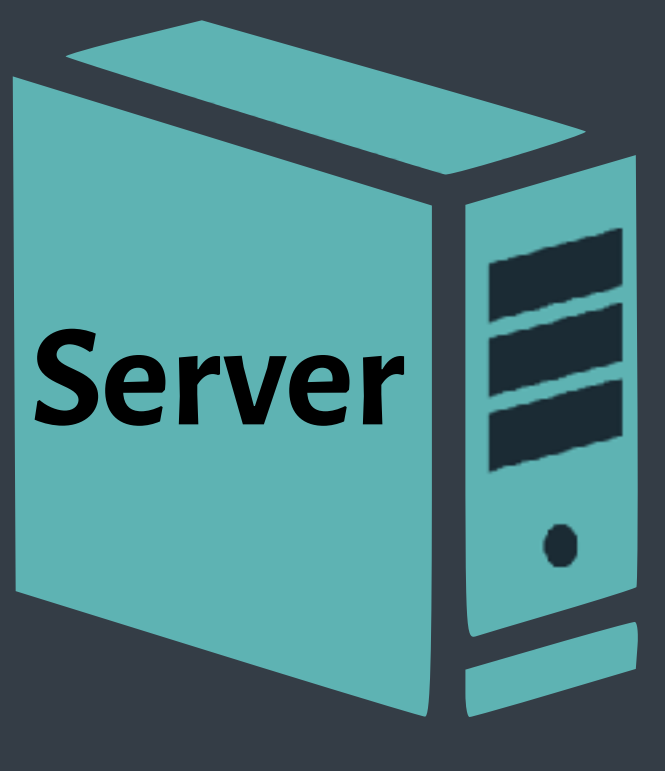
2 Lookup Domain Name

3 Return server's IP address

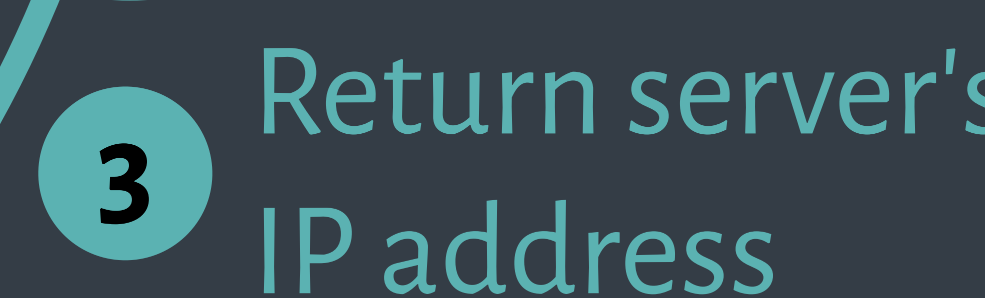


Host

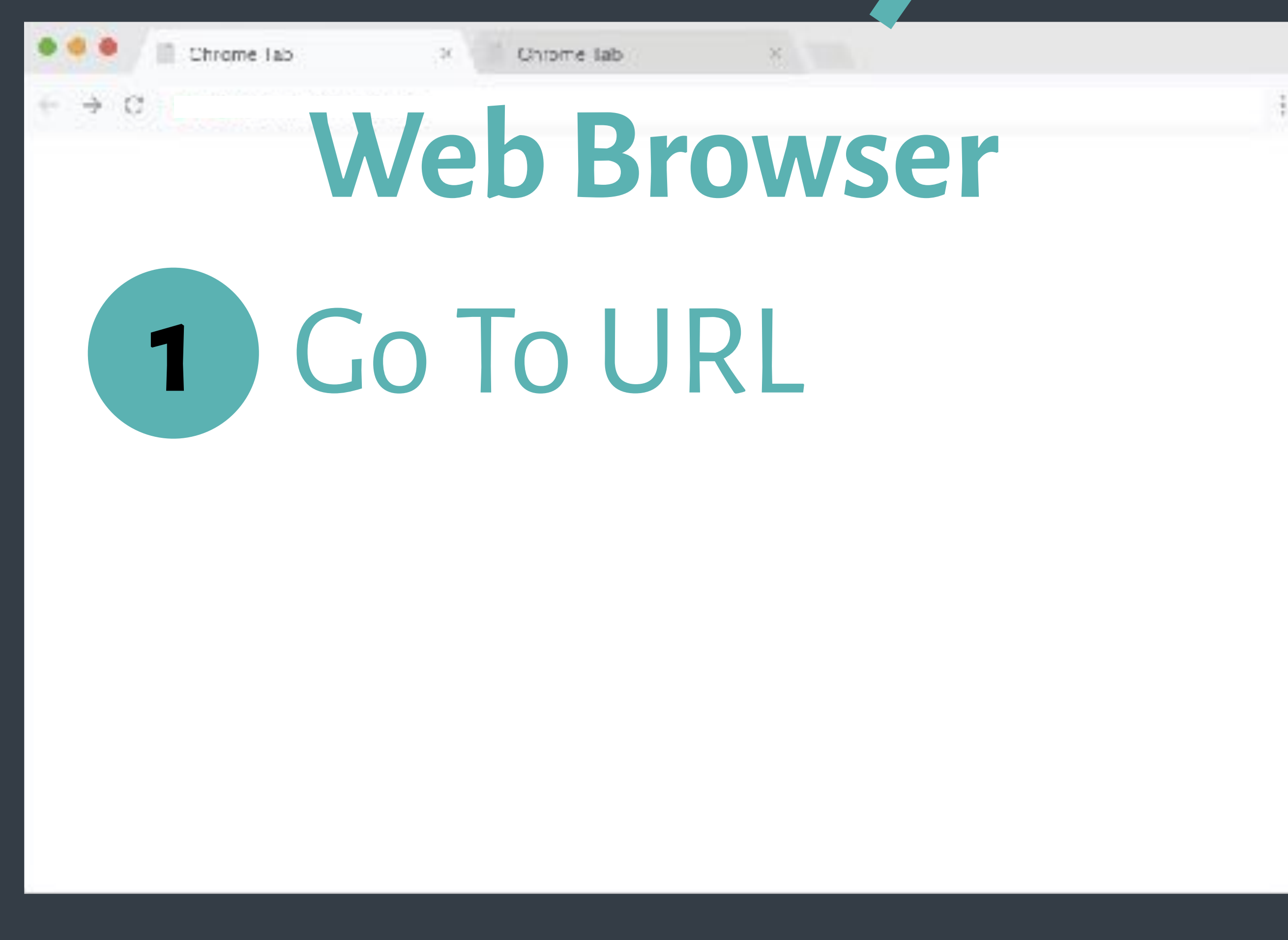
http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya



128.52.130.153



2 Lookup Domain Name

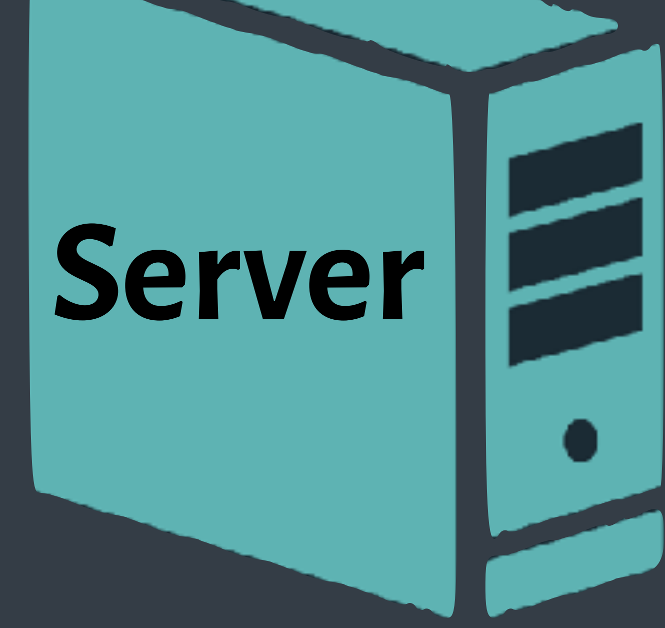


3 Return server's IP address

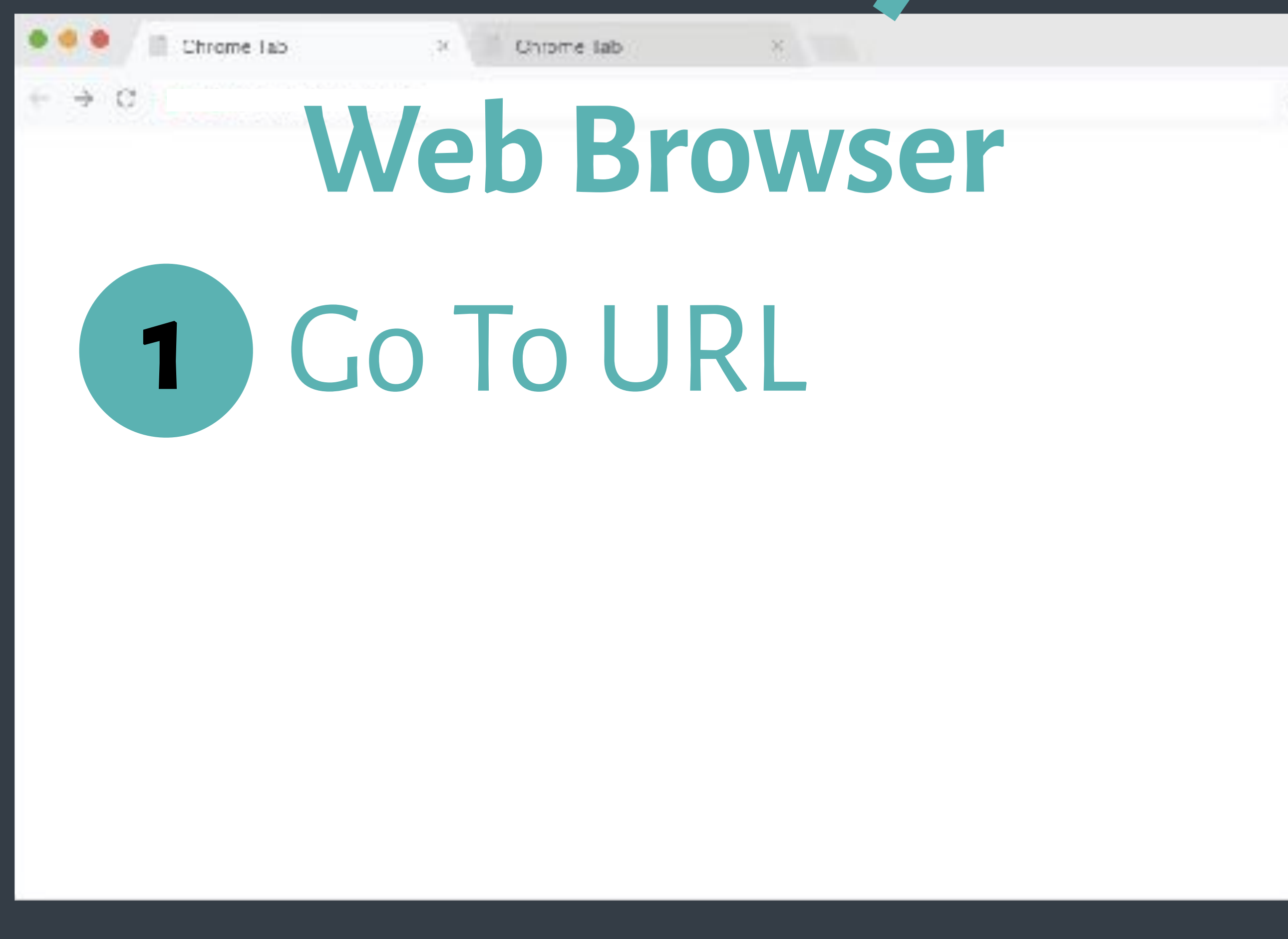
Host

Protocol http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya

4 HTTP Request



128.52.130.153



2 Lookup Domain Name

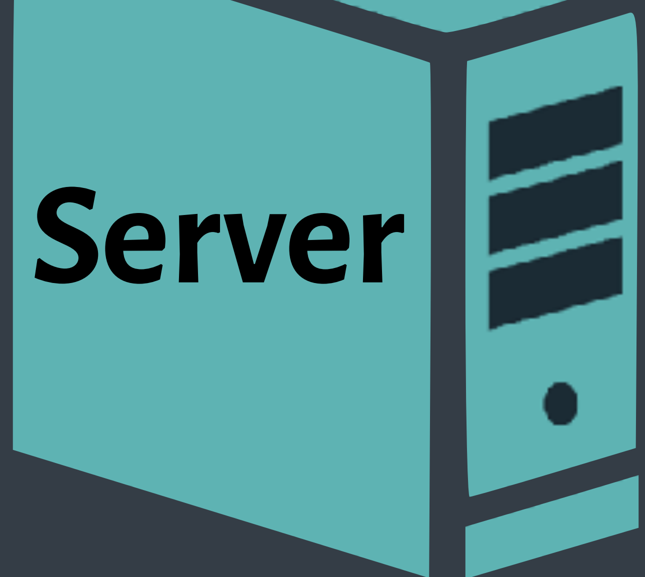
DNS

3 Return server's IP address

Host

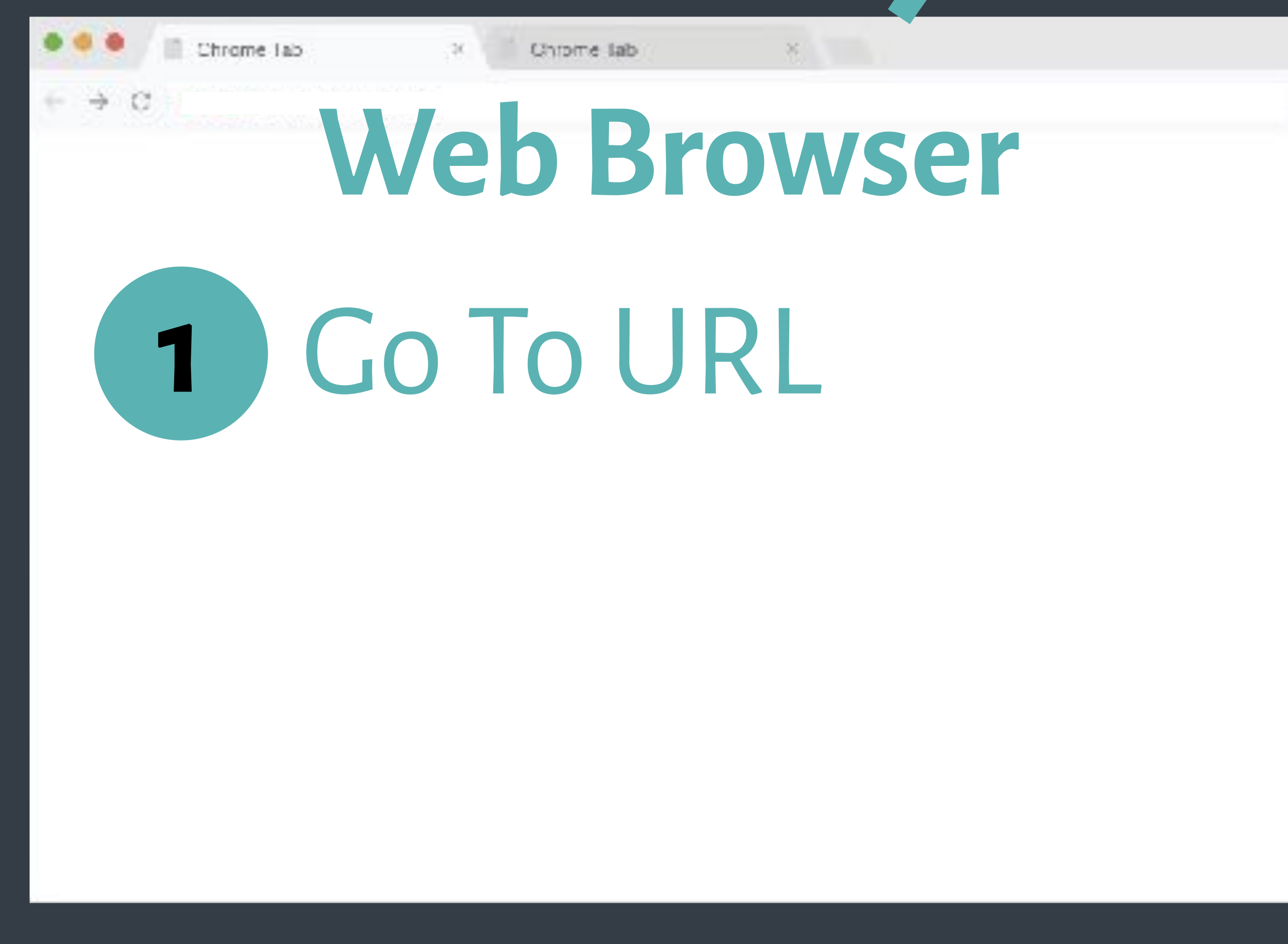
Protocol http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya

