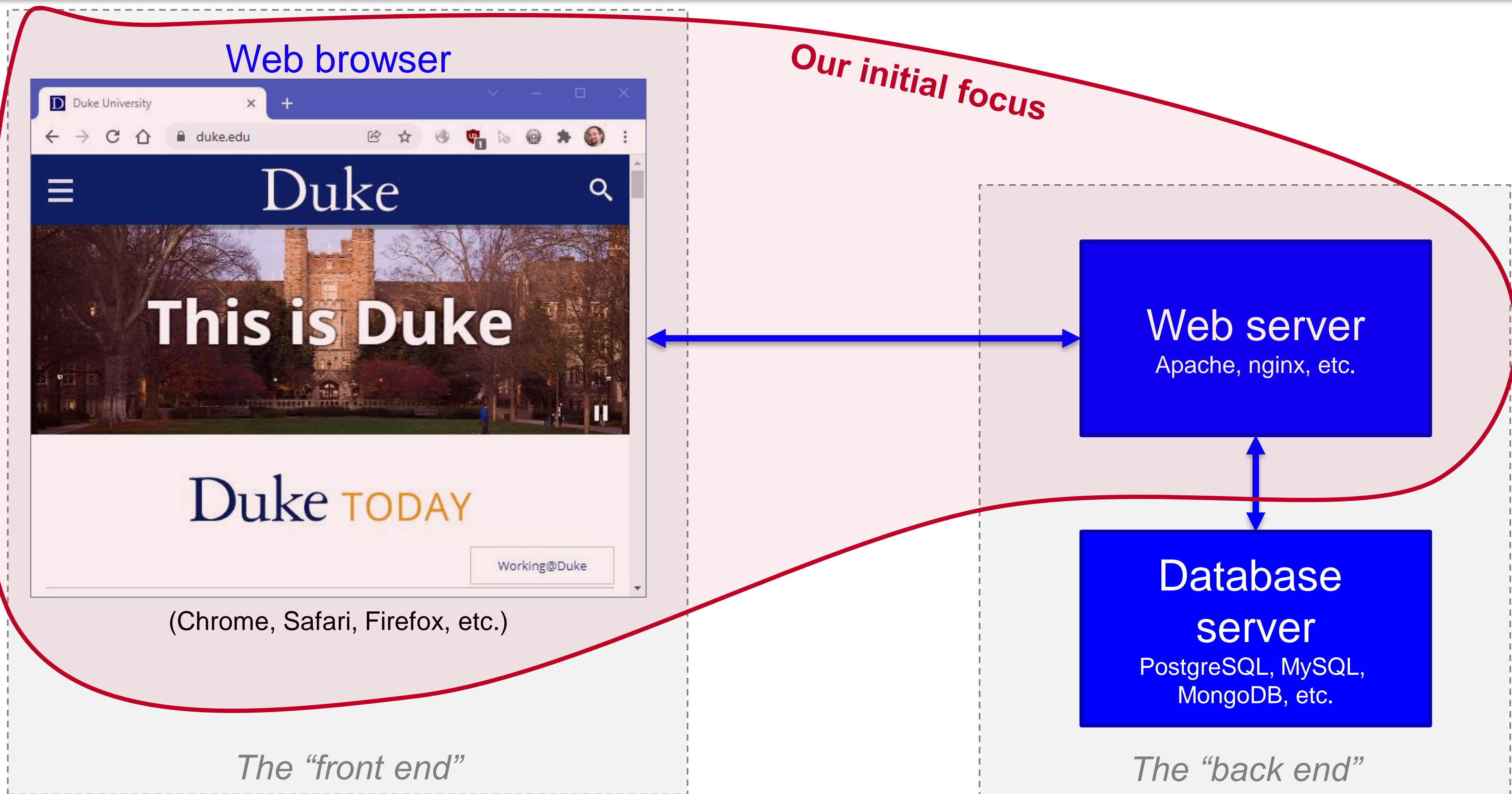
ECE 458
Engineering Software for Maintainability

Introduction to Web Development

Tyler Bletsch

Client server topology



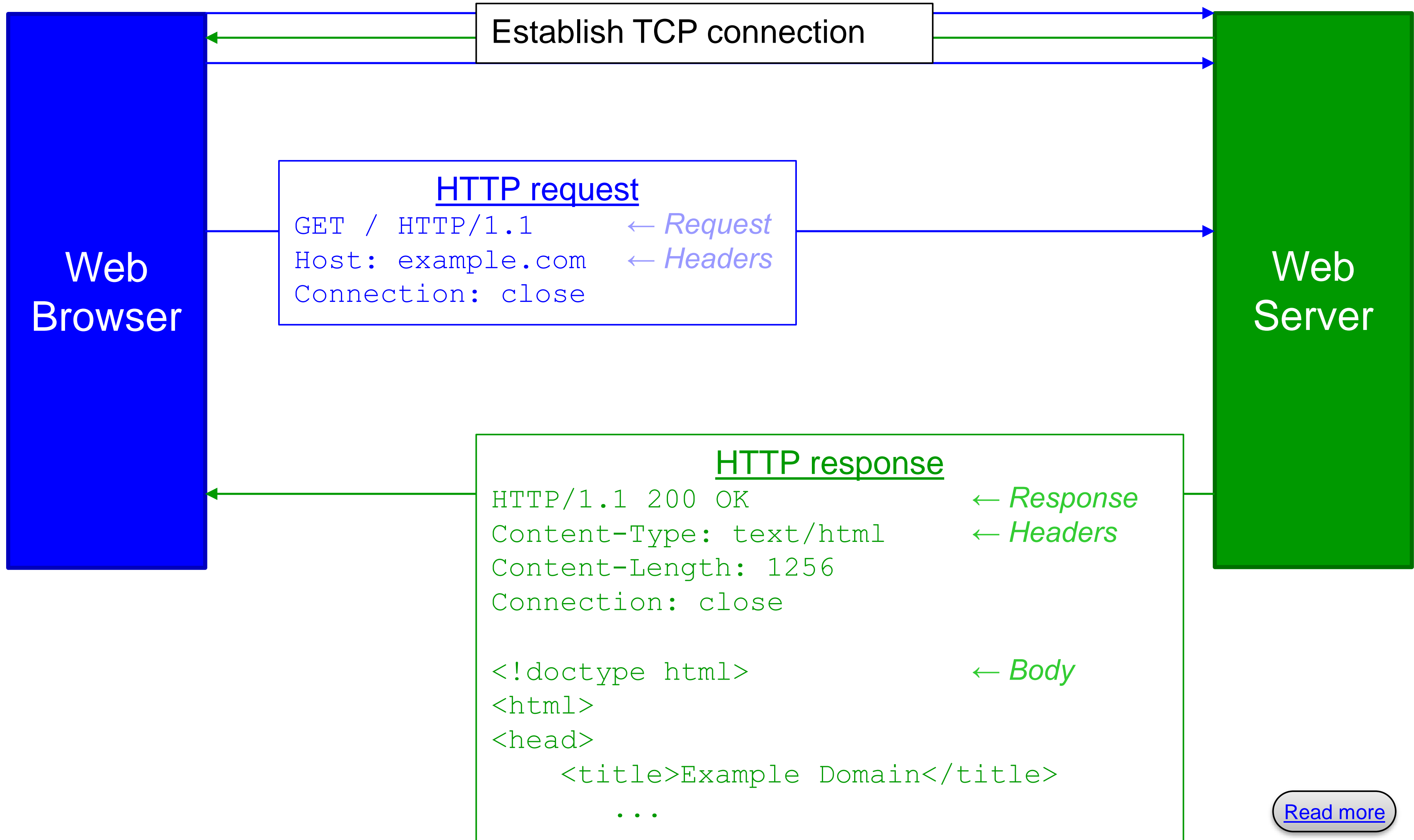
- Can get much fancier (middleware, clustering, proxies, etc.)

Part 1:
Nuts and bolts

A level of abstraction below where you'll be working,
but you need to know what's going on

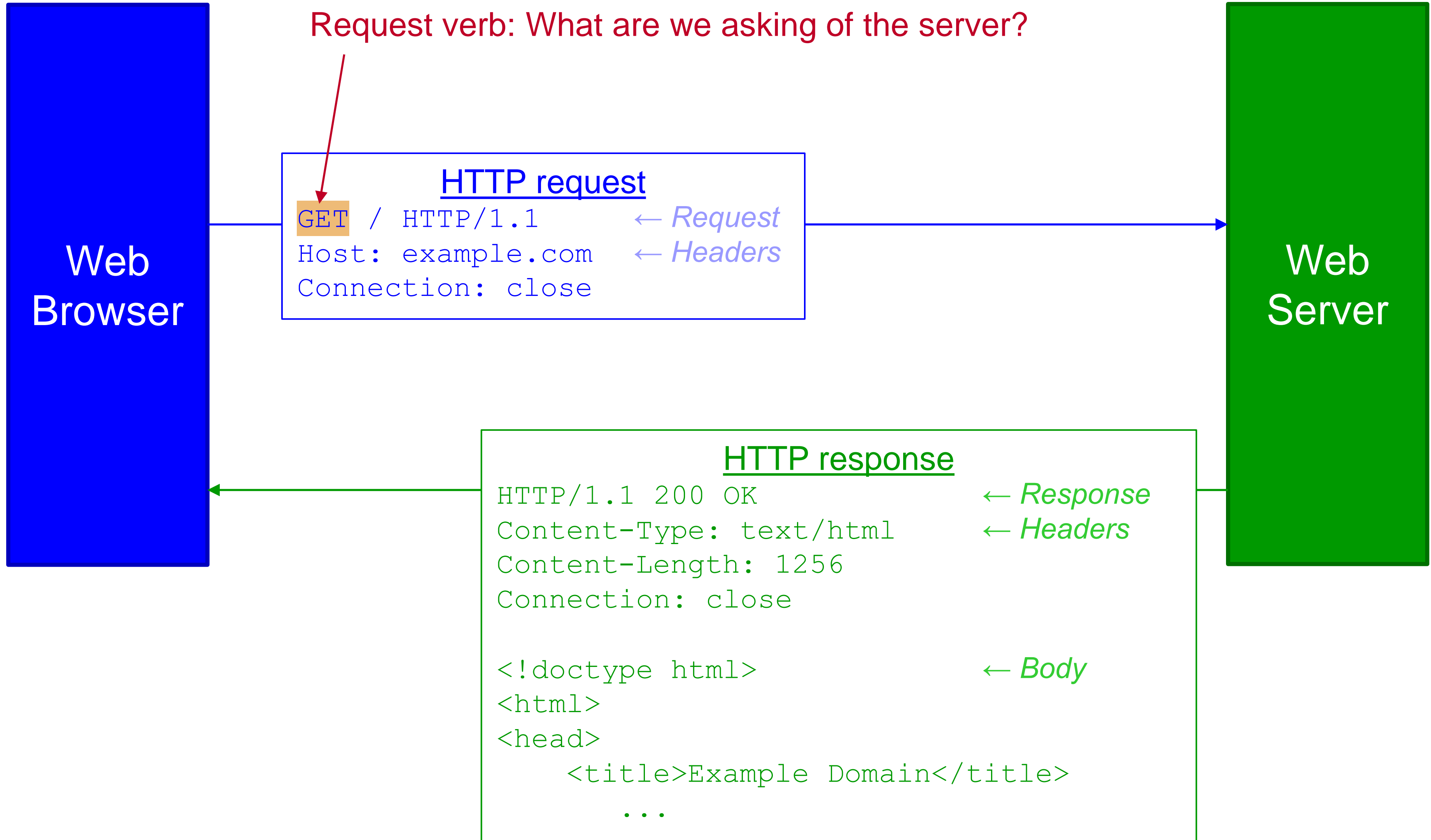
HTTP: Web request/response

(requesting http://example.com/)



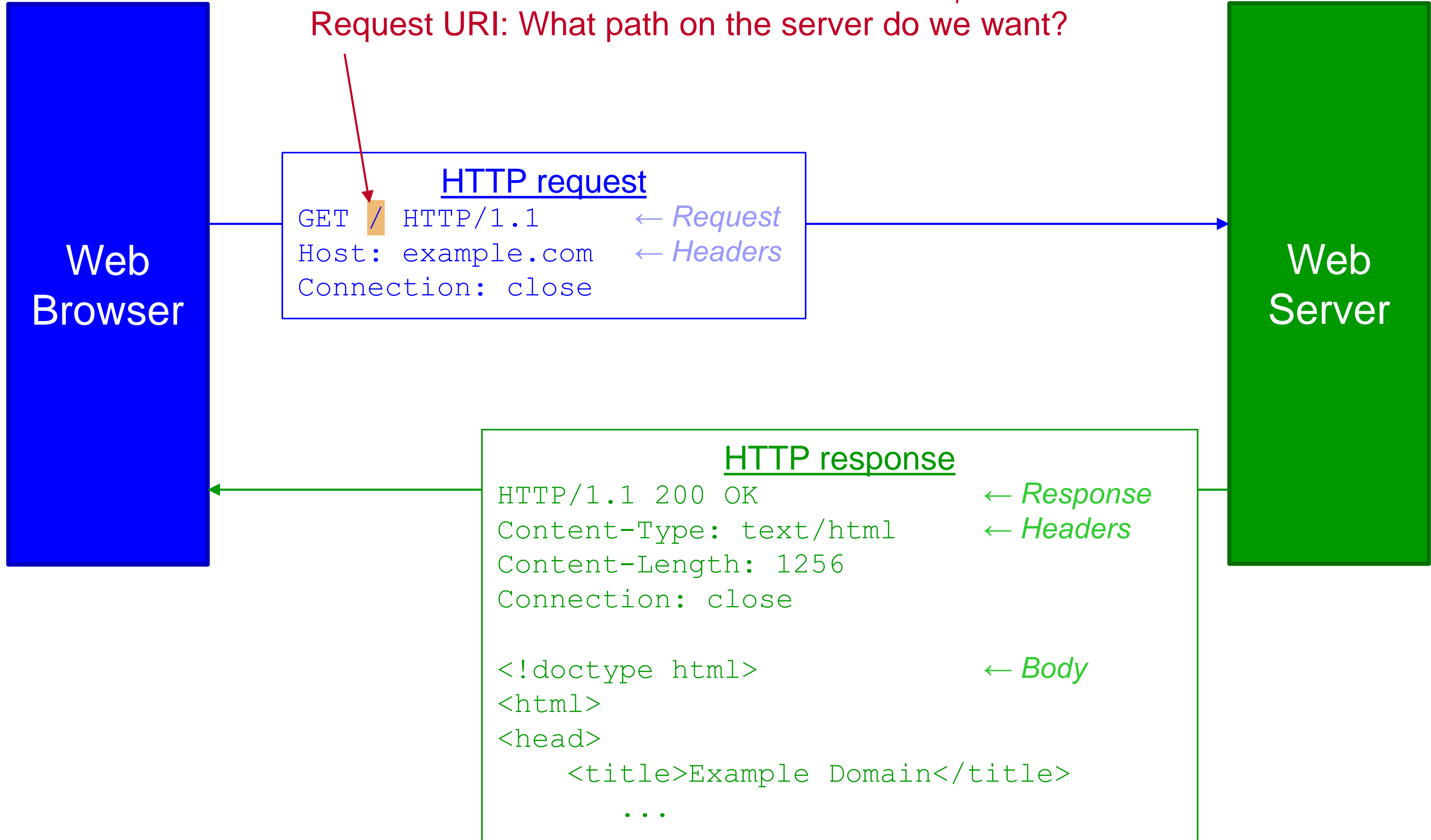
HTTP: Web request/response

(requesting http://example.com/)