4 HTTP Request



5 Route Request

128.52.130.153



2 Lookup Domain Name

DNS

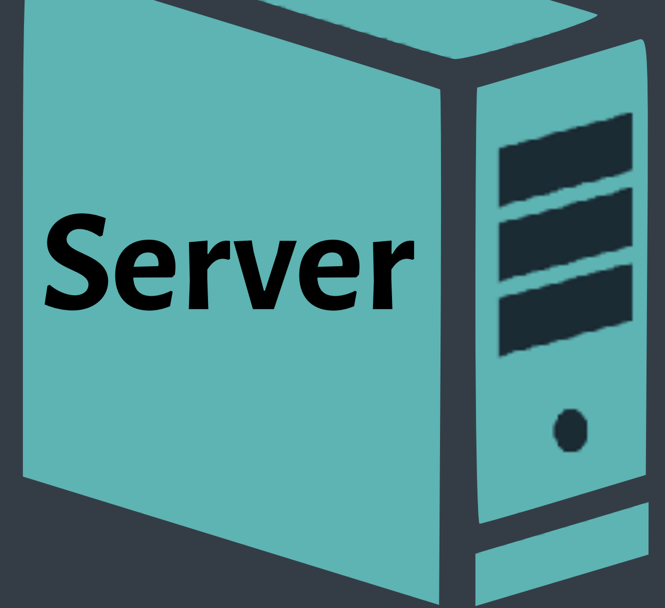
3 Return server's IP address

Host

Path

Protocol http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya

4 HTTP Request



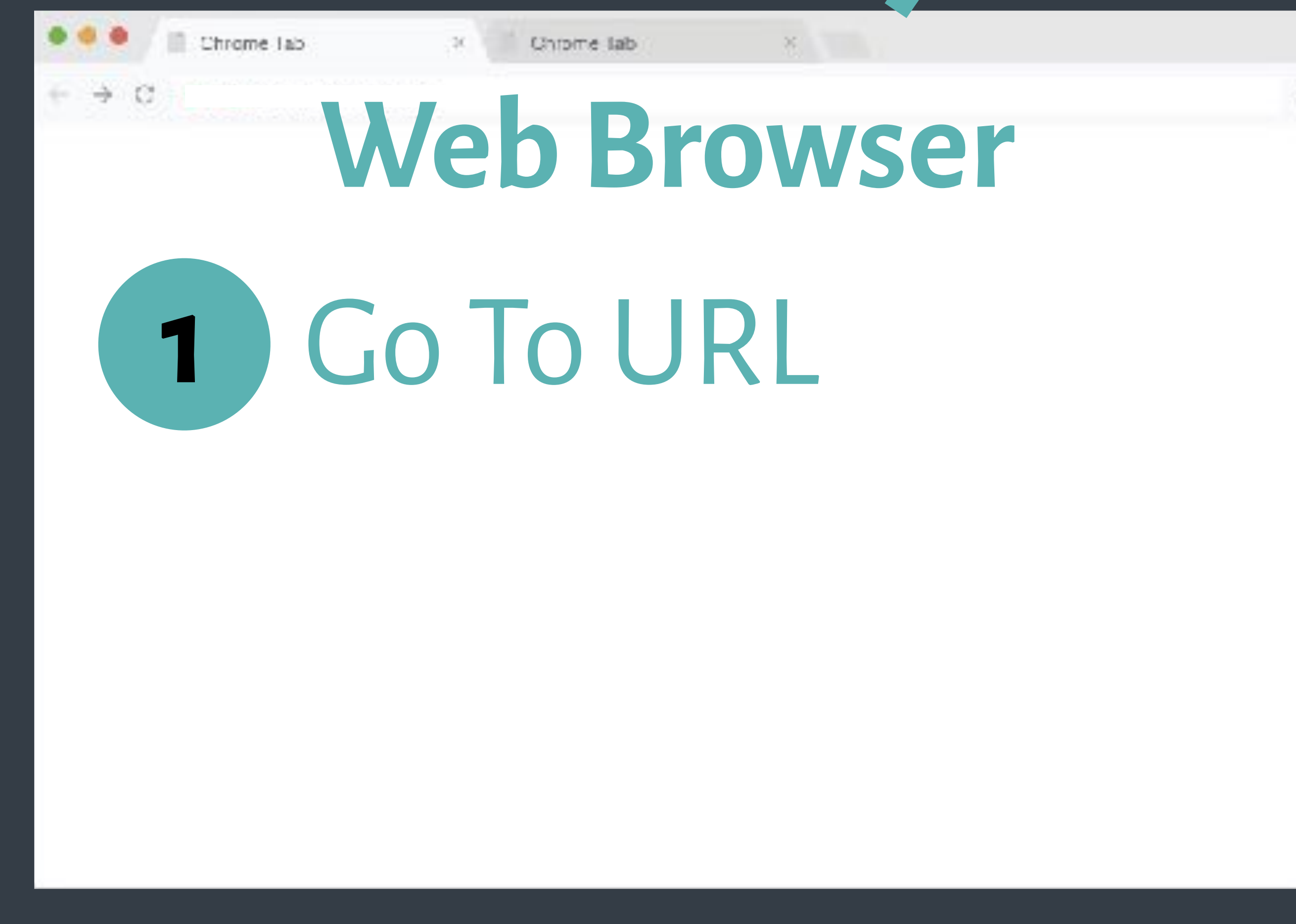
5 Route Request

128.52.130.153

DNS

2 Lookup Domain Name

3 Return server's IP address



Web Browser

1 Go To URL

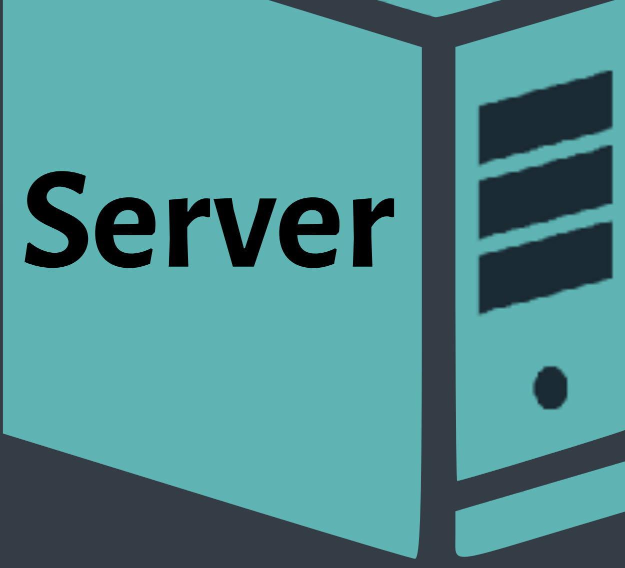
Host

Path

Protocol http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya

Query String

4 HTTP Request



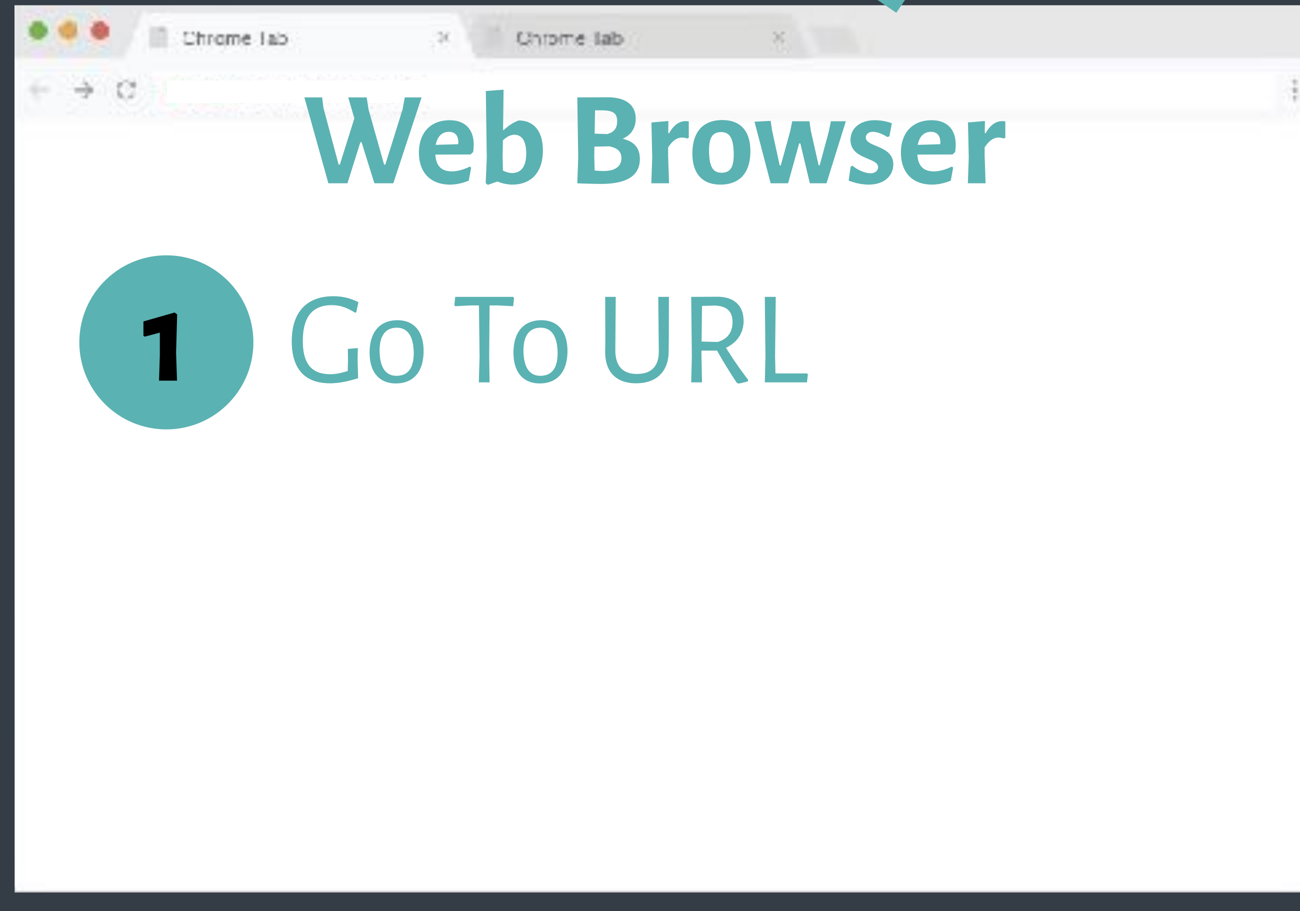
5 Route Request

128.52.130.153

DNS

2 Lookup Domain Name

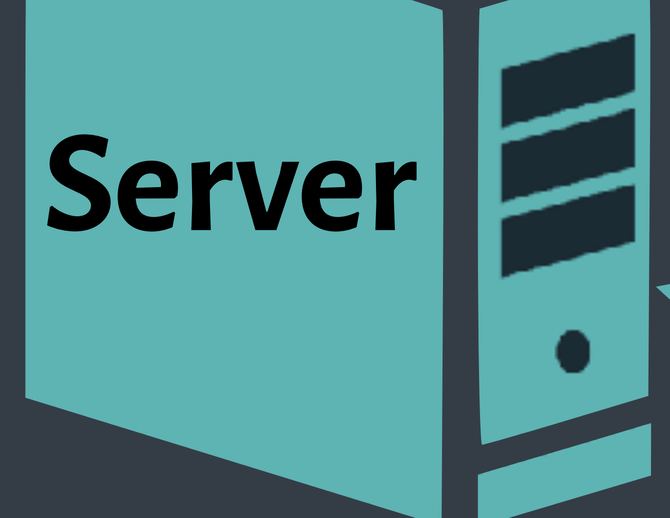
3 Return server's IP address



Protocol | Host | Path | Query String

http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya

4 HTTP Request

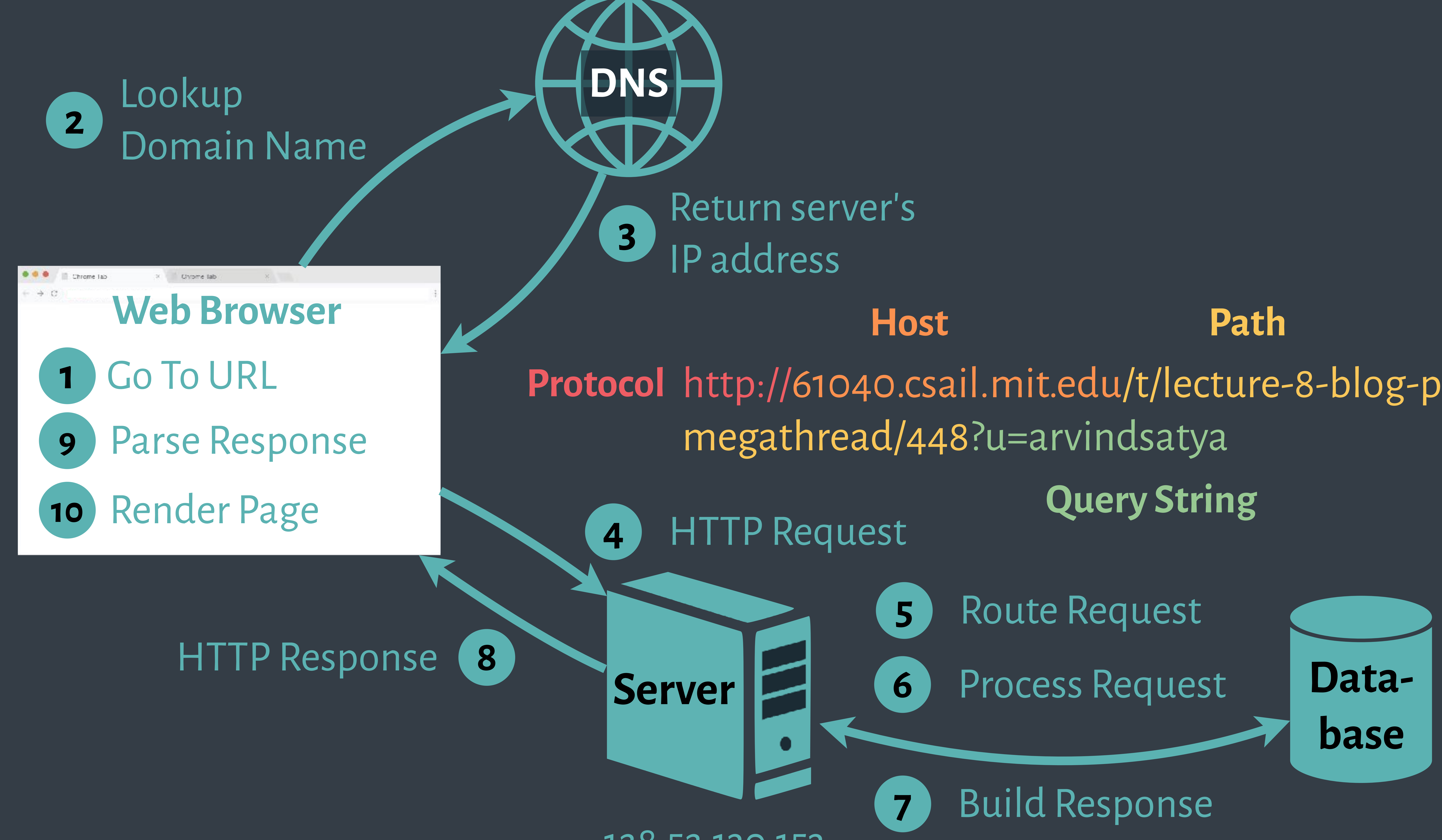


128.52.130.153

5 Route Request

6 Process Request

Database


DNS
2 Lookup Domain Name
3 Return server's IP address
Web Browser
1 Go To URL
9 Parse Response
10 Render Page
Host
Path
Protocol
http://61040.csail.mit.edu/t/lecture-8-blog-posts-megathread/448?u=arvindsatya
Query String
4 HTTP Request
HTTP Response 8
Server
128.52.130.153
5 Route Request
6 Process Request
7 Build Response
Data-base

Early web service URLs

```
/shopping_cart.asp?
action=update_qty&user=123

/postComment.jsp?
entryID=853&text=...

/services.php?
method=bid&item=236&...
```

Early web APIs were poorly designed:

- ✗ **Inconsistent:** APIs could be internally inconsistent. Different APIs might have different path/parameter conventions.
- ✗ **Difficult to maintain/extend.**
- ✗ **Not easily discoverable:** what goes in the path, what goes in the query parameters?

RESTful Design

Representational Transfer

RESTful Design

State

RESTful Design

"Applying verbs to nouns"

GET /profs/arvind

GET /profs/arvind

Noun aka Resource aka Concept
(U**R**L)

URL paths identify a *representation* of a resource

Profile page: `/profs/arvind.html`
Profile picture: `/profs/arvind.jpg`
Data structure: `/profs/arvind.json`

GET /profs/arvind

Use hierarchies to imply *structure*

collections

```
/profs
/profs/reviews
/profs/arvind/reviews
/profs/arvind/reviews?n=5
```

instances

```
/profs/arvind
/profs/arvind/reviews/275
```

GET /profs/arvind

Noun aka Resource

(U**R**L)

GET /profs/arvind

Verb
(HTTP Method)

Noun aka Resource
(U**R**L)

GET /profs/arvind

HTTP methods imply *different actions* on the *same resource*.

```
Create  POST   /profs/arvind/reviews
Read    GET    /profs/arvind/reviews
Update  PUT    /profs/arvind/reviews/4
Delete  DELETE /profs/arvind/reviews/5
```

```
POST /profs/arvind/reviews
```

Body

```
{
  "rating": 5,
  "text": "A+ taste in music",
  "date": 2023-09-25
}
```

GET /profs/arvind

HTTP methods imply *different actions* on the *same resource*.

```
Create   POST   /profs/arvind/reviews
Read     GET    /profs/arvind/reviews
Update   PUT    /profs/arvind/reviews/4
Delete   DELETE /profs/arvind/reviews/5
```

Safe methods *do not change* the resource.

Idempotent methods can be *called multiple times* and *always produce the same result*.

And help us think about *data safety*

Method	Safe	Idempotent
GET	✔	✔
POST	✘	✘
PUT	✘	✔
DELETE	✘	✔

GET /profs/arvind

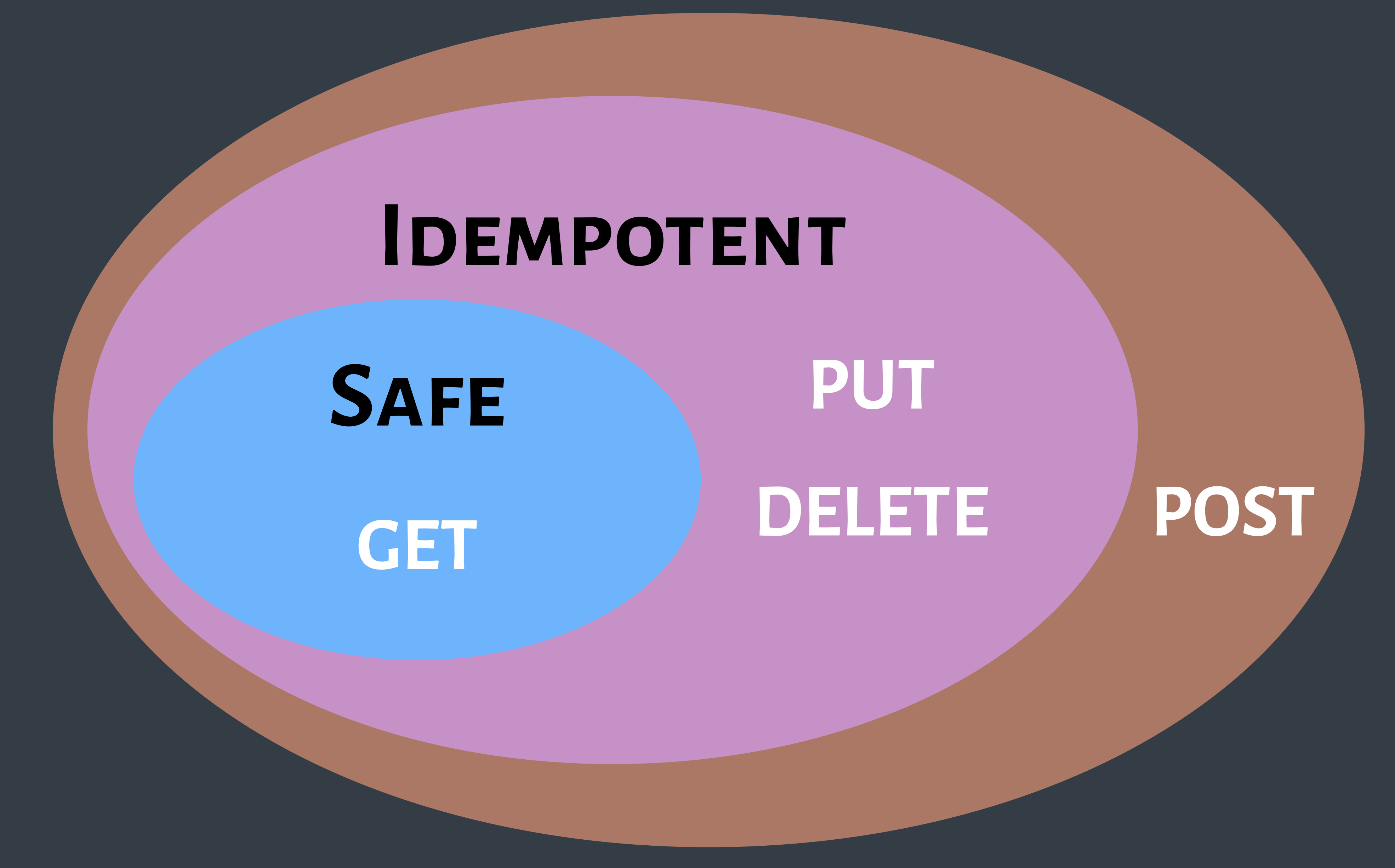
HTTP methods imply *different actions* on the *same resource*.