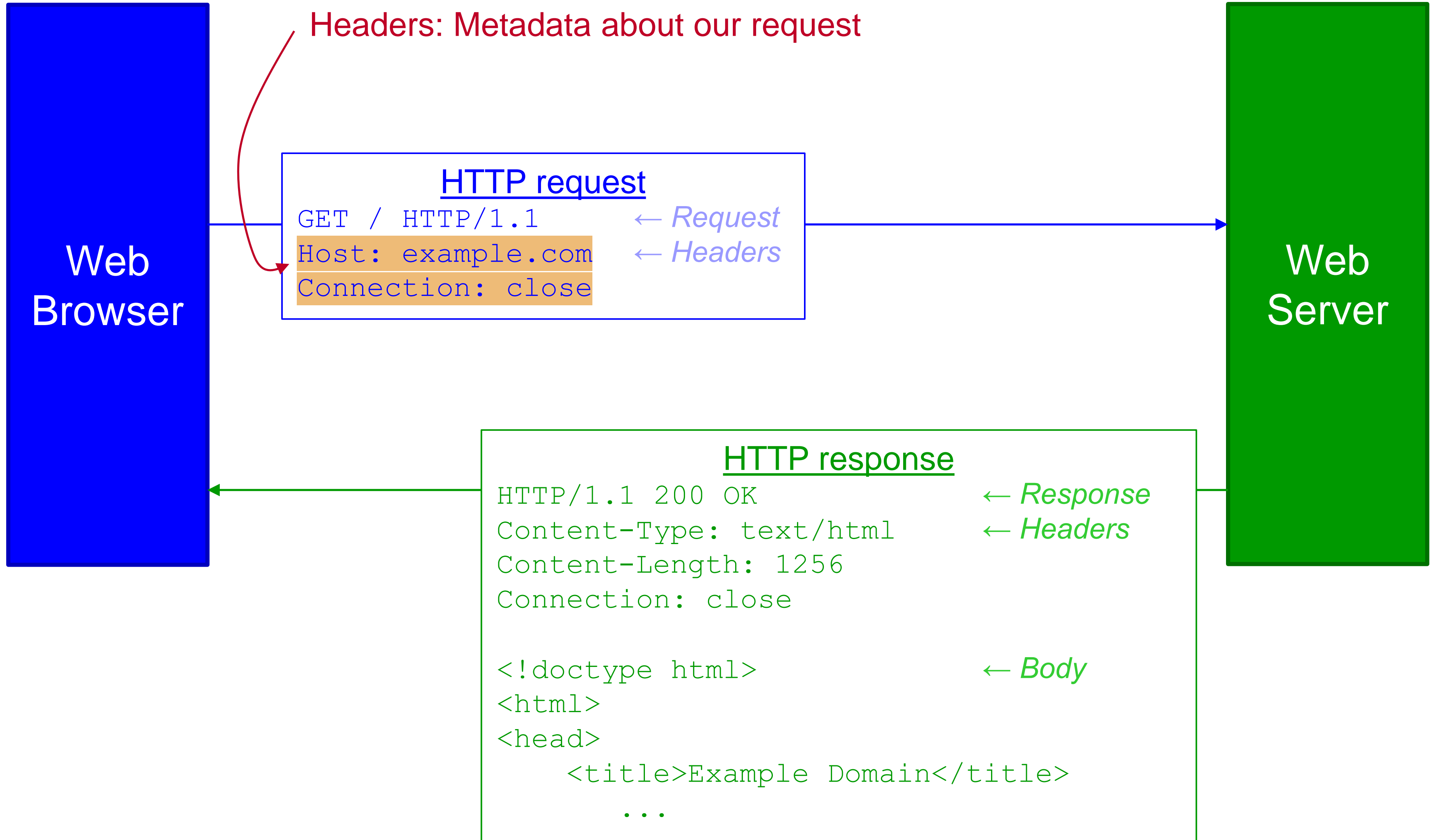
HTTP: Web request/response

(requesting http://example.com/)



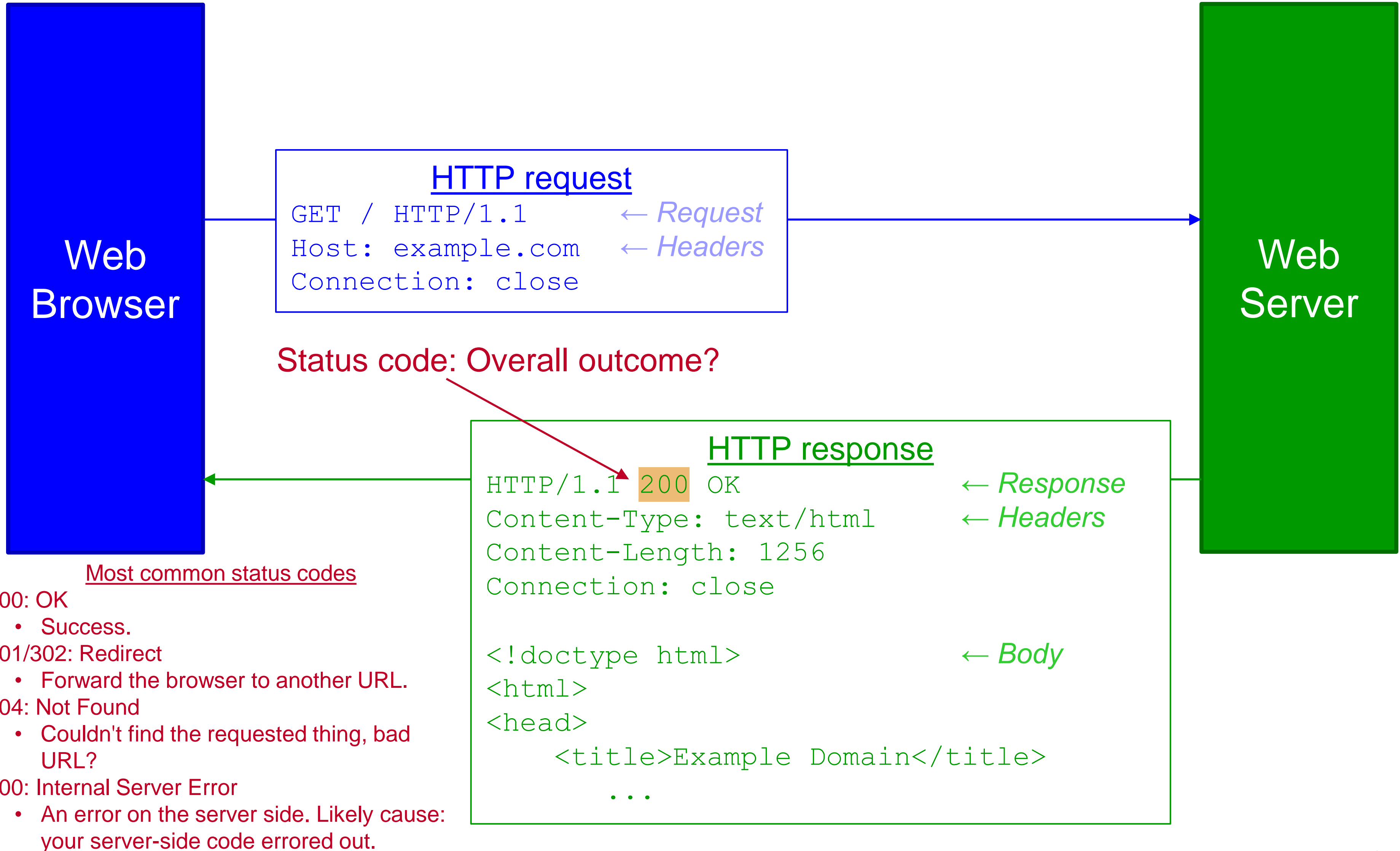
HTTP: Web request/response

(requesting http://example.com/)



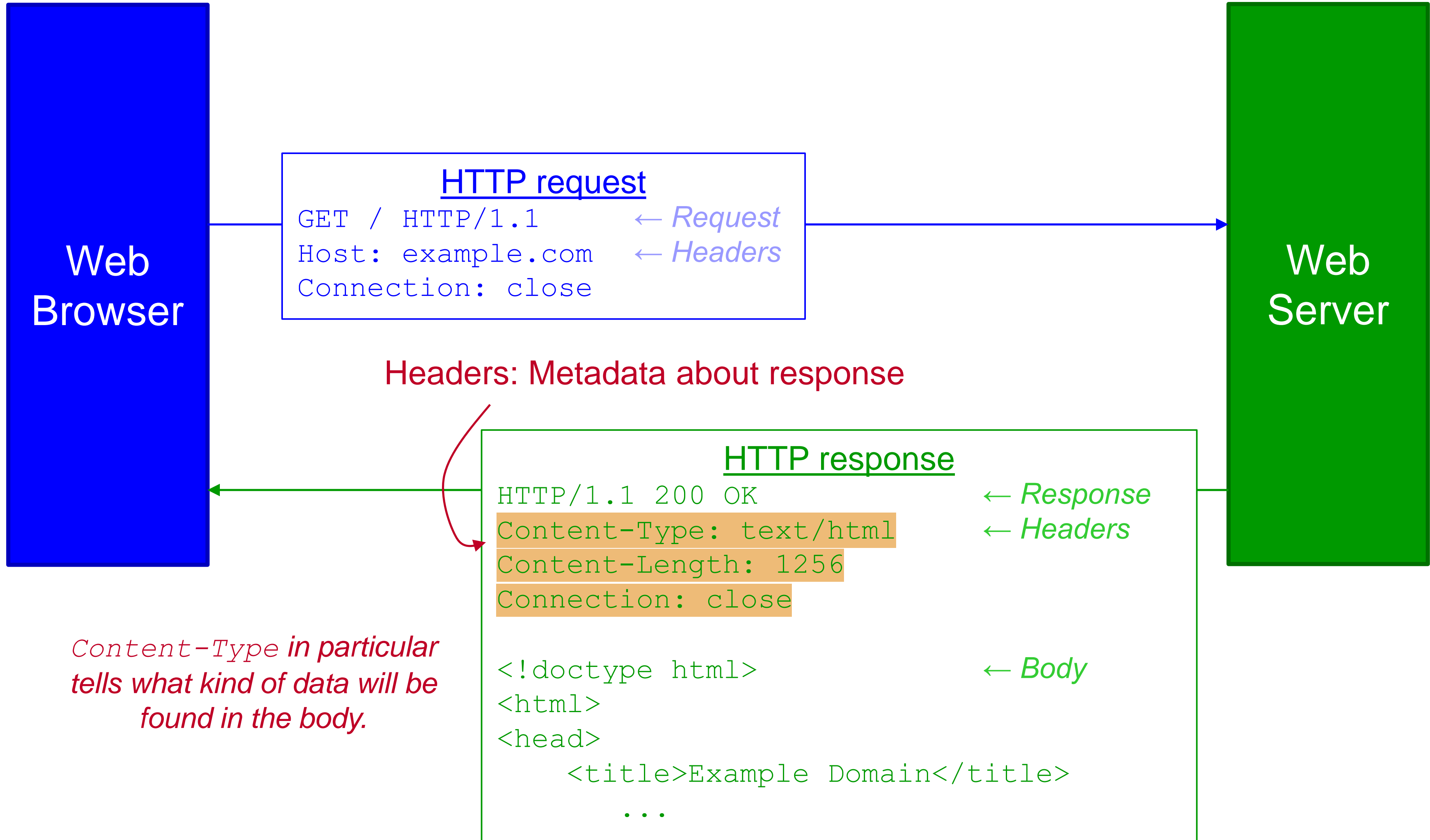
HTTP: Web request/response

(requesting http://example.com/)