Create	`POST /profs/arvind/reviews`
Read	`GET /profs/arvind/reviews`
Update	`PUT /profs/arvind/reviews/4`
Delete	`DELETE /profs/arvind/reviews/5`

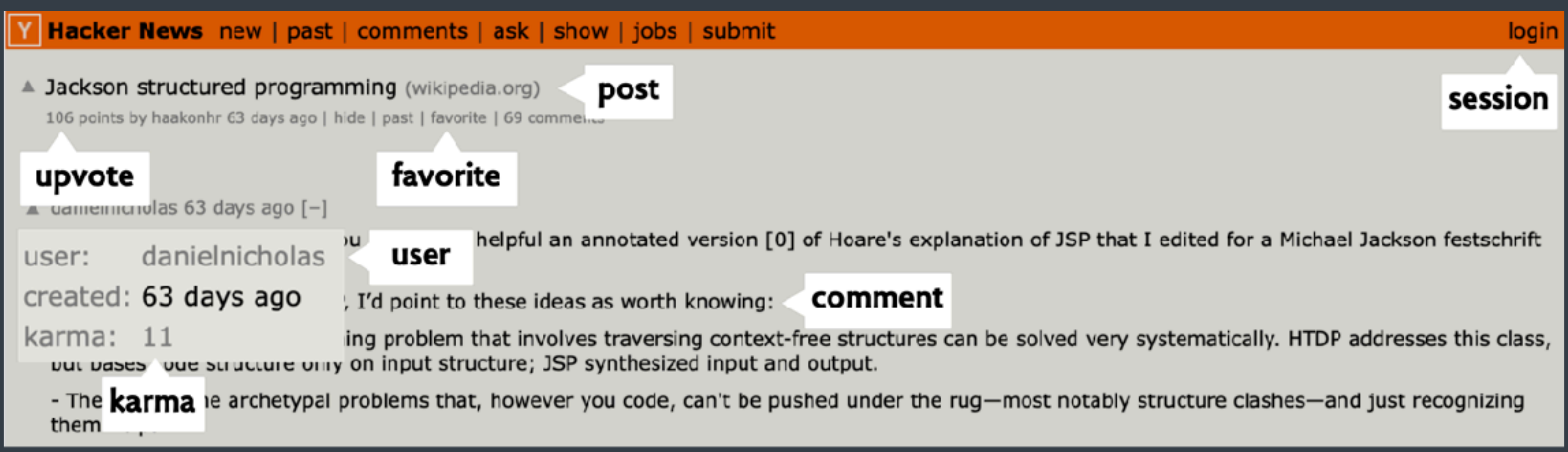
And help us think about *data safety*

Safe methods *do not change* the resource.

Idempotent methods can be *called multiple times* and *always produce the same result*.



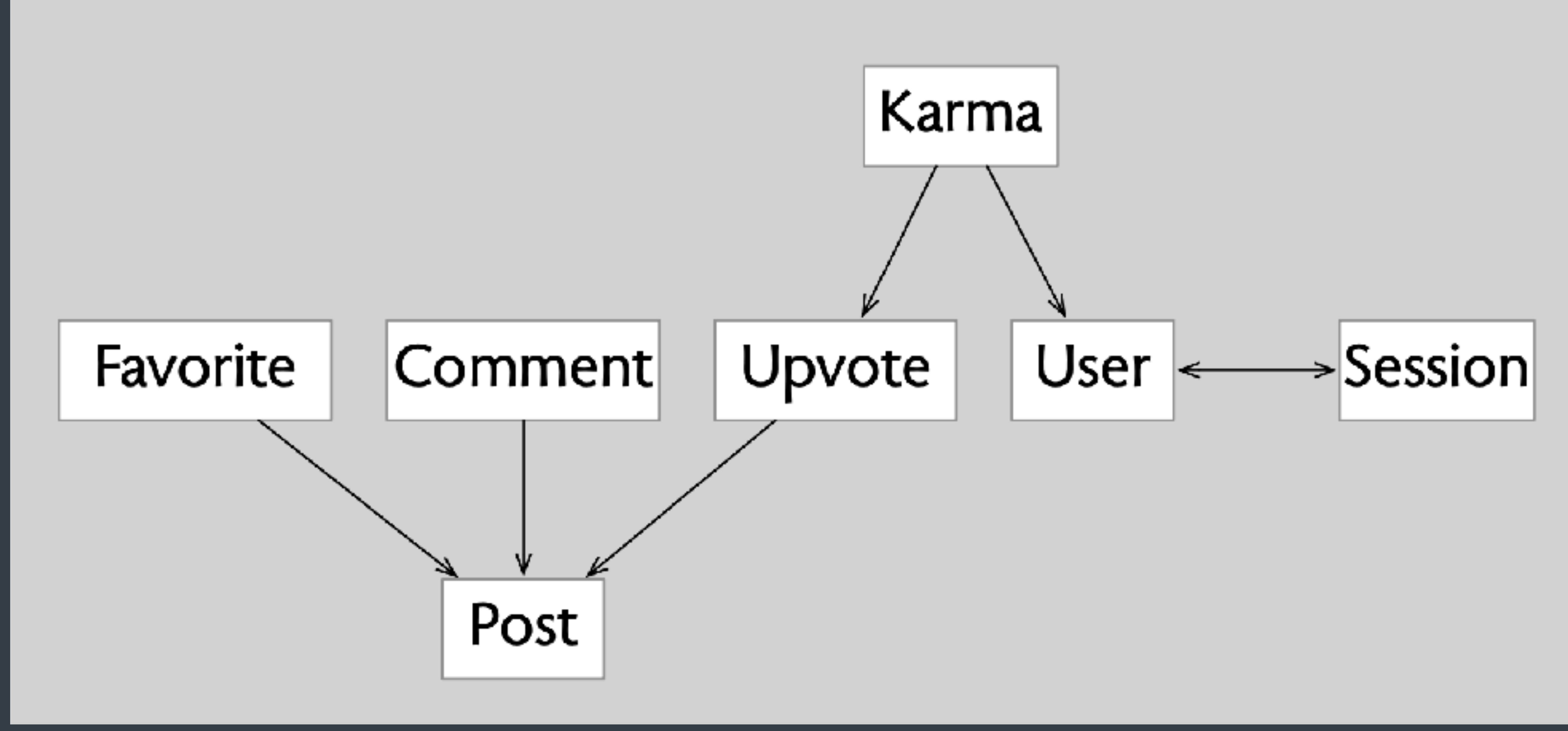
5-min Exercise: Design a RESTful API for Hacker News



What are the various API paths?

How are they nested to convey relationships between concepts?

What is the effect of different HTTP methods (verbs)?



User

```
GET /users/[username]
…
```

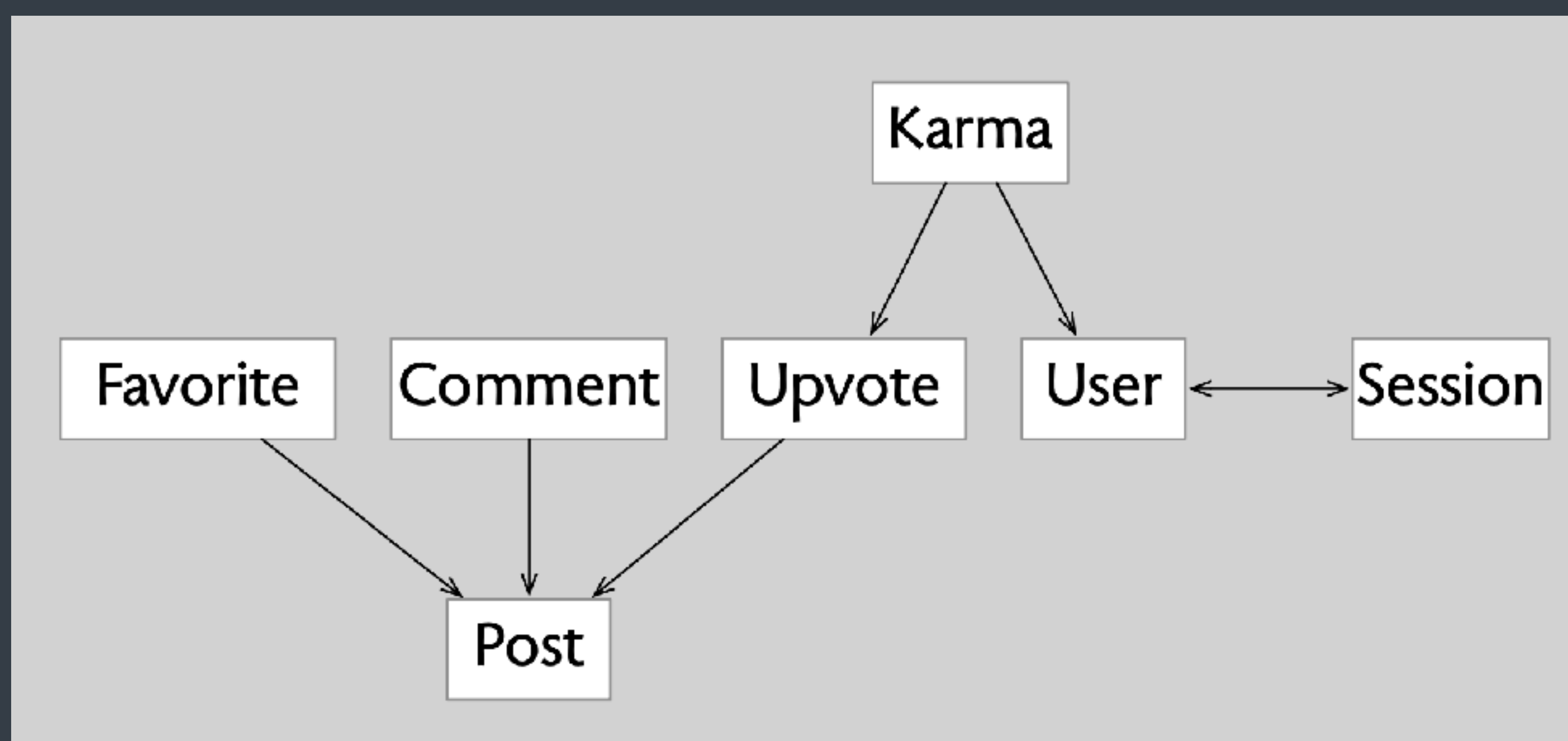
Session

```
POST /users/[username]/session
```

or

```
POST /session?user=[username]
```

5-min Exercise: Design a RESTful API for Hacker News



User/Session

```
GET /users/[username]
POST /users/[username]/session
POST /session?user=[username]
DELETE /users/[username]
```

Post

Karma

```
GET /users/[username]/karma
GET /karma?user=[username]&upvoteID
```

Favorite/Comment

```
DELETE /user/[username]/[postID]?type=comment
```

Upvote

```
GET /post/[postID] --> {....., upvote: }
GET /post/[postID]/upvotes
PUT /post/[postID]
```

Documentation

Twitter API v2 | Twitter API: Enterprise | Twitter API: Standard v1.1 | Tw Ad

Twitter API v2

Tweets

Bookmarks

- DELETE /2/users/:id/bookmarks/:tweet_id
- GET /2/users/:id/bookmarks
- POST /2/users/:id/bookmarks

Filtered stream

- GET /2/tweets/search/stream
- GET /2/tweets/search/stream/rules
- POST /2/tweets/search/stream/rules

Hide replies

- PUT /2/tweets/:id/hidden

Likes

- DELETE /2/users/:id/likes/:tweet_id
- DELETE /2/users/:id/likes/:tweet_id
- GET /2/tweets/:id/liking_users

Search

← Tableau Help > Tableau REST API Help > API Reference—All Methods

IN THIS ARTICLE

API Method Categories

The following table lists the Tableau Server REST API meth
table also indicates which methods can be used with Tablea

- Analytics Extensions Settings Methods
- Ask Data Lens Methods
- Authentication Methods
- Connected App Methods
- Content Exploration Methods
- Dashboard Extensions Settings Methods
- Data Sources Methods
- Extract and Encryption Methods

qualtrics XM | Instructions | Guides | API Reference | SDKs | Search...

- Surveys CRUD API
  - Surveys
    - Get Survey GET
    - Update Survey PUT
    - Delete Survey DELETE
    - List Surveys GET
    - Import Survey POST
    - Share Survey POST
  - Survey Embedded Data Fields
  - Survey Quotas
  - Schemas
- Ticketing API
- Transaction Batches
- Users
  - Users
    - List Users GET
    - Create User POST
    - Get User GET
    - Delete User DELETE
    - Update User PUT
    - Who Am I GET
  - Users API Tokens
  - Schemas
- WhatsApp Distributions

Surveys CRUD API

Export

v3.0.0

API Base URL

Canadian Data Center: https://yul1.qualtrics.com/API/v3
Washington, DC Area Data Center (previously CO1): https://iad1.qualtri
San Jose, California Data Center (previously AZ): https://sjc1.qualtrics.
European Union Data Center (previously EU2 or EU): https://fra1.qualtri
London, United Kingdom Data Center: https://lhr1.qualtrics.com/API/v3
Sydney, Australia Data Center (previously AU1): https://syd1.qualtrics.c
Singapore Data Center: https://sin1.qualtrics.com/API/v3
Tokyo, Japan Data Center: https://hnd1.qualtrics.com/API/v3
US Government Data Center: https://gov1.qualtrics.com/API/v3
Mock Server: https://stoplight.io/mocks/qualtricsv2/publicapidocs/60937

Security

▾ API Key

This is a schema for x-api-token header authentication.

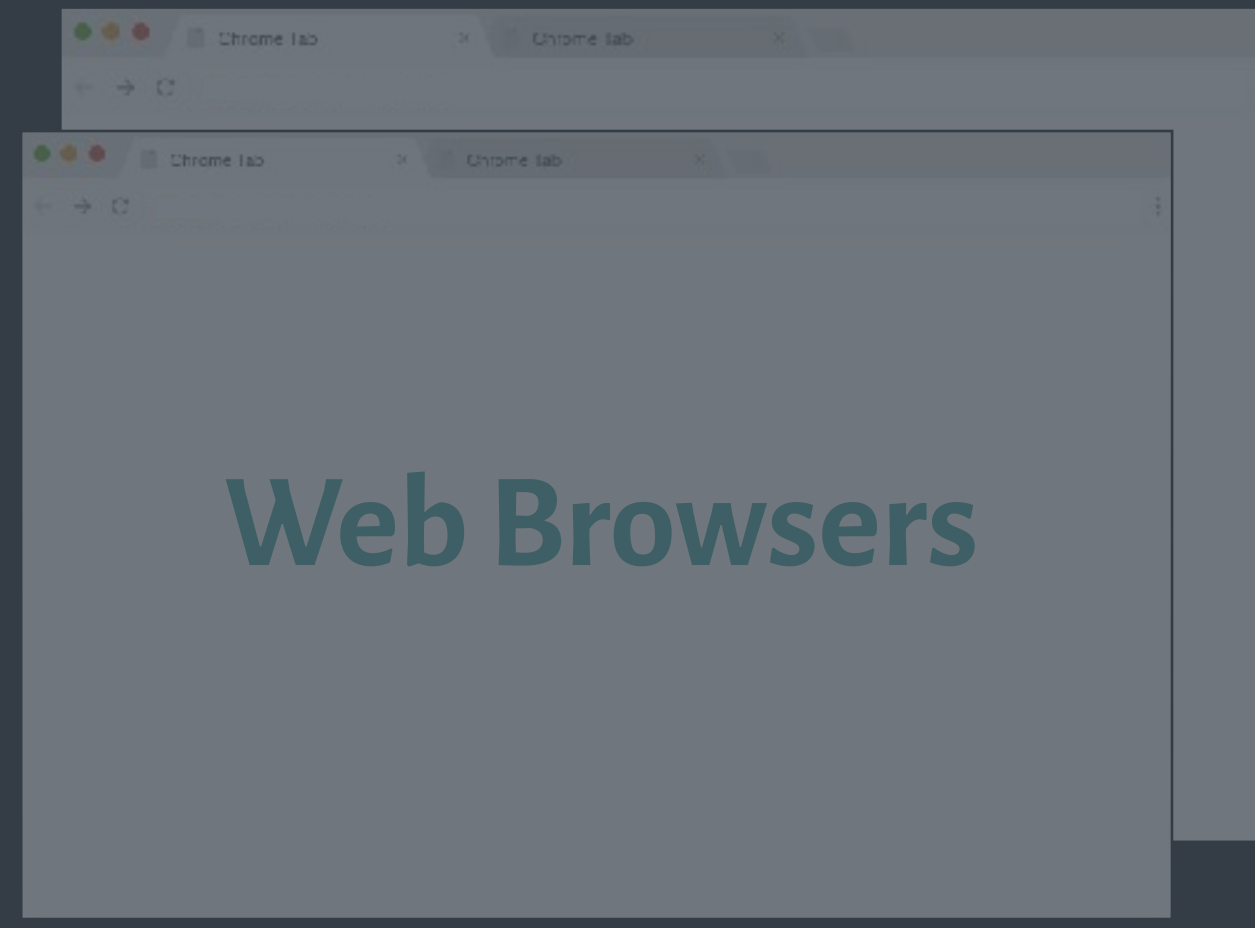
An API key is a token that you provide when making API calls. Include the token in a header parameter called `X-API-TOKEN`.