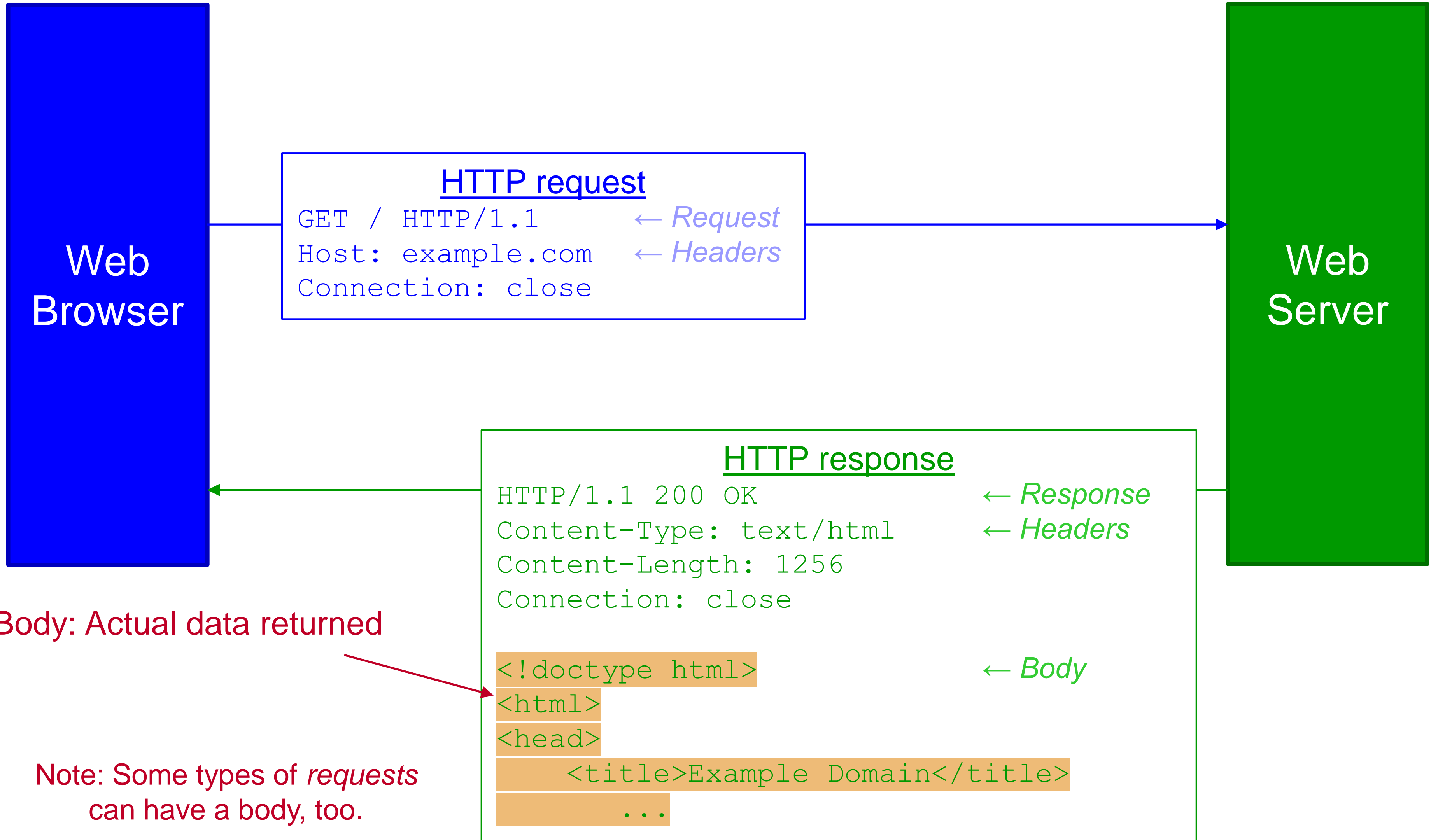
HTTP: Web request/response

(requesting http://example.com/)



HTTP: Web request/response

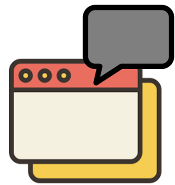
(requesting http://example.com/)



Two main purposes of a request

- The browser can load site-authored code written in **Javascript**; this code can itself make web requests.

- This gives rise to two categories of request:
  - The **browser** directly requesting content



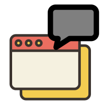
    - Done as part of document request process (when you click link)
    - Gets you content that's consumed by the browser.
    - Data types: HTML, CSS, JS code, images, etc.
    - Example: "Show me the HTML and any content I need to display it"
  - **Javascript on the site** requesting content (often API calls)

    - Could be requesting anything, but usually requesting Javascript-parsable content
    - Data types: JSON data objects, XML data objects
    - Example: "Look up the student list", "Create a student object", etc.
    - Often this is done as part of a "**RESTful API**" (covered later)

Request verbs and their semantics

Verb	Purpose	Note
GET	Retrieve data	Must not alter server state; read-only.
POST	Add/modify data	Request includes a body.
PUT	Add new data object	Request includes a body.
DELETE	Remove a data object	

There are a few more verbs than this, but they're usually handled automatically by your framework or aren't in common use.



Examples of browser-based requests

`GET /about.html`

- Request the **`about.html`** document, which may refer to the image "/logo.png", the stylesheet "/styles/main.css", and the javscript "/js/site.js". The browser would then **`GET`** each of those in turn.

`POST /contactform`

- Send form-based content to the given URL, which would process it and do something. Returns an HTML document.

Note: Browsers do not typically issue PUT and DELETE directly.

Examples of javascript-based requests

`GET /students`

- Request a JSON list of students, like [{"name": "Jimmy", "age": 14}, …]

`PUT /students`

- Body is JSON of a student, adds that student to the server's database

`POST /students/251`

- Body is JSON of an updated version of student number 251; server commits these changes

`DELETE /students/251`

- Student 251 is deleted from the server's database

Read more in general.
Read more with respect to REST.

Types of content

- Remember the `Content-Type` header? Common types:
  - **Hypertext Markup Language (HTML)**: Describes structure and content of a web document, marked up with <tags>.
  - **Cascading Stylesheets (CSS)**: Describes *how* the HTML content should be shown (color, spacing, etc.).
  - **Javascript (JS)**: Code to be run in the web browser.
  - **Images (PNG, JPEG, GIF, etc.)**: Pictures
  - **Javascript Object Notation (JSON)**: A text-based record format
    - Has plain numbers, strings in "quotes", lists in brackets [1,2,3], and dictionaries in braces {"key": "value", ...}
    - Lists/dictionaries can nest, so you can represent whole data structures.
  - **Extensible Markup Langauge (XML)**: An older text-based record format. Uses <tags> like HTML, but customizable. Thankfully dying.

Examples of types of content

```
<!DOCTYPE html>
<html>

<head>
  <title>My First Webpage</title>
  <meta name="viewport" content="width=devic
  <link rel="stylesheet" type="text/css" hre
</head>

<body>

  <div class="container">

   <h1>Heading 1</h1>
```

Read more

HTML

```
.screen-reader-text:hover,
.screen-reader-text:active,
.screen-reader-text:focus {
    background-color: #f1f1f1;
    border-radius: 3px;
    box-shadow: 0 0 2px 2px rgba(0, 0, 0, 0.6);
    clip: auto !important;
    color: #21759b;
    display: block;
    font-size: 14px;
    font-size: 0.875rem;
    font-weight: bold;
    height: auto;
    left: 5px;
    line-height: normal;
    padding: 15px 23px 14px;
    text-decoration: none;
    top: 5px;
    width: auto;
    z-index: 100000; /* Above WP toolbar
}
```

Read more

CSS

```
let meetups = [
  {name:'JavaScript', isActive:true, members:700},
  {name:'Angular', isActive:true, members:900},
  {name:'Node', isActive:false, members:600},
  {name:'React', isActive:true, members:500}
];
let sumFPChain = meetups.filter((m)=>{
    return m.isActive;
  })
  .map((m)=>{
    return m.members- (0.1*m.members);
  })
  .reduce((acc, m)=>{
    return acc + m;
  },0);
console.log(sumFPChain);   // Output will be 1890
```

Read more

JS

```
"name": "myapplication",
"description": "some description here",
"version": "0.0.1",
"private": true,
"scripts": {
  "start": "node ./bin/www"
},
"dependencies": {
  "express": "~4.12.2",
  "jade": "~1.9.2"
}
```

Read more

JSON

```
<EmployeeData>
 - <employee id="34594">
     <firstName>Heather</firstName>
     <lastName>Banks</lastName>
     <hireDate>1/19/1998</hireDate>
     <deptCode>BB001</deptCode>
     <salary>72000</salary>
   </employee>
 - <employee id="34593">
     <firstName>Tina</firstName>
     <lastName>Young</lastName>
     <hireDate>4/1/2010</hireDate>
     <deptCode>BB001</deptCode>
     <salary>65000</salary>
   </employee>
```

XML

JPEG

Types of HTTP

- This shouldn't matter a ton day-to-day,
  but will come up when *deploying* your webserver

- HTTP Versions:
  - HTTP 1.1: Classic. Still most common.
  - HTTP 2: Newer standard, more efficient.

- HTTP**S**: HTTP **S**ecure
  - Applies *certificates* and *encryption* to ensure confidentiality & integrity
  - Will skip details here, but you'll need to set it up
  - Only tricky bit: you need a **certificate** for your site
    - Rolling your own server? Lets Encrypt can do this for you
    - Using a cloud service? If they can't do this for you, they're clowns

Read more

Knowing who you're talking to

- HTTP is anonymous by default: "Some rando sent a request"
- We want **authentication**: recognition of a specific user
- How to identify distinct users?
  - **Cookies**: Server can ask browser to include a bit of text in every subsequent request
    - Example: `SESSION=123`
    - Server remembers what this means (see next slide)
  - **Storage**: Javascript provides **SessionStorage** (kept as long as the tab is open) and **LocalStorage** (kept indefinitely)
    - Not automatically sent by browser to server, but can be included when Javascript makes a request
    - Can achieve authentication, also other stuff
      - When doing authentication, use **LocalStorage** (closing a tab shouldn't log you out)

Read more

What to store in cookies / storage?

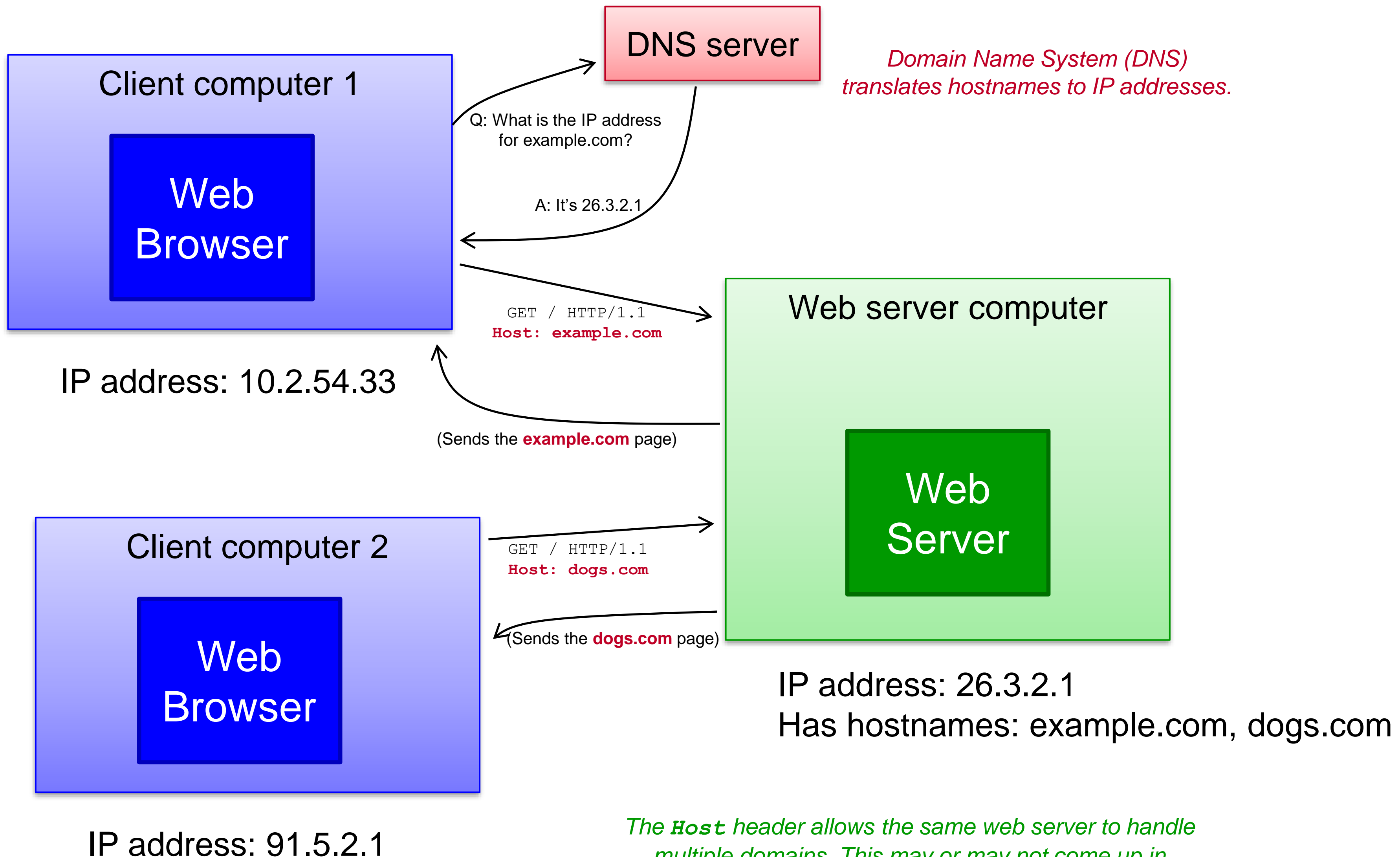
- Classic: If using cookies, you can just have some kind of **session identifier**, and have the server keep a list of those
  - Cookie: "`SESSION=123`"
  - Database: "Session 123 was created when user 'bob' logged in"
  - Conclusion: "This is bob"

Read more

- Modern: Store a **JSON Web Token (JWT)** in a cookie or LocalStorage
  - Server can provide a bit of JSON that is cryptographically signed by the server (can't be tampered with)
  - Can be used to hold login info
  - Example: `{"logged_in_user": "bob"}`
  - Server need not remember what it means, because client cannot fabricate it

Read more

Even lower level: networking details



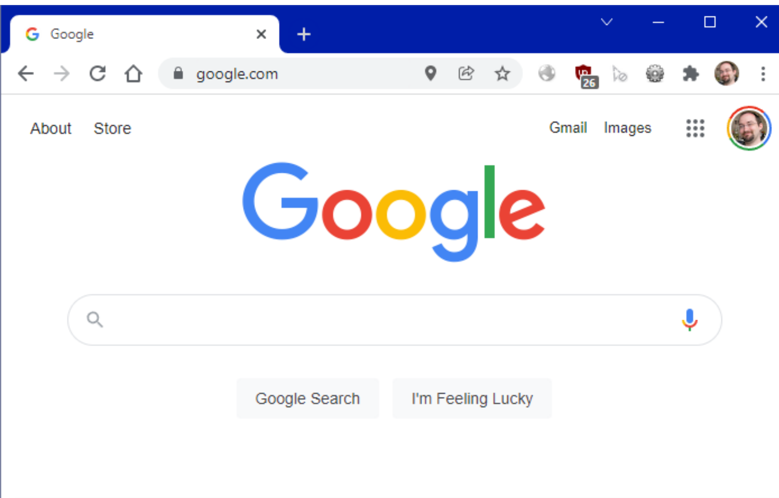
Web application architecture

Types of web applications

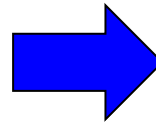
- Two main types of web applications:
  - **Content generation**:
    - Browser makes requests for HTML, server renders HTML that addresses the request
    - Google Search works this way
  - **API driven**:
    - Browser loads static HTML content and JS code
    - Javascript in browser makes data requests of the server; javascript updates the document displayed in the browser, thus showing new content
    - Google Maps works this way
    - The API in this case is often a **RESTful** API
      (it follows certain rules and design patterns for simplicity)