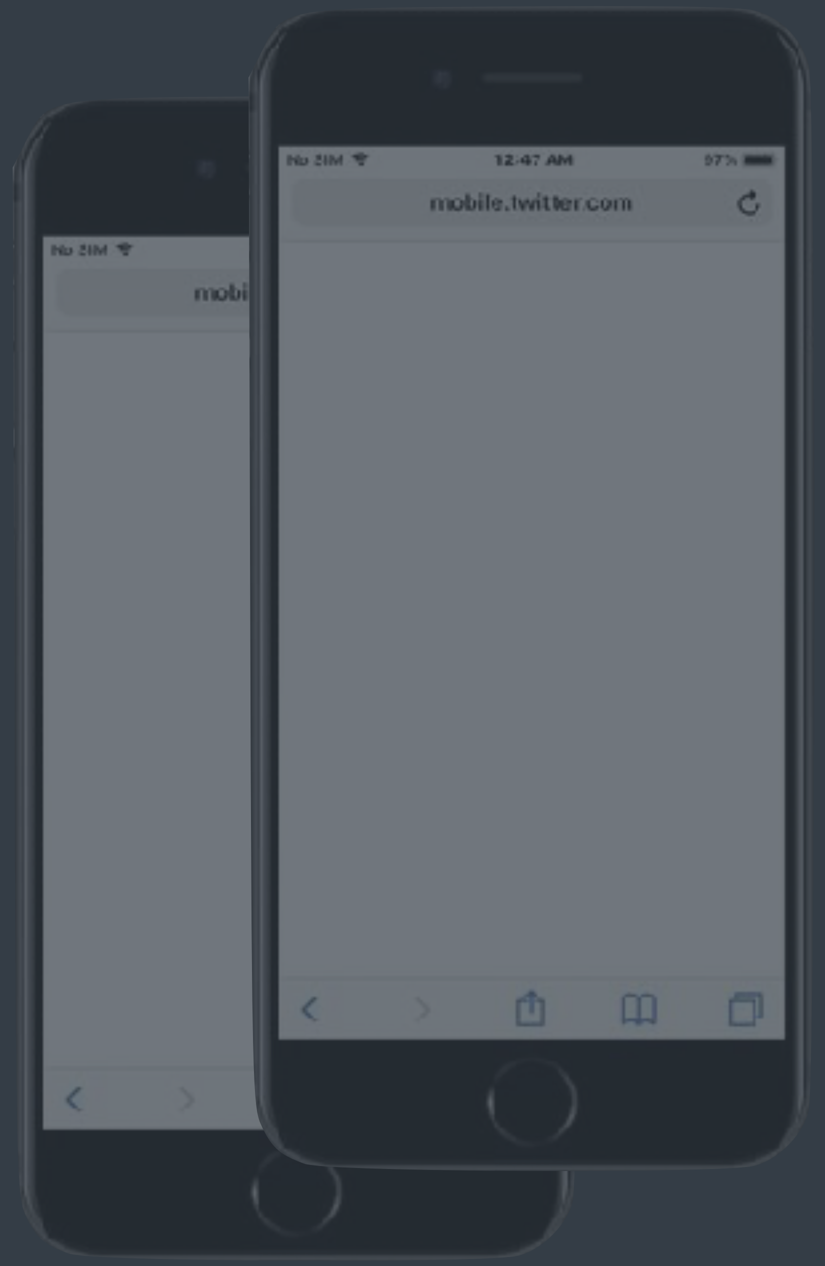
Example: `X-API-TOKEN: 123`

▸ OAuth 2.0

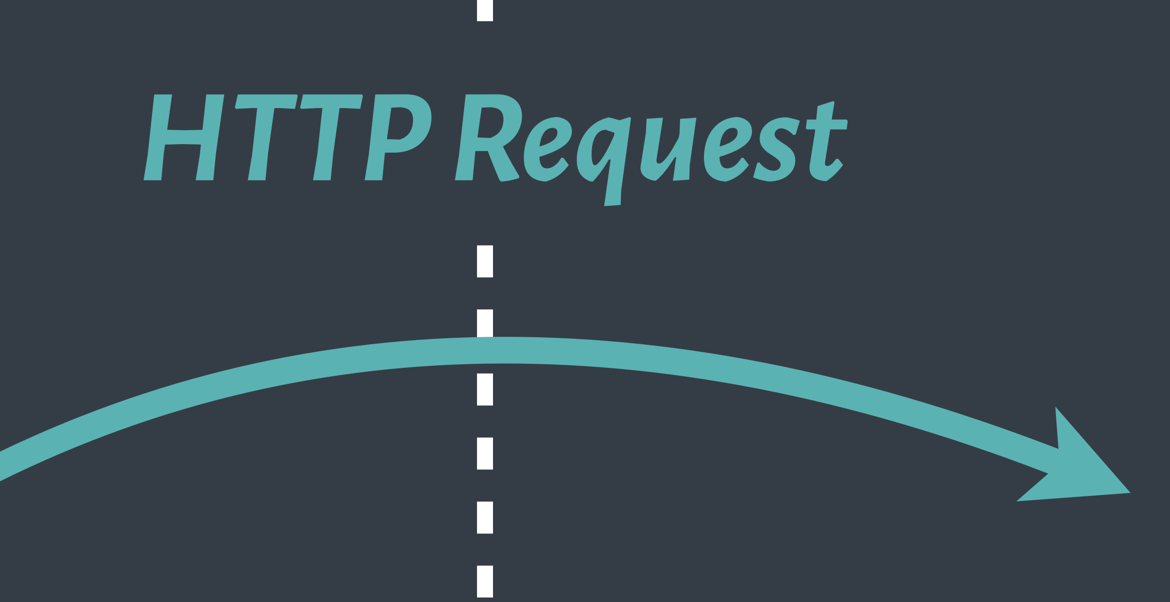
Client Side

Server Side

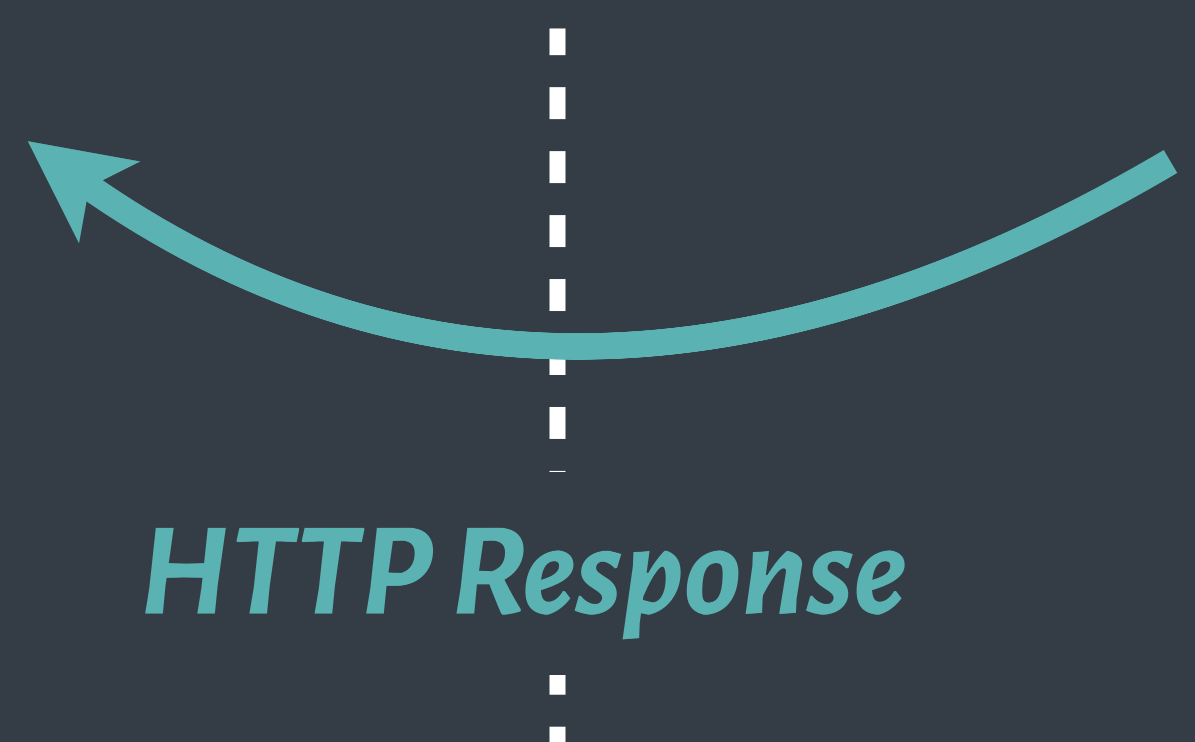


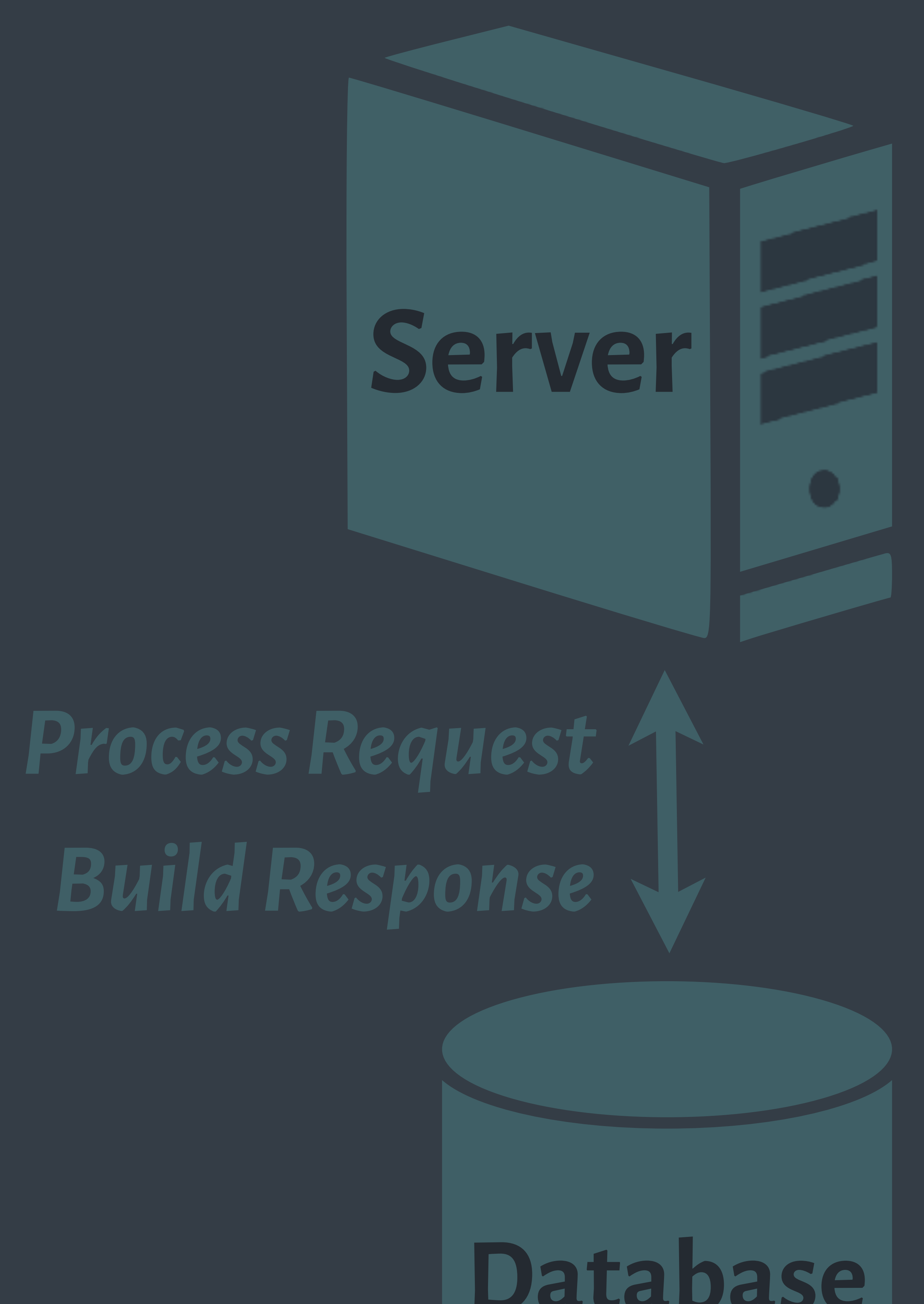


Mobile Devices



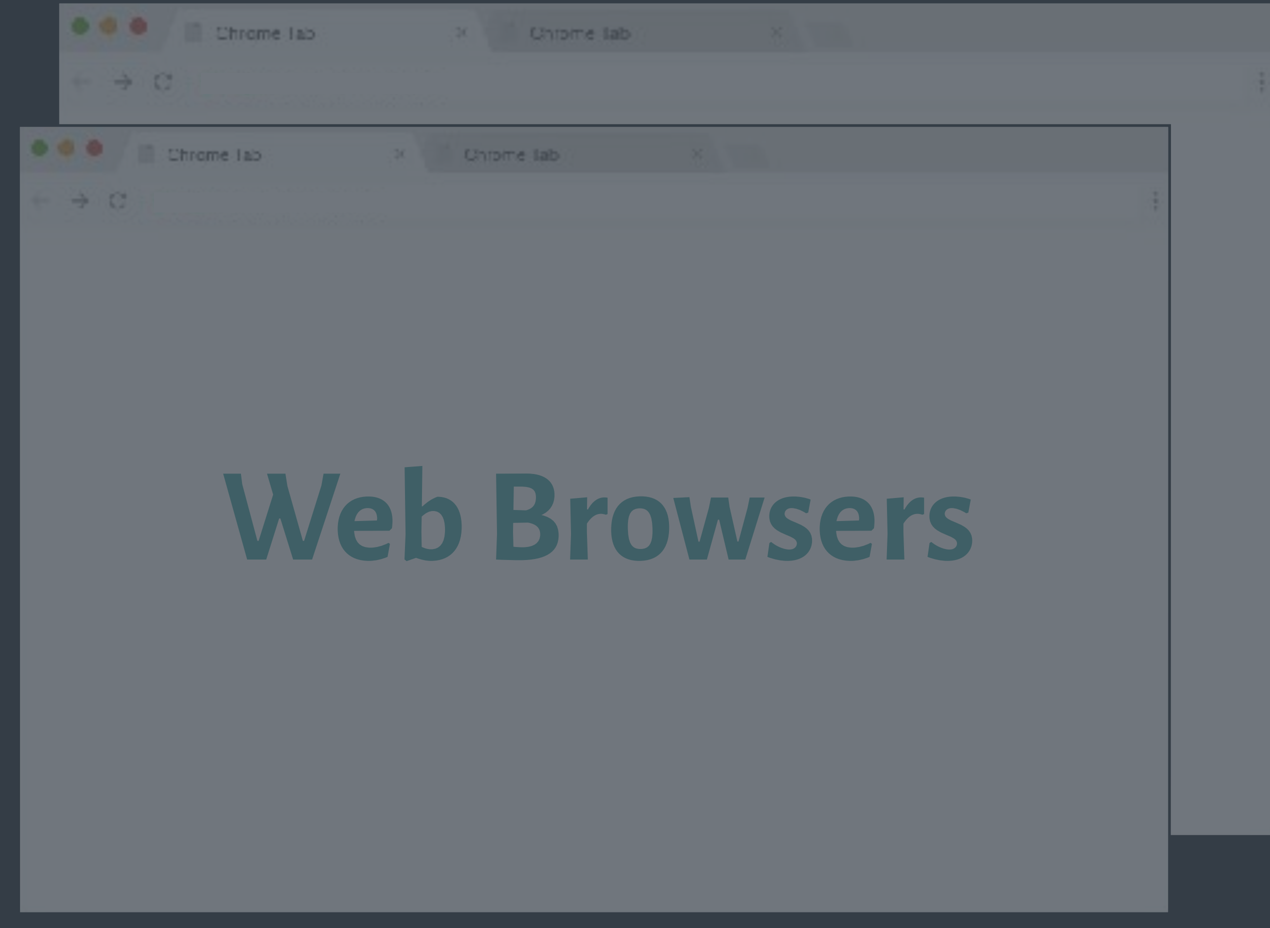
URLs are an **interface** that **require design**