Content generation example: Google Search

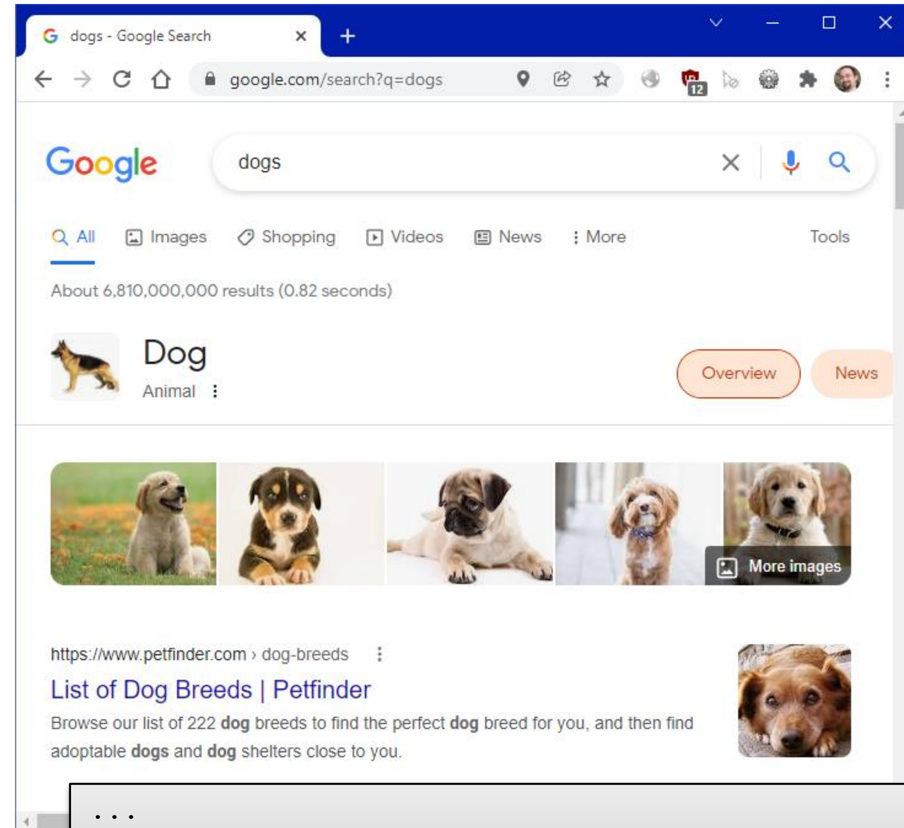
Simple HTML form




```
...
<form action="/search" method="GET">
  <input name="q" type="text">
  <input type="submit" value="Google Search">
  <input type="submit" value="I'm Feeling Lucky">
</form>
...
```



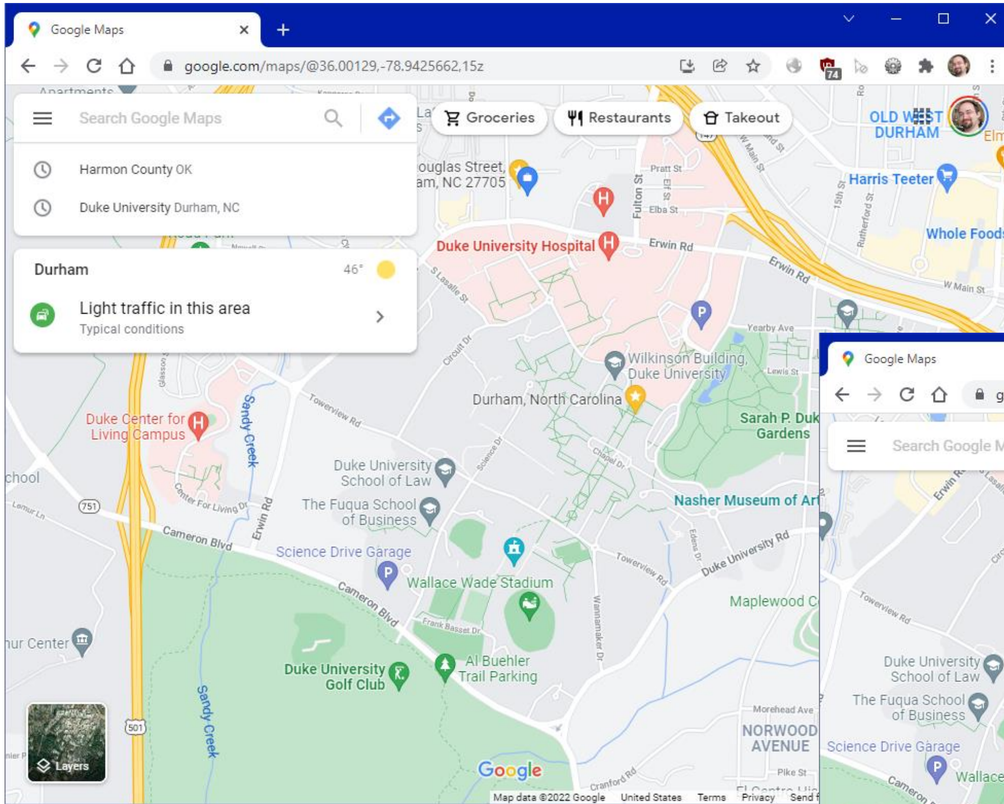
Simple HTML results

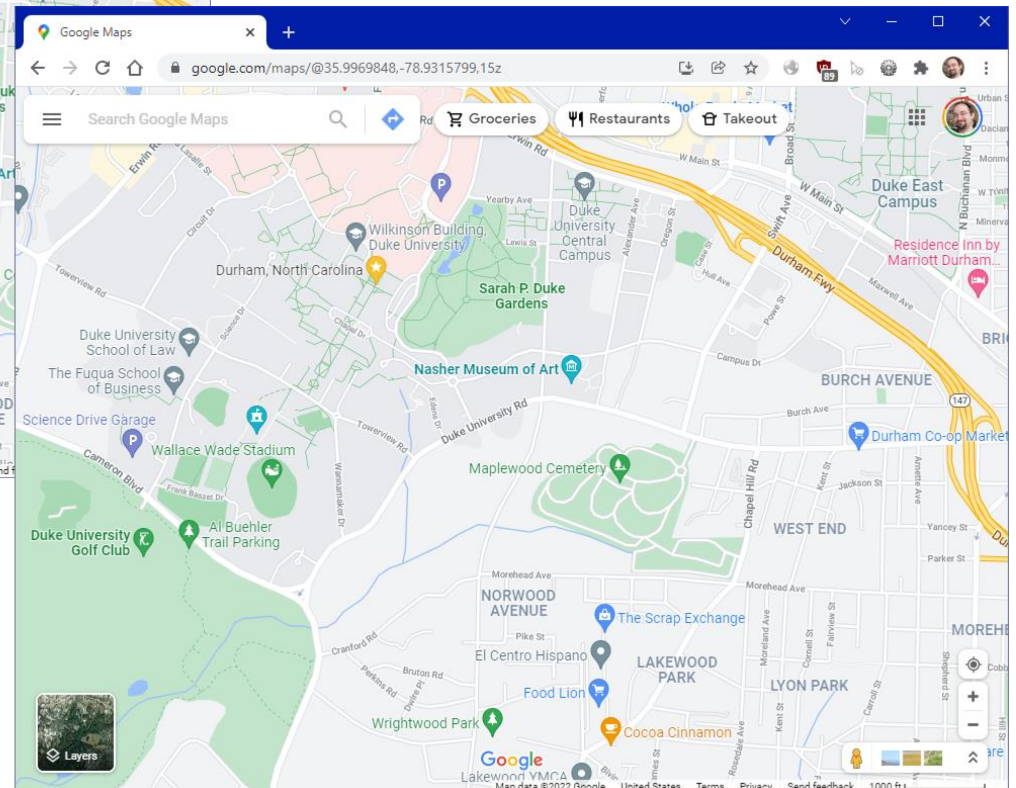



```
...
<a href=“https://www.petfinder.com/dog-breeds/”>
 List of Dog Breeds | Petfinder
</a>
...
```