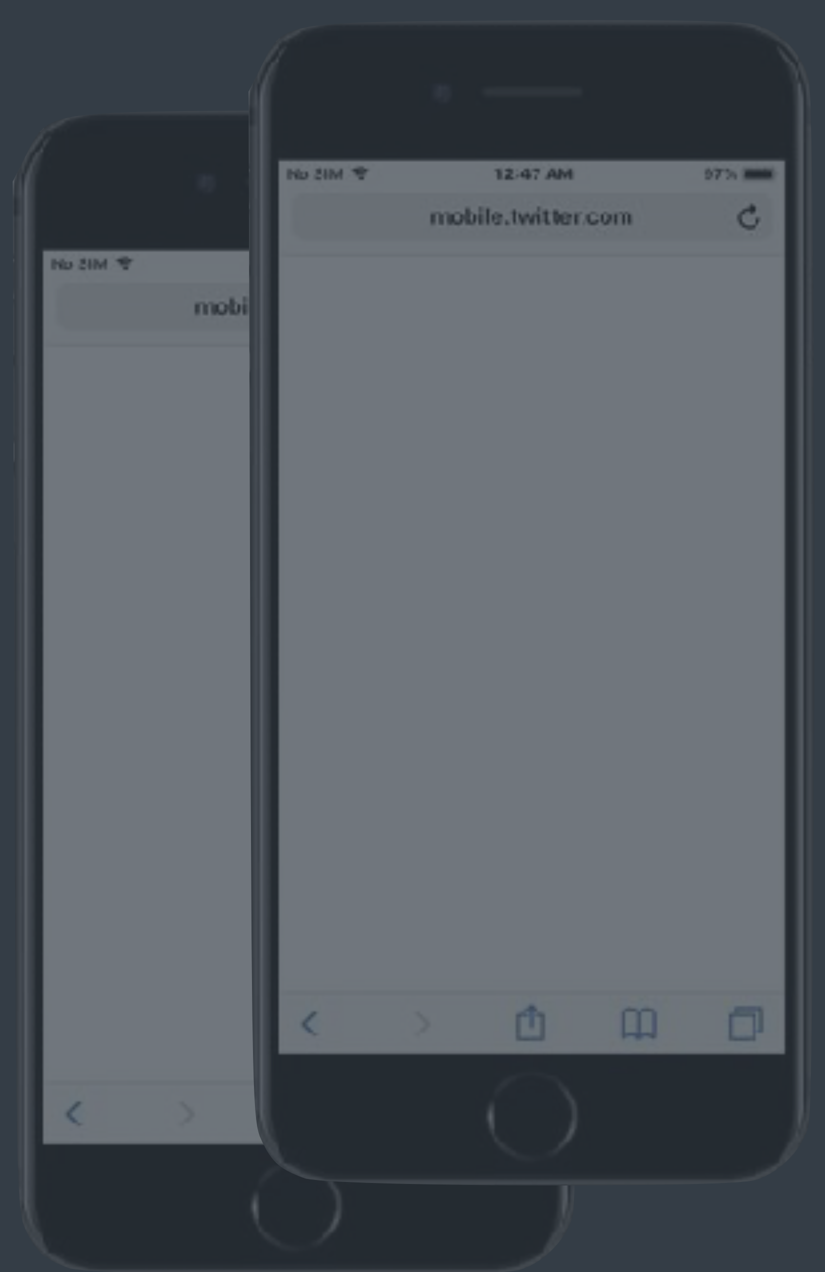




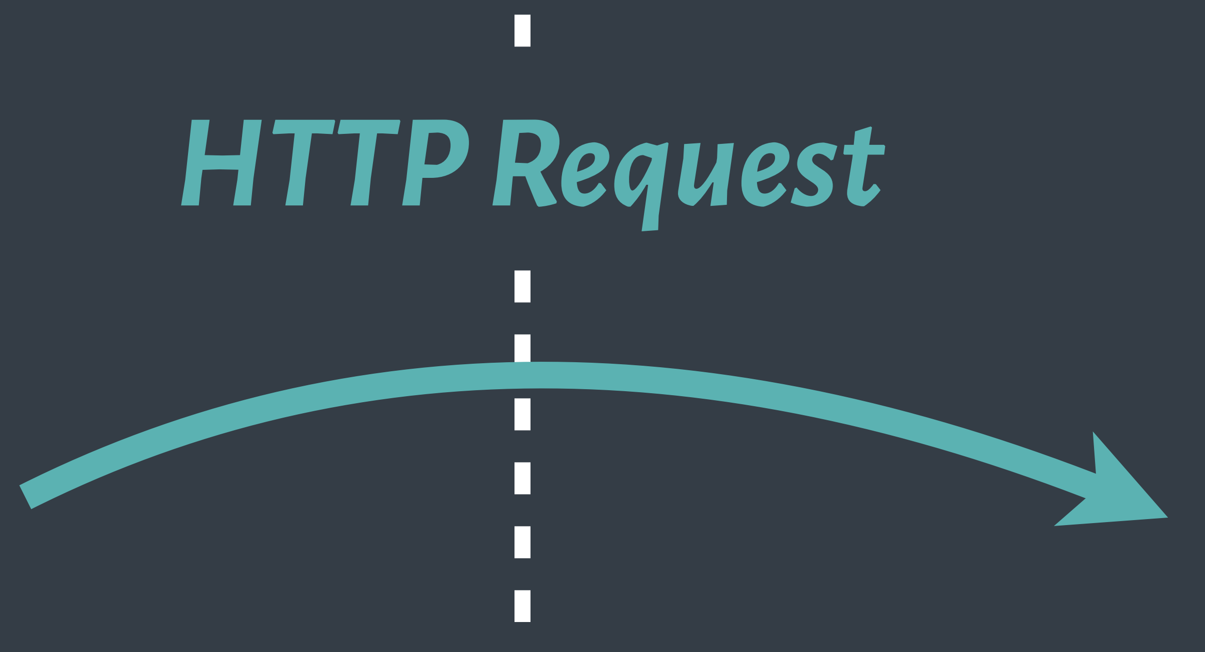
Client Side

Server Side

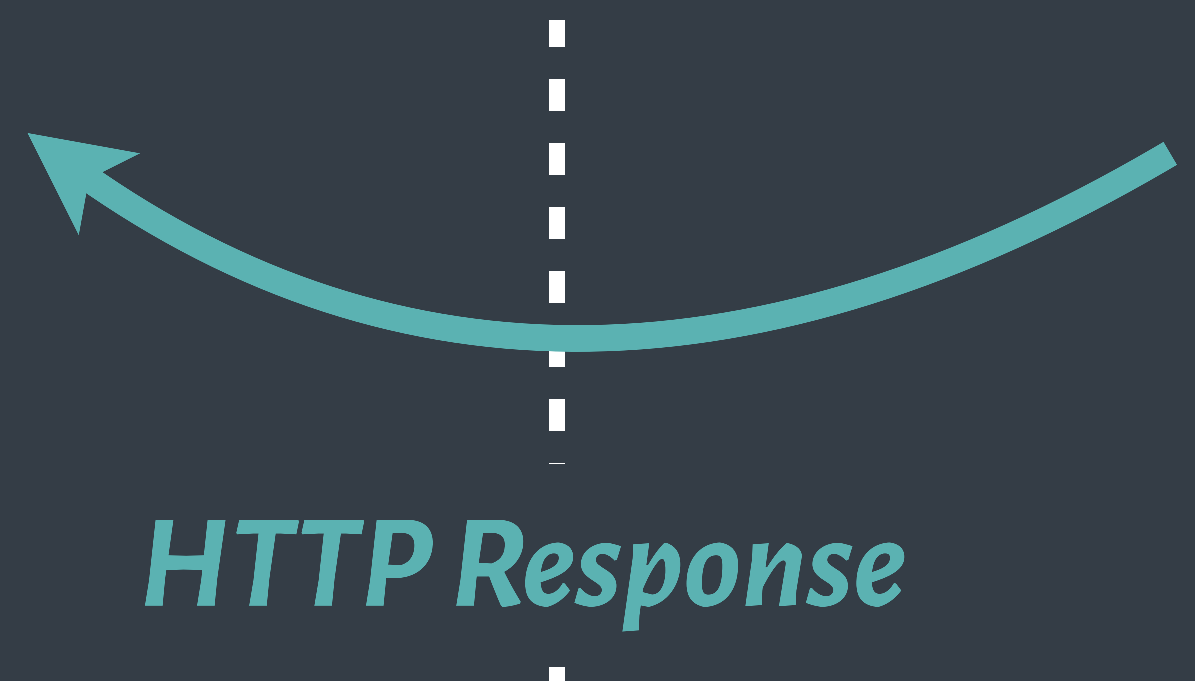


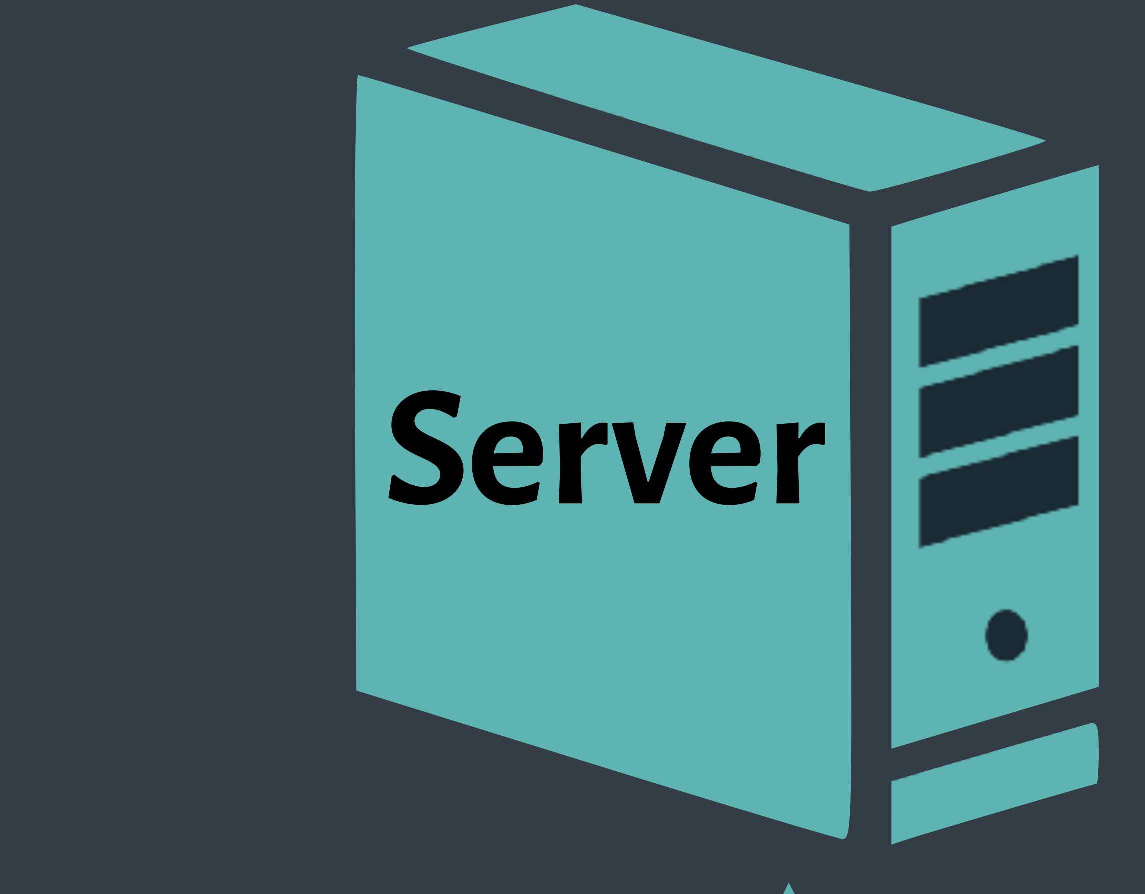


Mobile Devices



URLs are an **interface** that **require design**



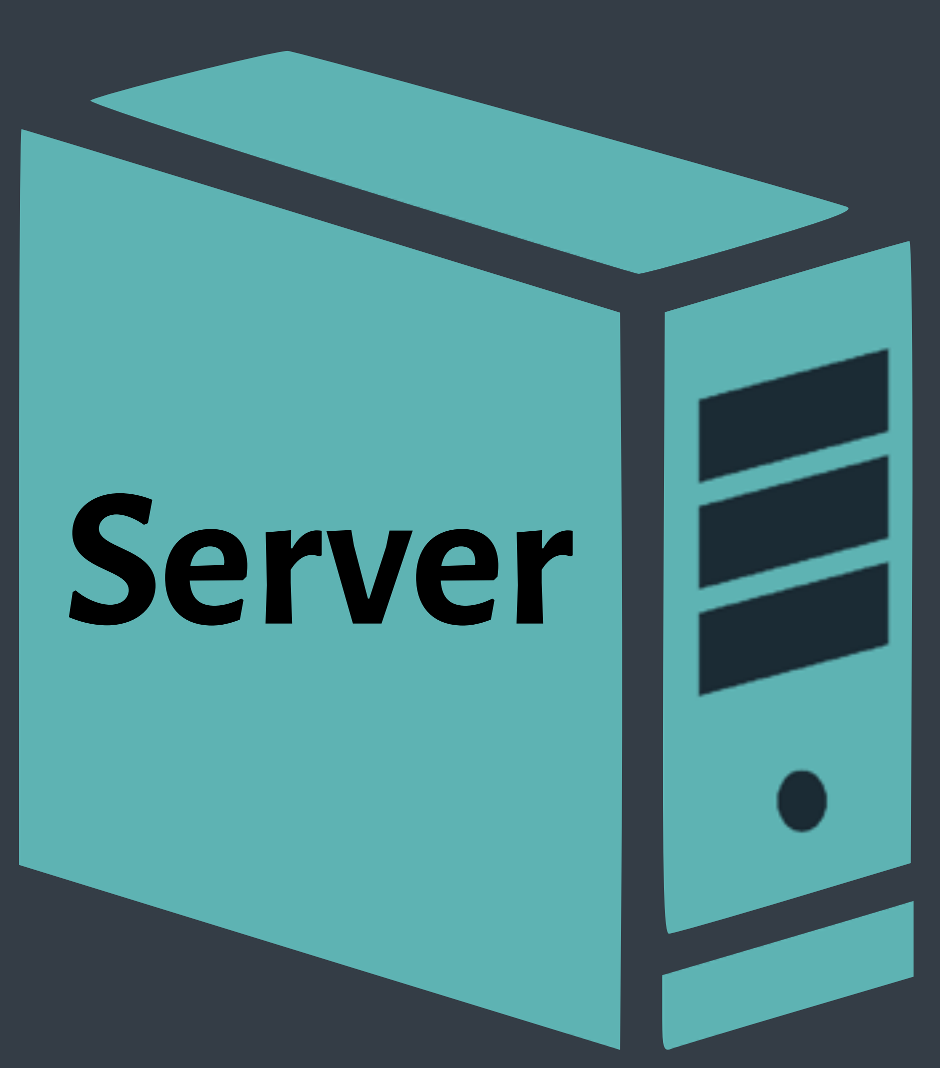


Process Request

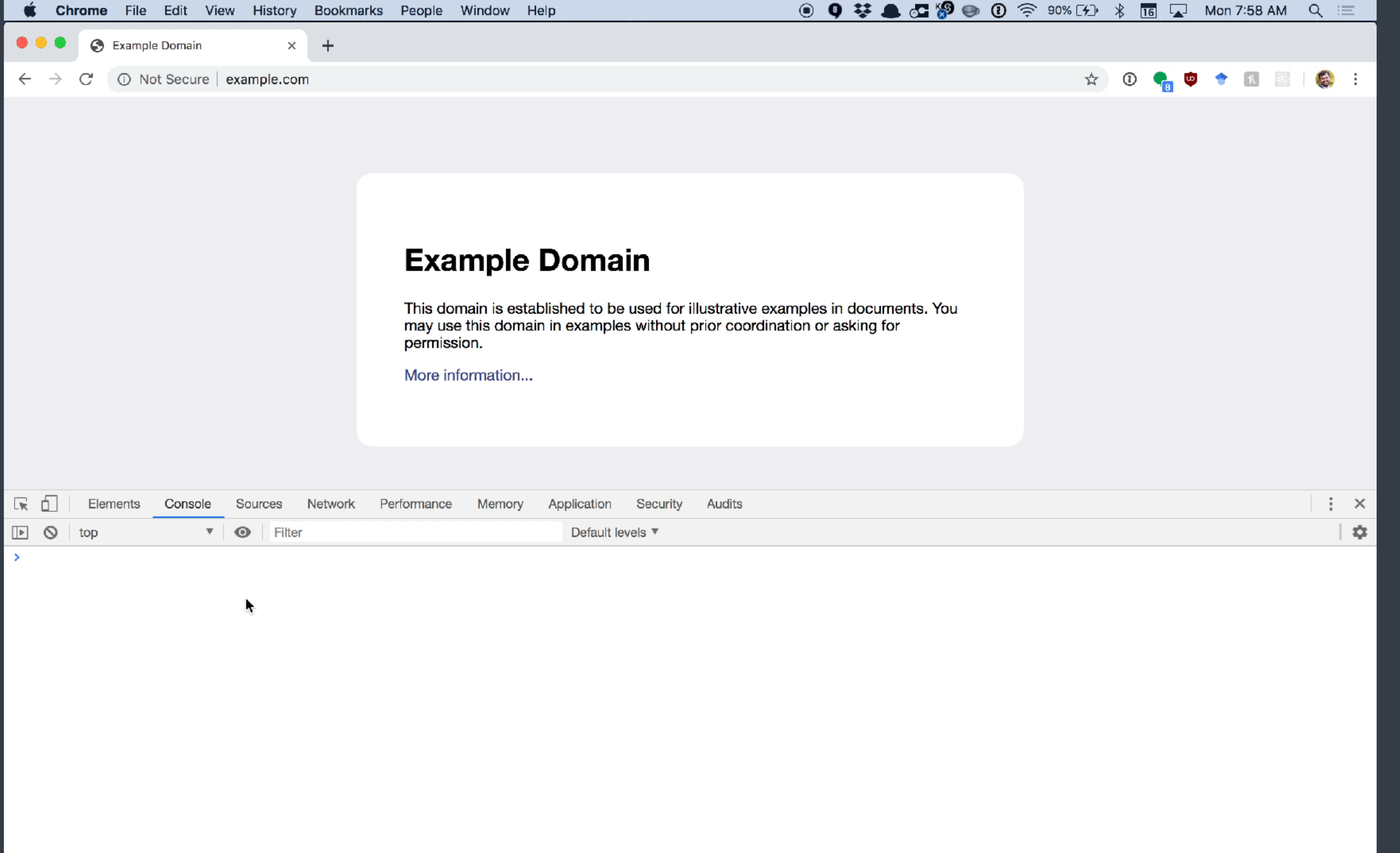
Build Response

Why node.js over other technology?

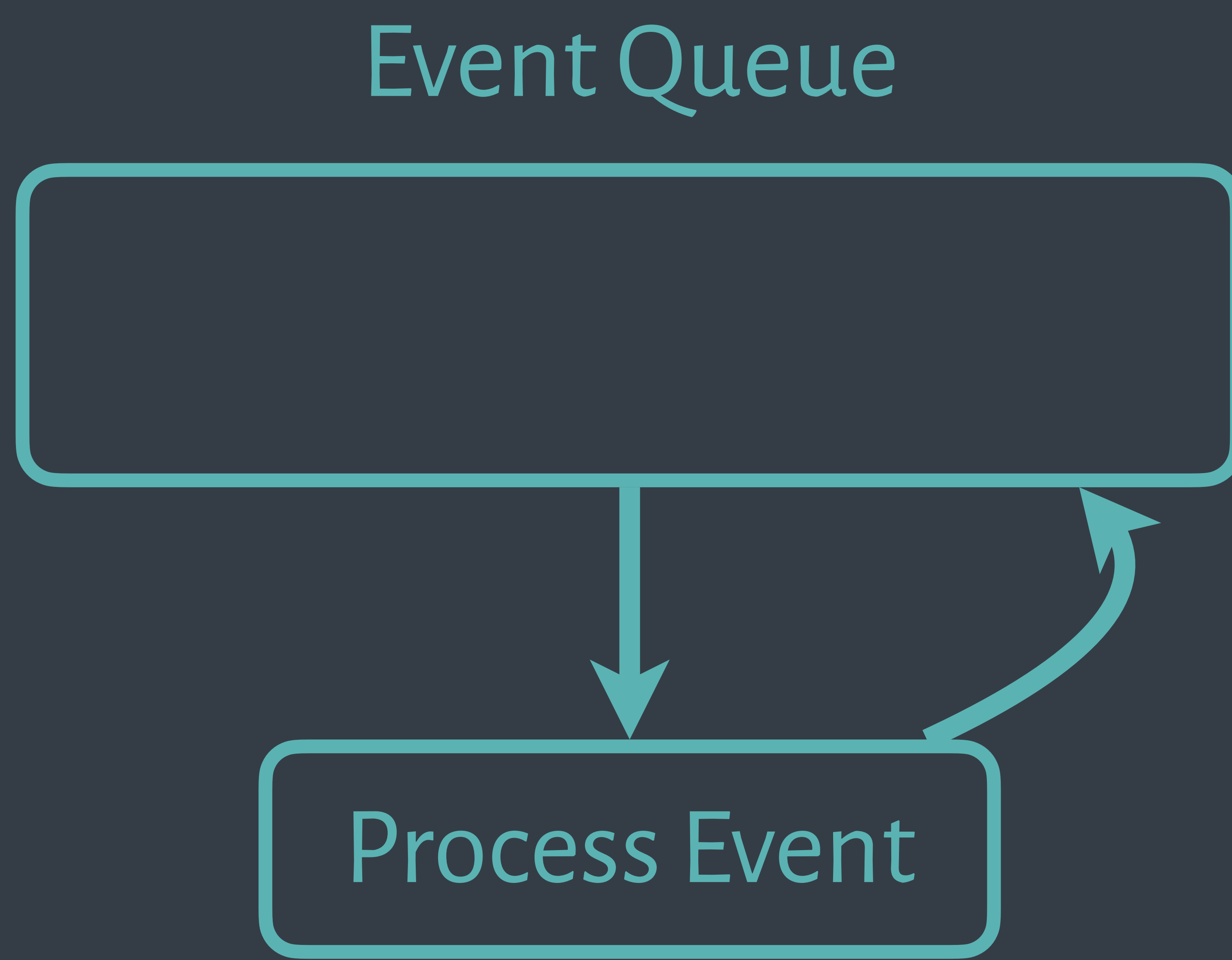
- ✓ Use a single language (JavaScript) across client and server.
- ✓ Fill gaps in core JavaScript (modules, input/output, networking).
- ✓ Rich community & ecosystem = many modules available on npmjs.com.
- ✓ Very little magic (vs. e.g., Rails).



Javascript is single threaded


Chrome File Edit View History Bookmarks People Window Help
Mon 7:58 AM
Example Domain
Not Secure
example.com
Example Domain
This domain is established to be used for illustrative examples in documents. You may use this domain in examples without prior coordination or asking for permission.
More information...
Elements Console Sources Network Performance Memory Application Security Audits
top
Filter
Default levels

The JavaScript *Event Loop*



Stack Frame

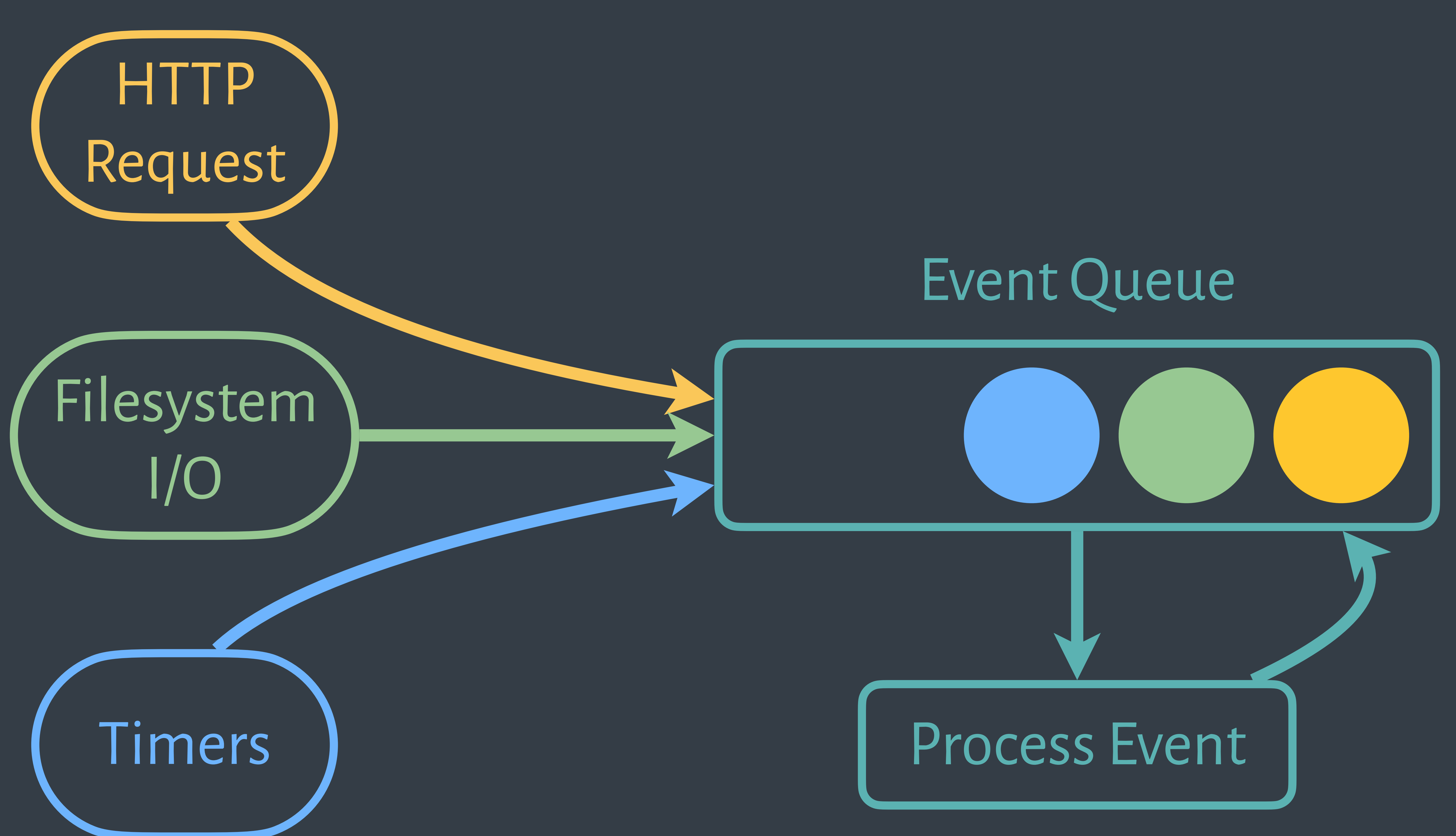
Stack Frame

Stack Frame

Stack Frame

Call Stack

The JavaScript *Event Loop*



Stack Frame

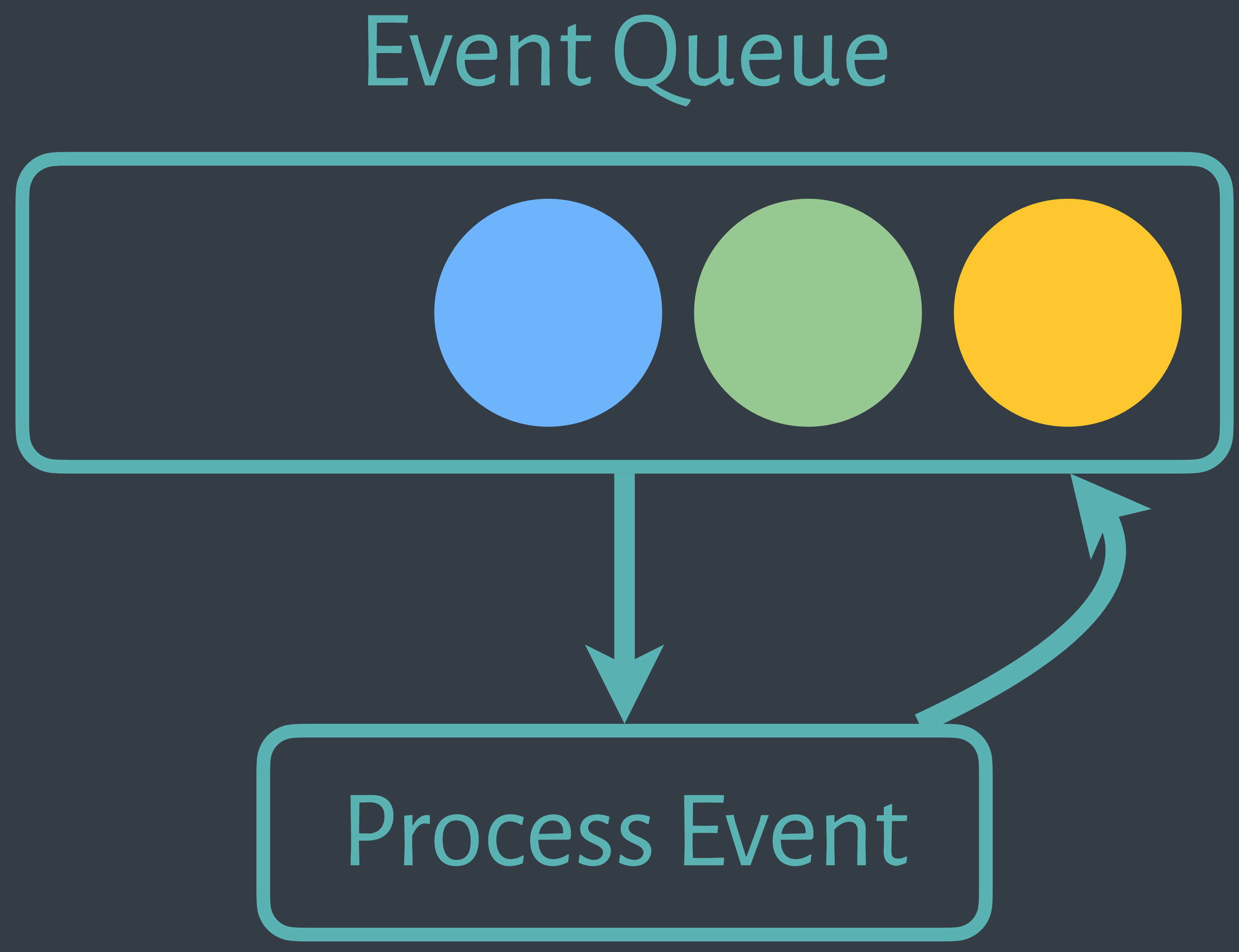
Stack Frame

Stack Frame

Stack Frame

Call Stack

The JavaScript *Event Loop*



Event Handler

Event Handler

Event Handler

Stack Frame

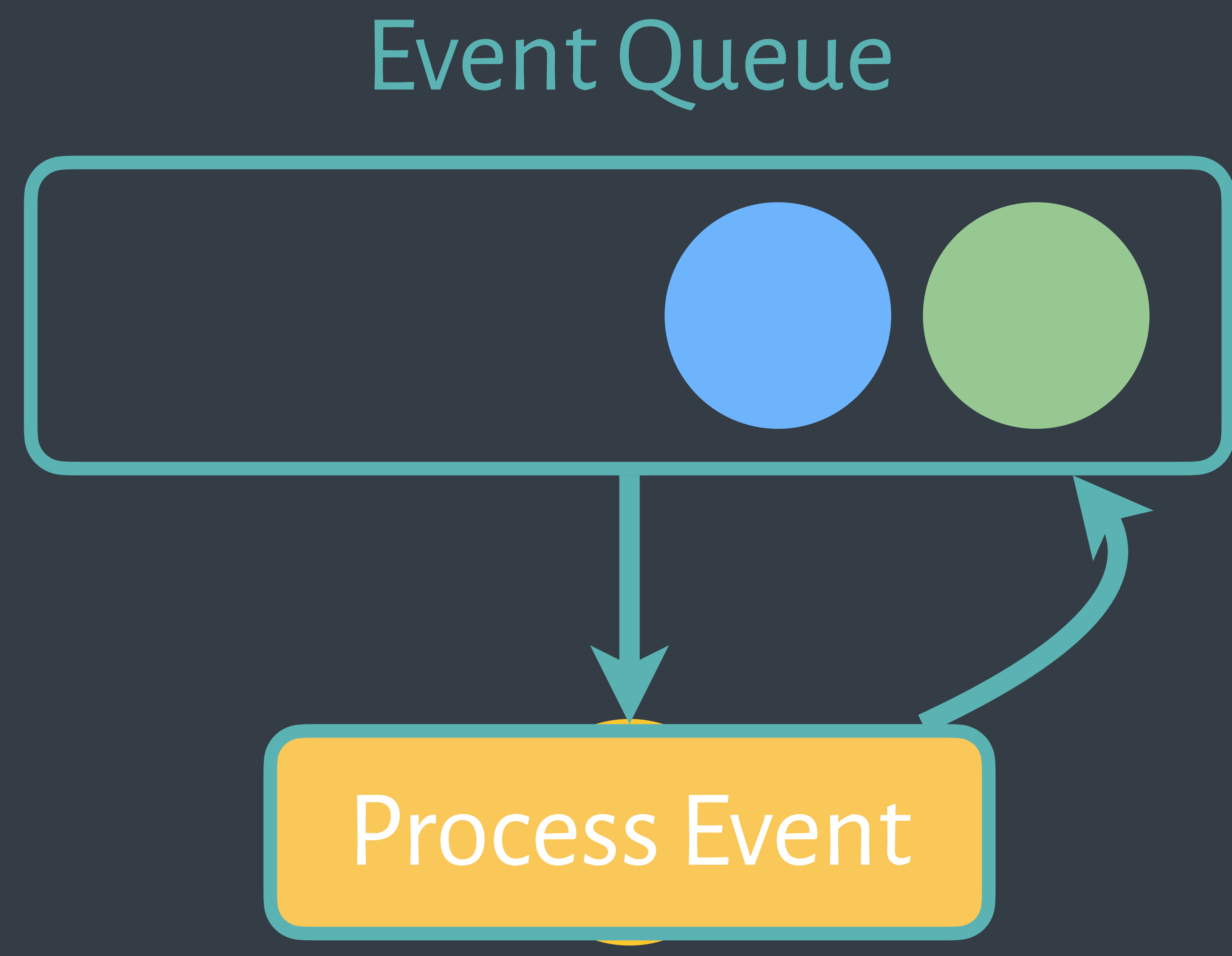
Stack Frame

Stack Frame

Stack Frame

Call Stack

The JavaScript *Event Loop*



Stack Frame

Stack Frame

Stack Frame

Event Handler

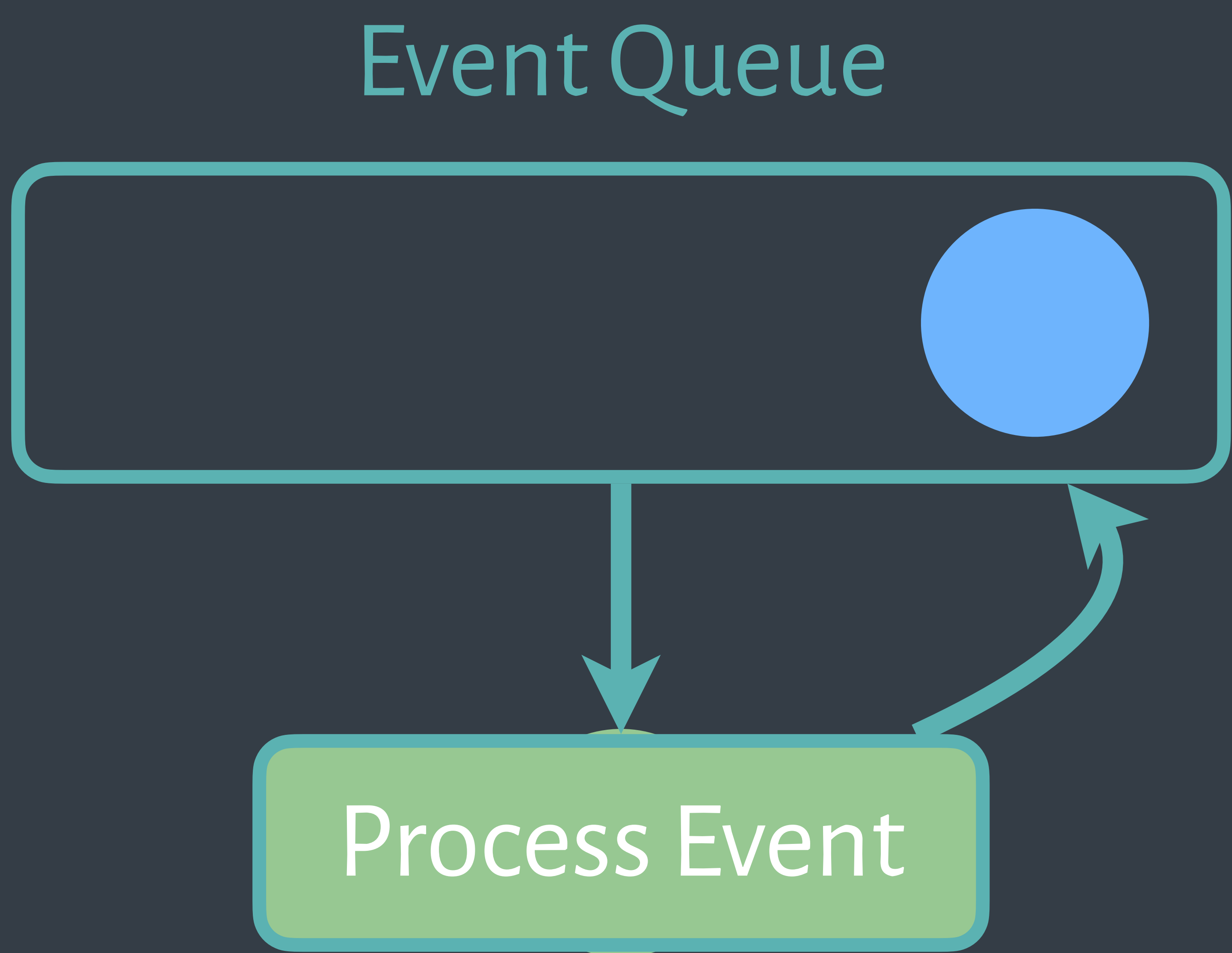
Call Stack

Event Handler

Event Handler

Event Handler

The JavaScript *Event Loop*



Stack Frame

Stack Frame

Stack Frame

Event Handler

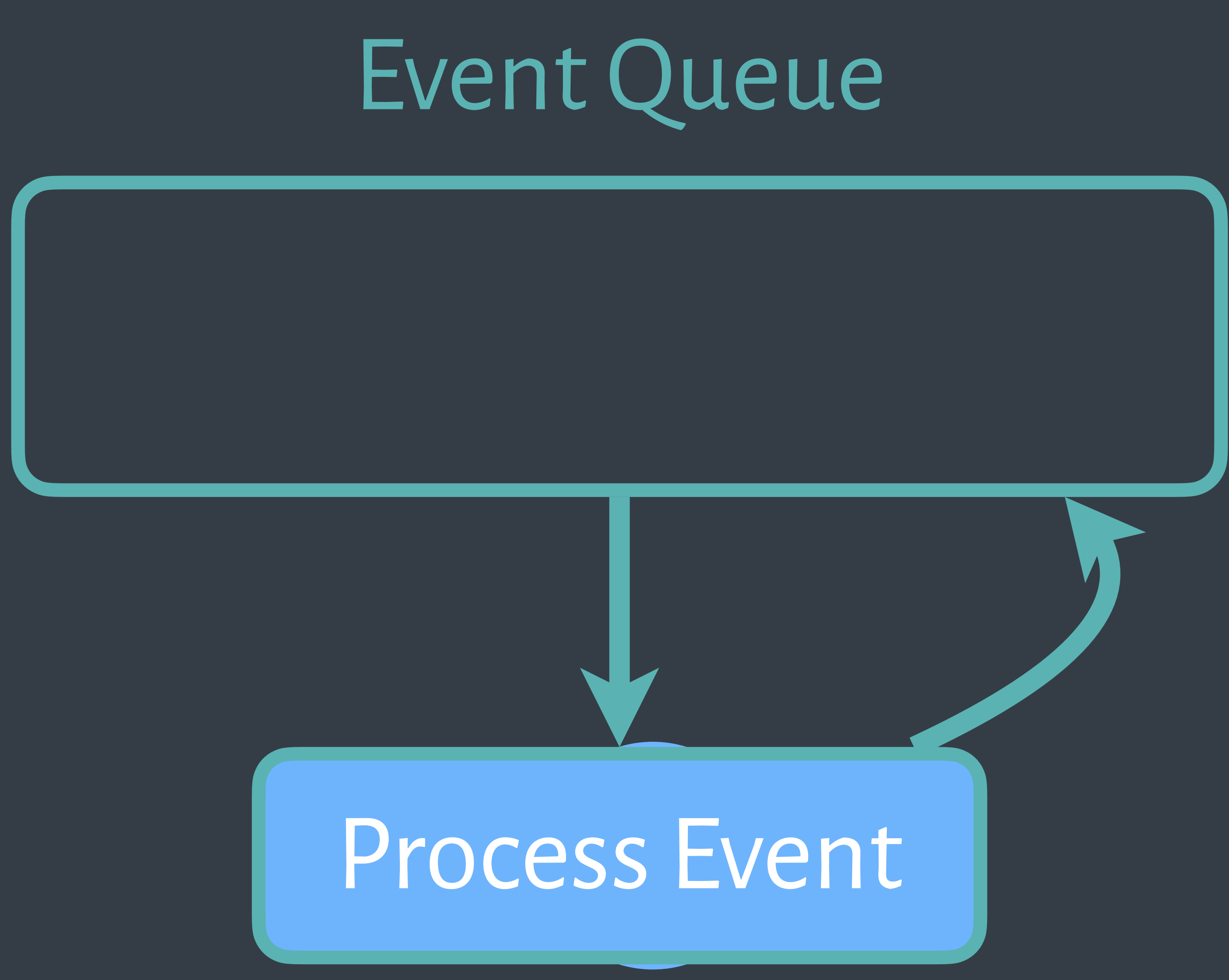
Call Stack

Event Handler

Event Handler

Event Handler

The JavaScript *Event Loop*



Stack Frame

Stack Frame

Stack Frame

Event Handler

Call Stack

Event Handler

Event Handler

Event Handler

Execution in Node.js

node.js

file system

```
fs.readFile =
  function(path, cb) {
    ...
  }
```

Execution in Node.js

web server
(server.js)

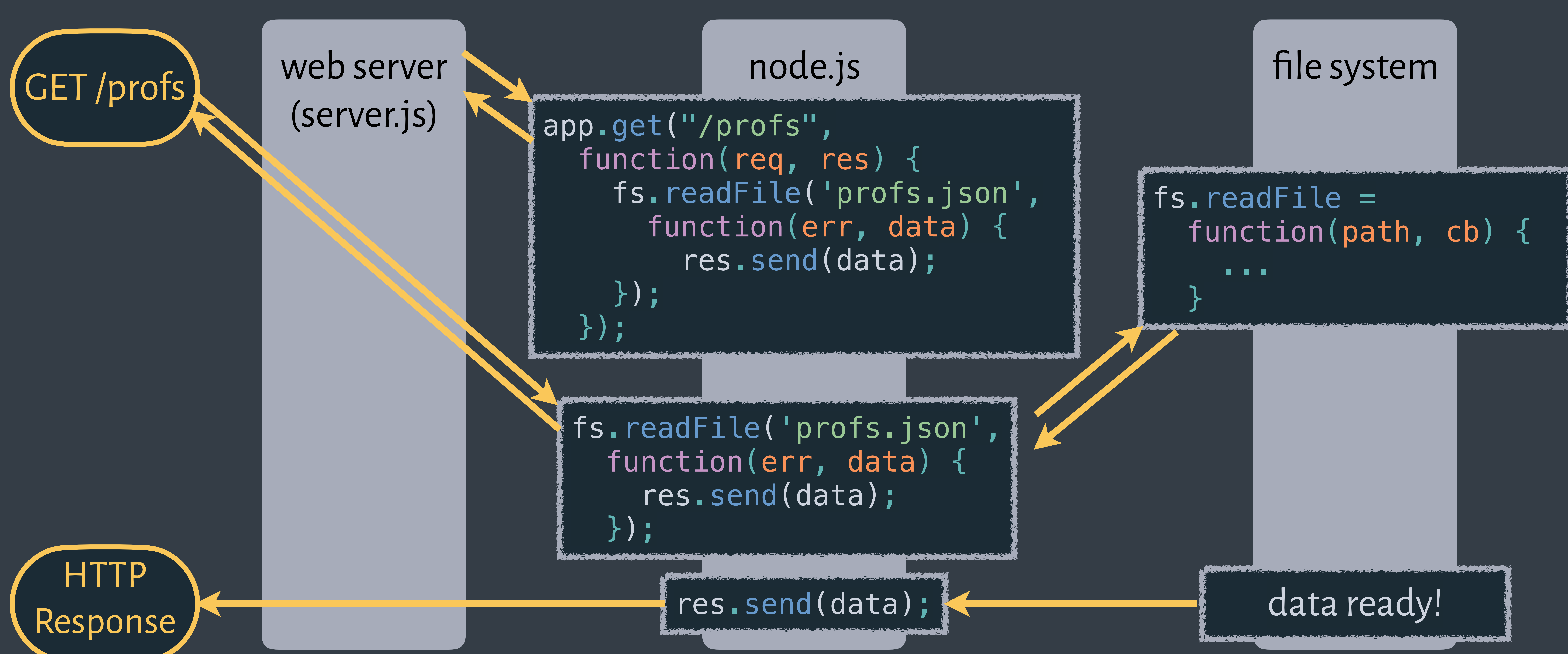
node.js

file system

```
app.get("/profs",
  function(req, res) {
    fs.readFile('profs.json',
      function(err, data) {
        res.send(data);
    });
  });
```

```
fs.readFile =
  function(path, cb) {
    ...
  }
```

Execution in Node.js



Dealing with Asynchrony: Callbacks

```
function setTimeout(callback, delay)
```

This function is called a *callback*.
`setTimeout` returns immediately. But when the timer event fires, execution is passed to this function.

```
setTimeout(function() {
  ...
}, 1000);
```

Dealing with Asynchrony: Callbacks

This function is called a *callback*.
`get` returns immediately. But when the browser receives the server's response, execution is passed to this function.

```
setTimeout(function() {

  ...
}, 1000);
```

Dealing with Asynchrony: Callbacks

This function is called a *callback*.
`get` returns immediately. But when the browser receives the server's response, execution is passed to this function.

```
setTimeout(function() {

  // can't return a value from this function!
}, 1000);
```

The callback is called at some arbitrary point in the future, outside the normal call stack. So, we cannot capture its result using a normal return statement.

Dealing with Asynchrony: Callbacks

Instead, we have to introduce additional *external state*.

```
let i = 0;

setTimeout(function() {
  i++;
}, 1000);
```

And set this state within the callback. This causes a *side effect*: the callback is no-longer self-contained.

As code grows more complex, side effects become increasingly difficult to debug.

Dealing with Asynchrony: Callbacks

```
setTimeout(function() {



}, 1000);
```

Dealing with Asynchrony: Callbacks

```
setTimeout(function() {
 fs.readDir("/assets", function(err, files) {





  });
}, 1000);
```

Dealing with Asynchrony: Callbacks

```
setTimeout(function() {
 fs.readDir("/assets", function(err, files) {
    files.forEach(function(file) {



    });
  });
}, 1000);
```

Dealing with Asynchrony: Callbacks

```
setTimeout(function() {
 fs.readDir("/assets", function(err, files) {
    files.forEach(function(file) {
      fs.readFile(file, function(err, data) {
        ...
      });
    });
  });
}, 1000);
```

Dealing with Asynchrony: *Callback Hell*

```
setTimeout(function() {
 fs.readDir("/assets", function(err, files) {
    files.forEach(function(file) {
      fs.readFile(file, function(err, data) {
        ...
      });
    });
  });
}, 1000);
```

The *Pyramid of Doom*
aka *Callback Hell*

Dealing with Asynchrony

Have to read code carefully to understand what order functions get called.

```
setTimeout(readFiles, 1000);

function readFiles() {
  fs.readDir("/assets", loopFiles)
}

function loopFiles(err, files) {
  files.forEach(function(file) {
    fs.readFile(file, renderFile);
  });
}

function renderFile(err, data) {
  ...
}
```

Dealing with Asynchrony: *Promises*

A Promise is a *proxy* object for a value that will be determined some time in the future.

It is returned *synchronously* from an asynchronous function.

Dealing with Asynchrony: *Promises*

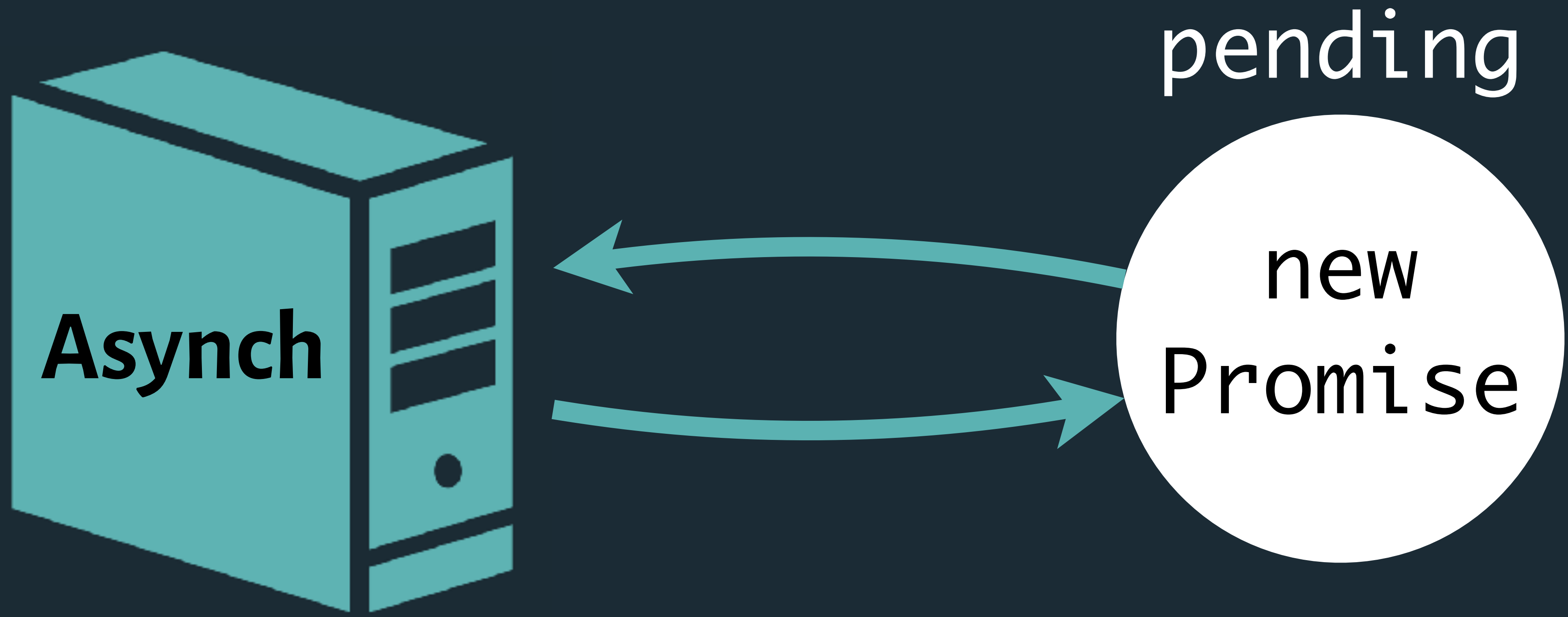
A Promise is a *proxy* object for a value that will be determined some time in the future.

It is returned *synchronously* from an asynchronous function.

Dealing with Asynchrony: *Promises*

A Promise is a *proxy* object for a value that will be determined some time in the future.

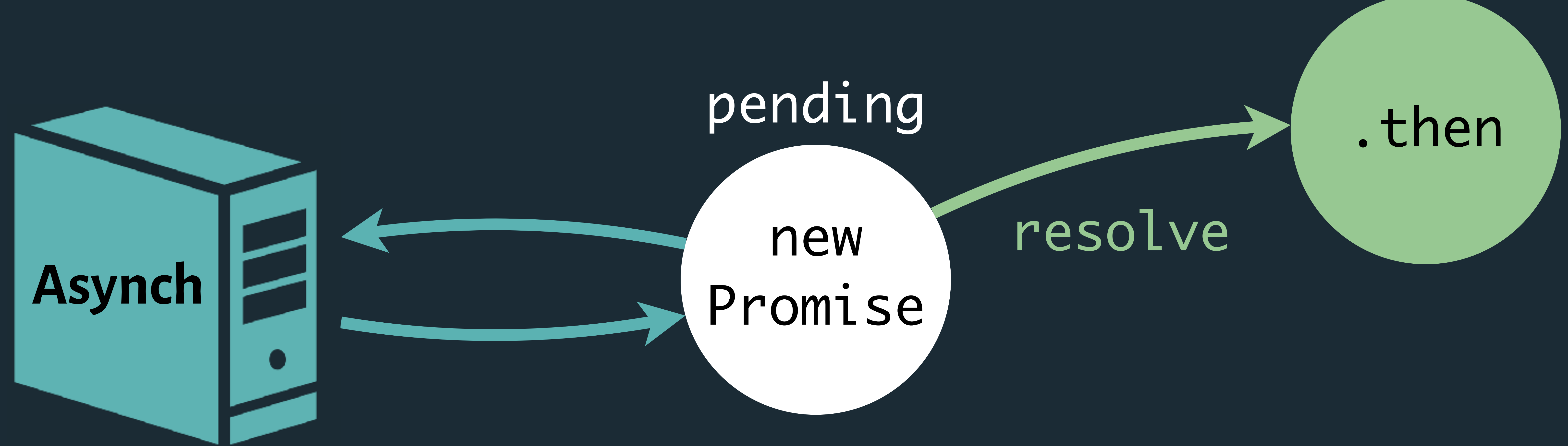
It is returned *synchronously* from an asynchronous function.



Dealing with Asynchrony: *Promises*

A Promise is a *proxy* object for a value that will be determined some time in the future.

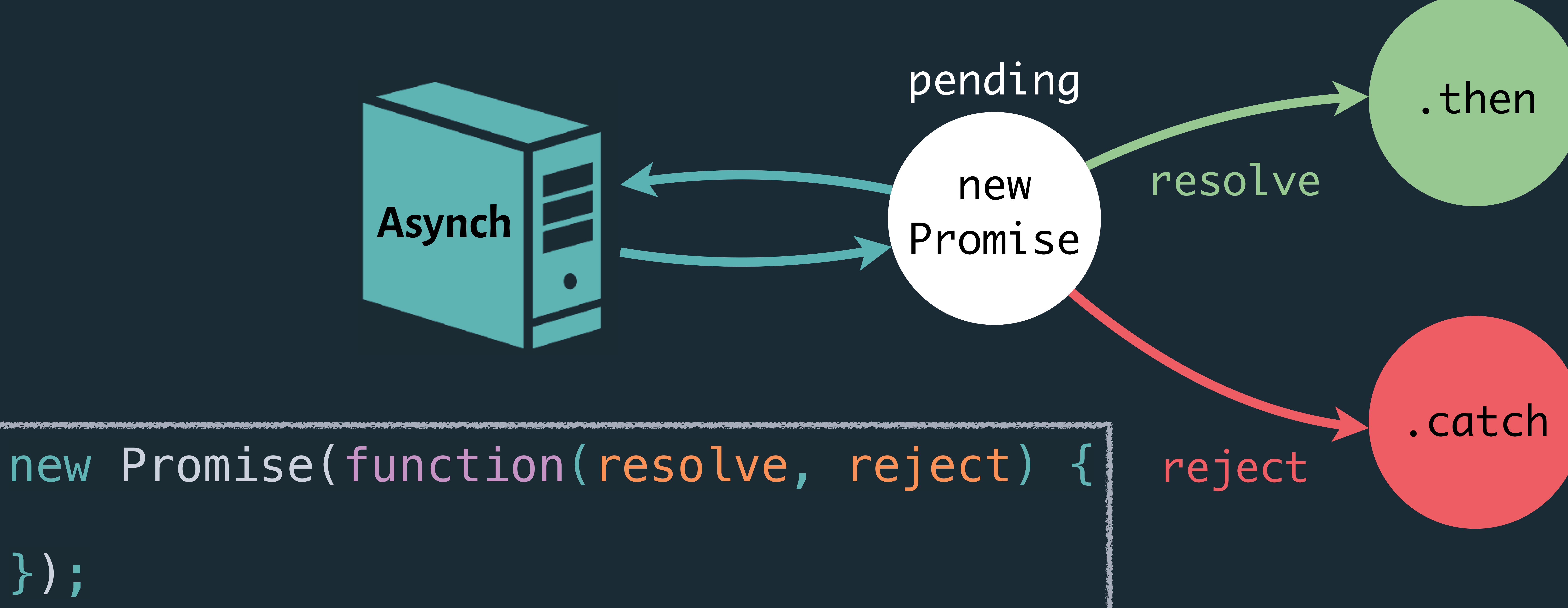
It is returned *synchronously* from an asynchronous function.



Dealing with Asynchrony: *Promises*

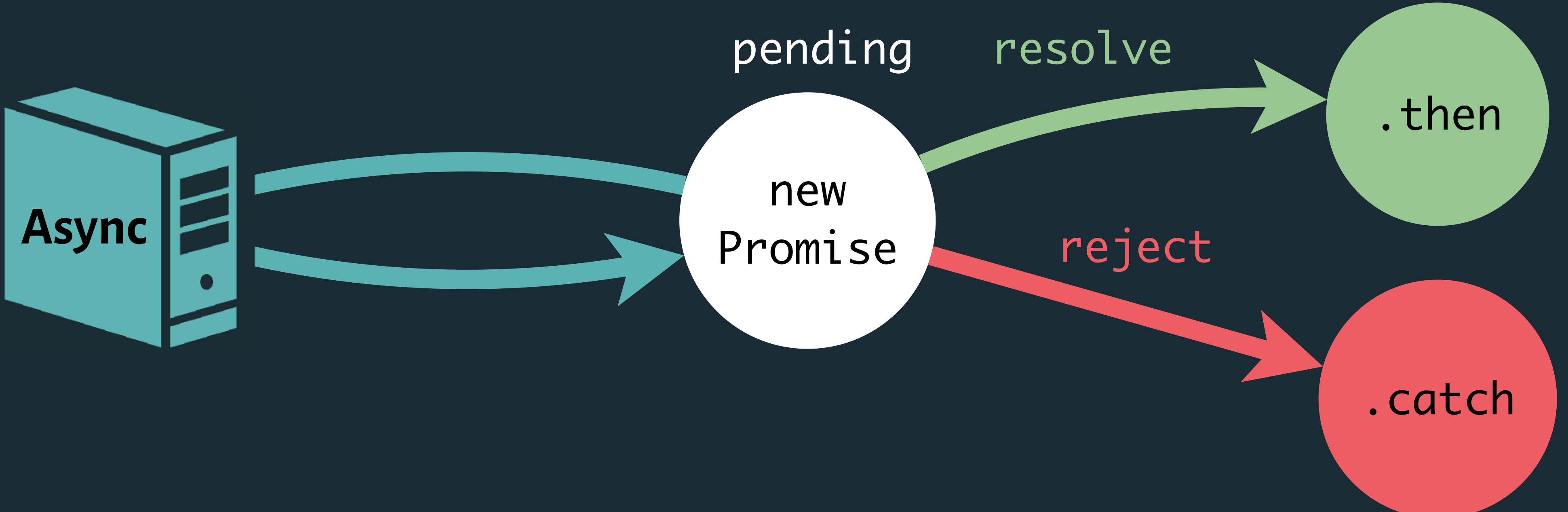
A Promise is a *proxy* object for a value that will be determined some time in the future.

It is returned *synchronously* from an asynchronous function.



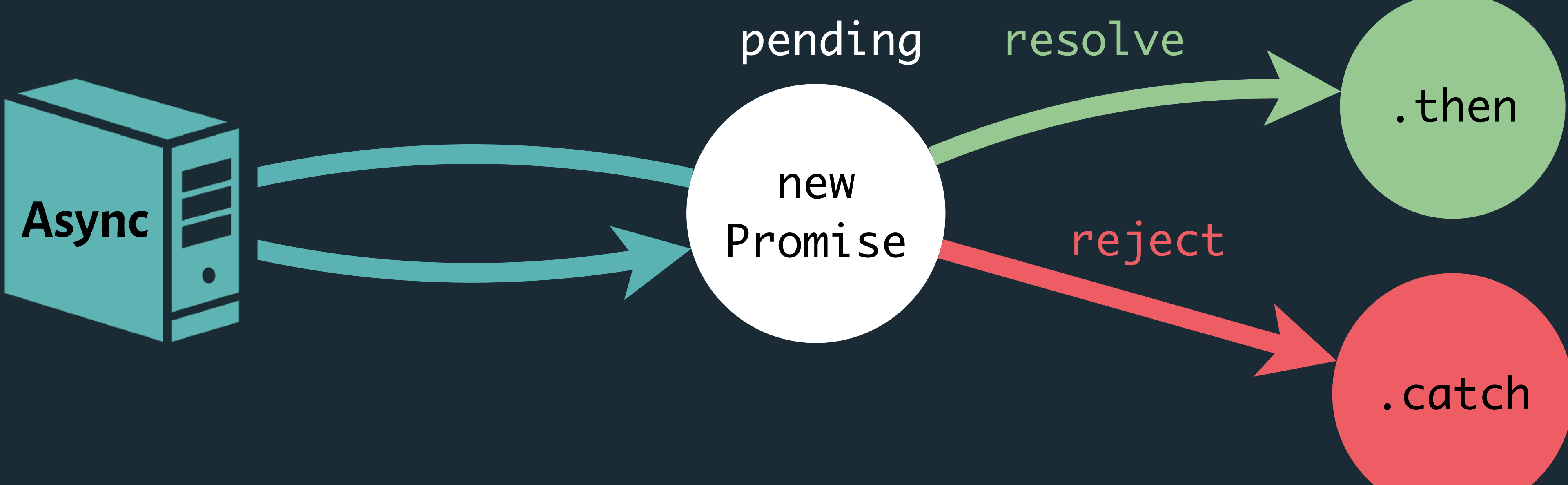

```
new Promise(function(resolve, reject) {

});
```




```
function setTimeout(callback, delay)
```

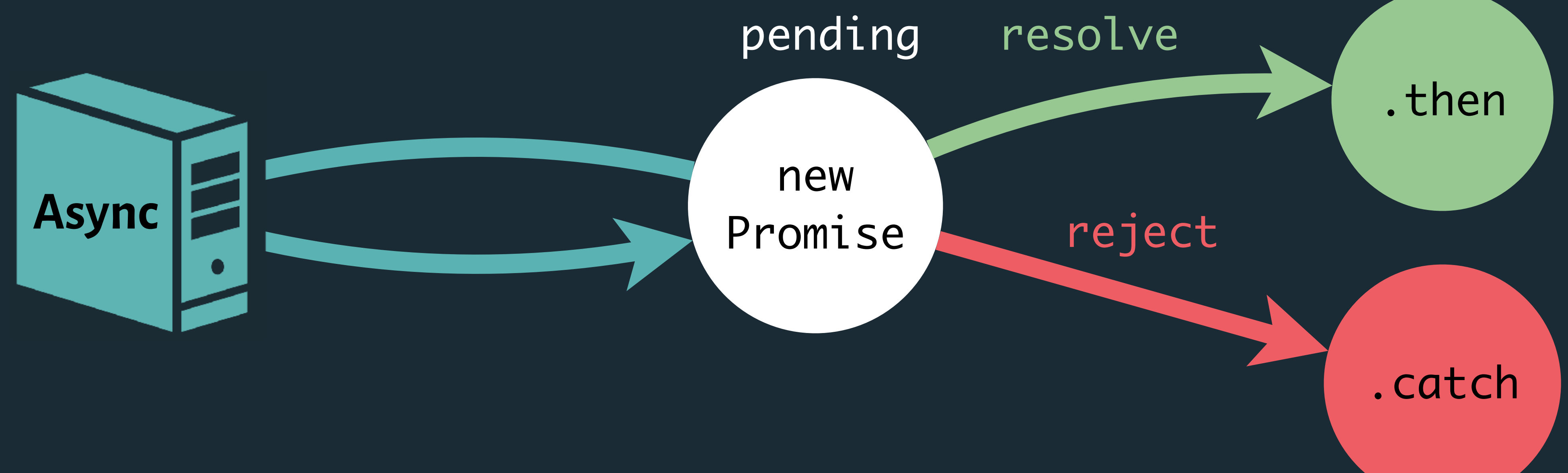
```
function later(delay) {
  return new Promise(function(resolve, reject) {
    setTimeout(resolve, delay);
  });
}
```




```
function setTimeout(callback, dela
```

```
function later(delay) {
  return new Promise(function(reso
, reject) {
    setTimeout(resolve, delay);
  });
}
```

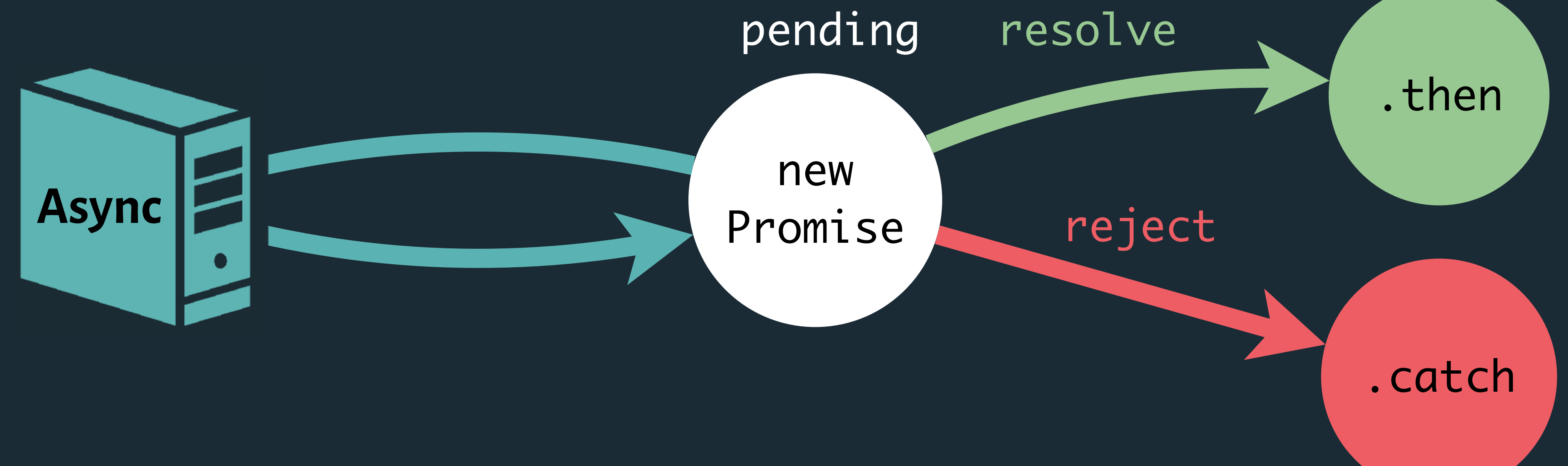
```
later(1000)
```




```
function setTimeout(callback, dela
```

```
function later(delay) {
  return new Promise(function(reso
, reject) {
    setTimeout(resolve, delay);
  });
}
```

```
later(1000)
  .then(function() {
    return fsPromises.readDir("/assets");
  })
```




```
function setTimeout(callback, dela
```

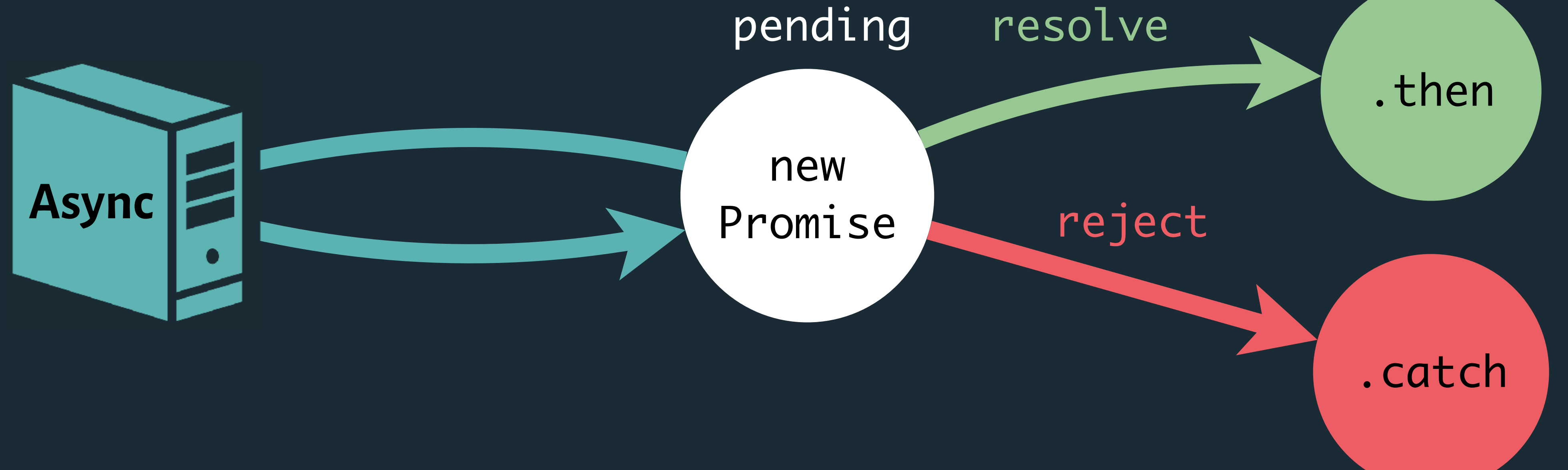
```
function later(delay) {
  return new Promise(function(reso
, reject) {
    setTimeout(resolve, delay);
  });
}
```

```
later(1000)
  .then(function() {
    return fsPromises.readDir("/assets");
  })
  .then(function(files) {



  })
```

Chain: execute async operations one after another by returning a promise within a then function.



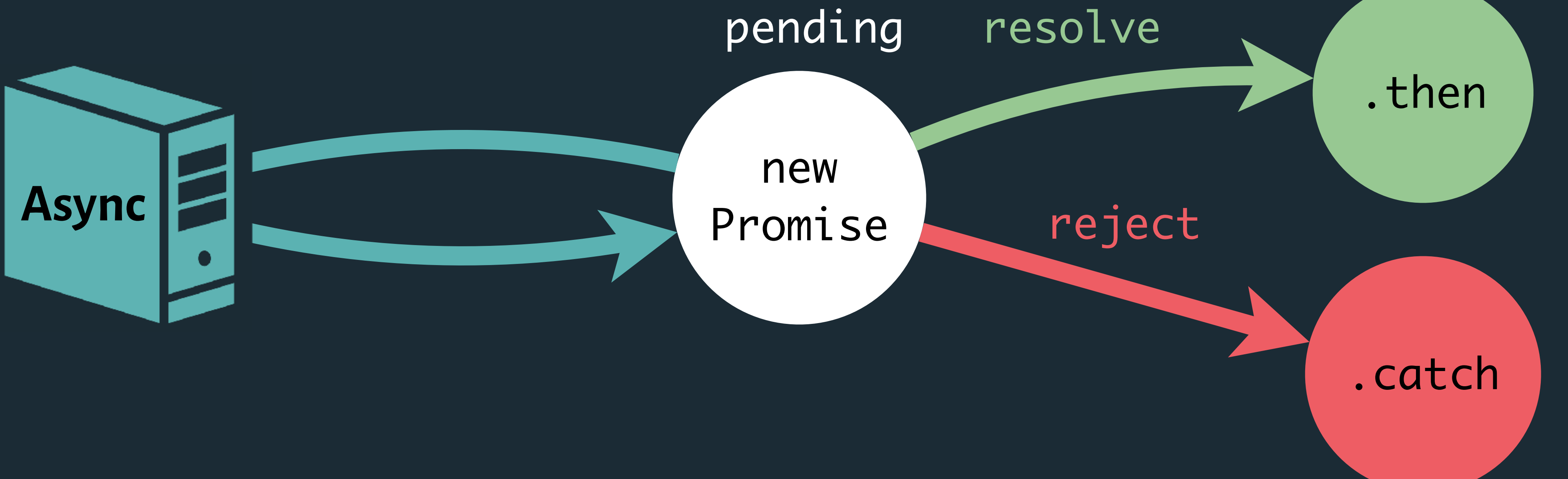

```
function setTimeout(callback, dela
```

```
function later(delay) {
  return new Promise(function(reso
, reject) {
    setTimeout(resolve, delay);
  });
}
```

```
later(1000)
  .then(function() {
    return fsPromises.readDir("/assets");
  })
  .then(function(files) {
    return Promise.all(files.map(function(file) {
      return fsPromises.readFile(file)
    }));
  })
```

Chain: execute async operations one after another by returning a promise within a then function.

`Promise.all` returns a *single* Promise that resolves when all passed promises have resolved.



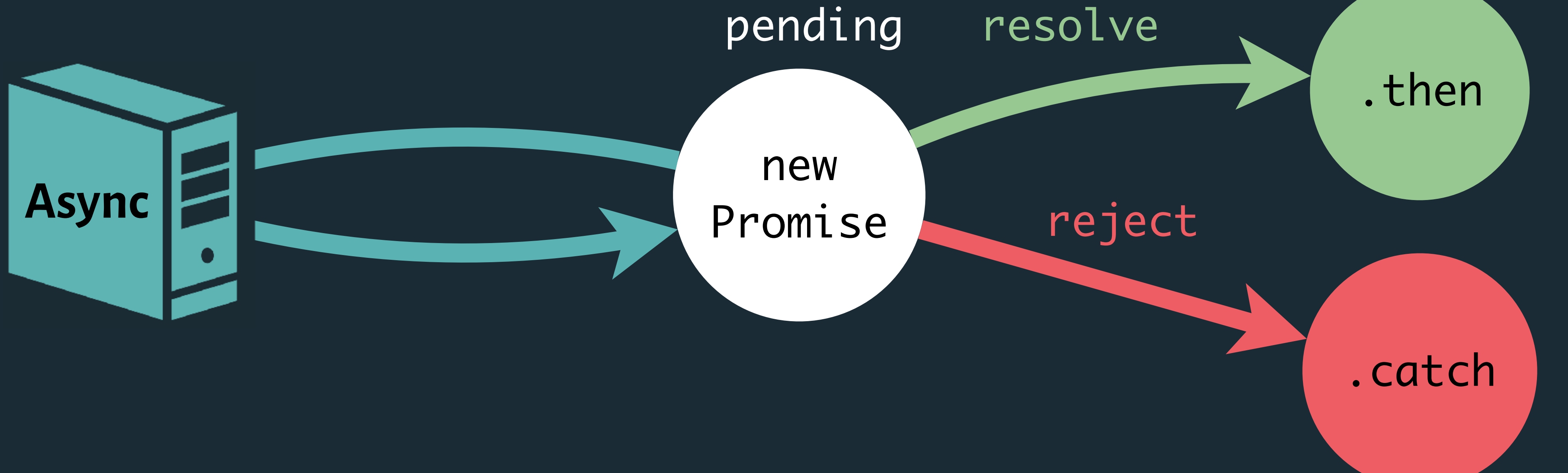

```
function setTimeout(callback, dela
```

```
function later(delay) {
  return new Promise(function(reso
, reject) {
    setTimeout(resolve, delay);
  });
}
```

```
later(1000)
  .then(function() {
    return fsPromises.readDir("/assets");
  })
  .then(function(files) {
    return Promise.all(files.map(function(file) {
      return fsPromises.readFile(file)
    }));
  })
  .then(function(data) {
    ...
  })
```

Chain: execute async operations one after another by returning a promise within a then function.

`Promise.all` returns a *single* Promise that resolves when all passed promises have resolved.




```
function setTimeout(callback, dela
```

```
function later(delay) {
  return new Promise(function(reso
, reject) {
    setTimeout(resolve, delay);
  });
}
```

```
later(1000)
  .then(function() {
    return fsPromises.readDir("/assets");
  })
  .then(function(files) {
    return Promise.all(files.map(function(file) {
      return fsPromises.readFile(file)
    }));
  })
  .then(function(data) {
    ...
  })
  .catch(console.error);
```

Chain: execute async operations one after another by returning a promise within a `then` function.

`Promise.all` returns a *single* Promise that resolves when all passed promises have resolved.

If an error occurs at any step, the chain is interrupted and we can catch and handle the error.

Give us Feedback

https://tinyurl.com/6104-feedback