- The form data hits the server, server generates HTML response

API driven example: Google Maps





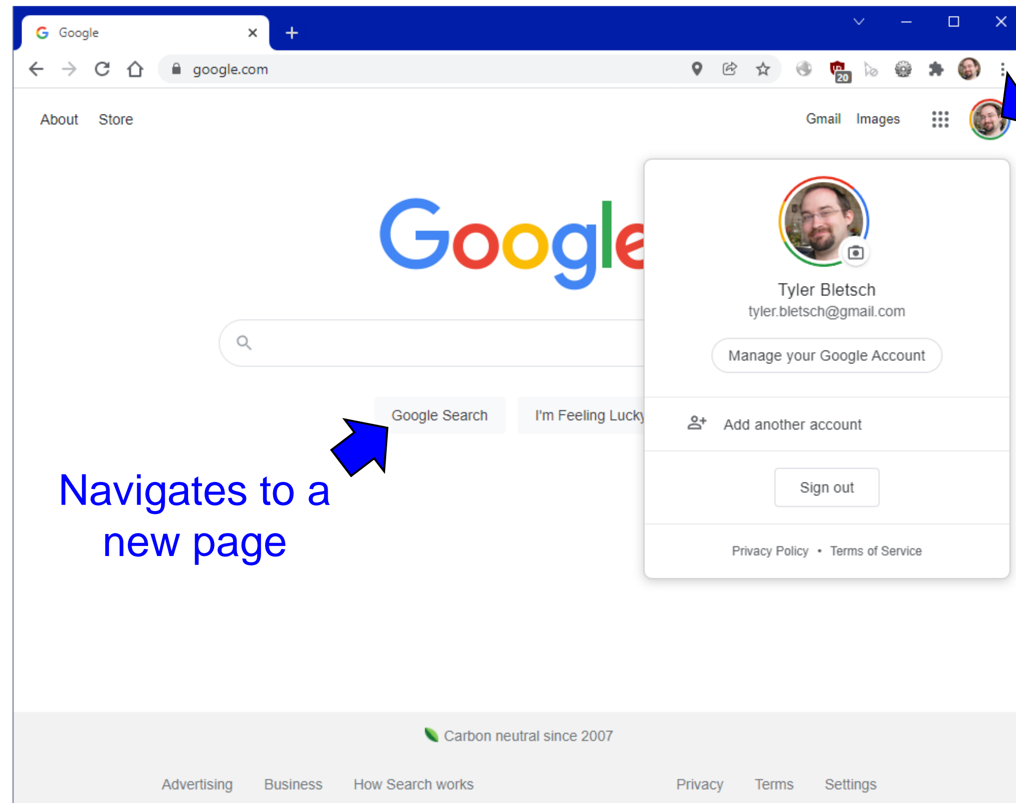
Click and drag the map?

- Java script repositions image tiles, updates browser view
- Can load landmark metadata as JSON, can load new map tiles as images

- Document is generated and updated dynamically, calls server as needed

Hybrid approaches

- You can mix the two approaches. Example:
  - Google homepage has a classic form for search
  - Also has a menu widget with my face on it



Renders a menu on this page

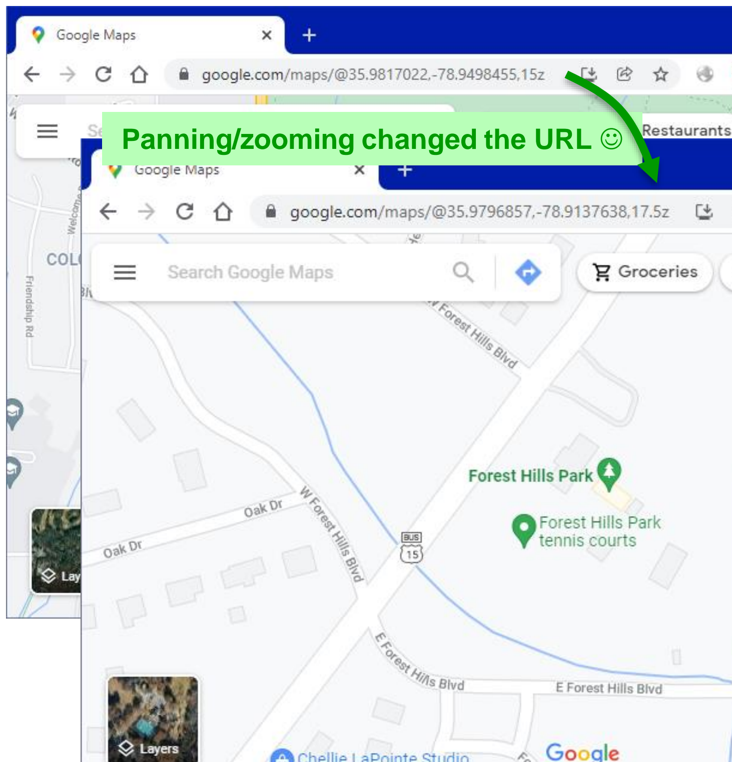
About API-driven sites

- On API-driven sites, tons of software abstraction is used
- Examples:
  - Entities on page are **widgets**, often created an managed as separate software modules
  - Calls to server-side API are often wrapped in abstraction
    - You don't "issue an request for a JSON object", rather "this table has its data sync'd to the server's" (lots of magic hidden behind the scenes)
- You typically use client-side **frameworks** for this, like React

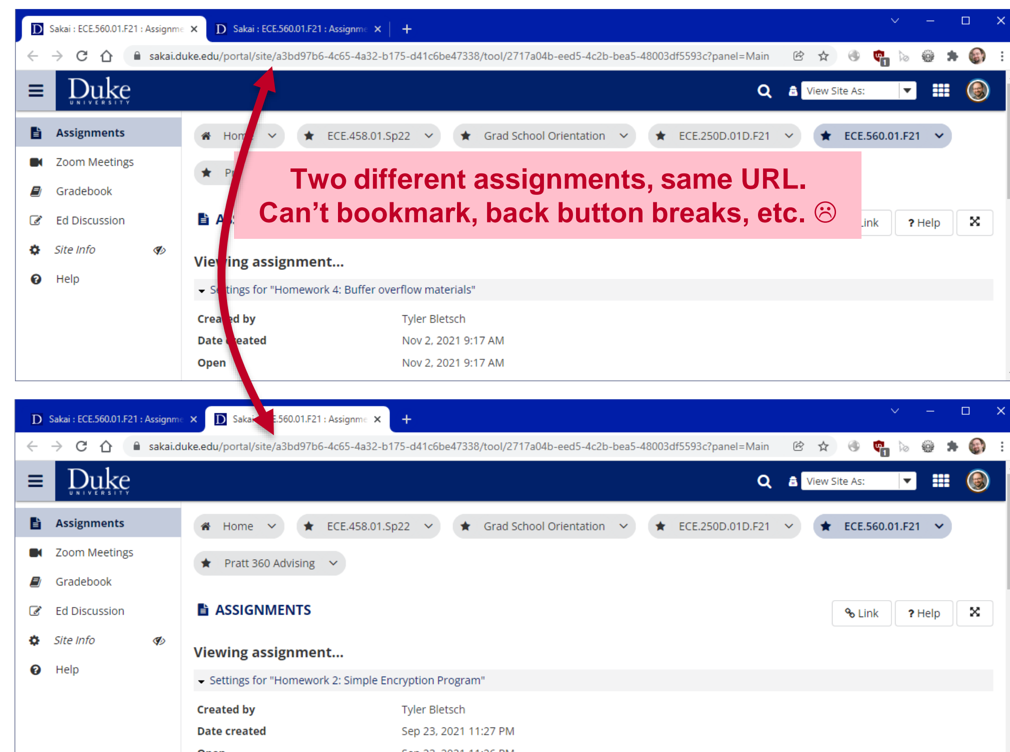
Don't break the browser!

- A common pitfall of API-driven sites: **breaking URLs** and **breaking the back button** ☹
- Once Javascript is rendering your page content rather than it being loaded as HTML from the server, it's easy to forget that URLs are useful and should still work!

Good site: Google Maps



Bad site: Sakai

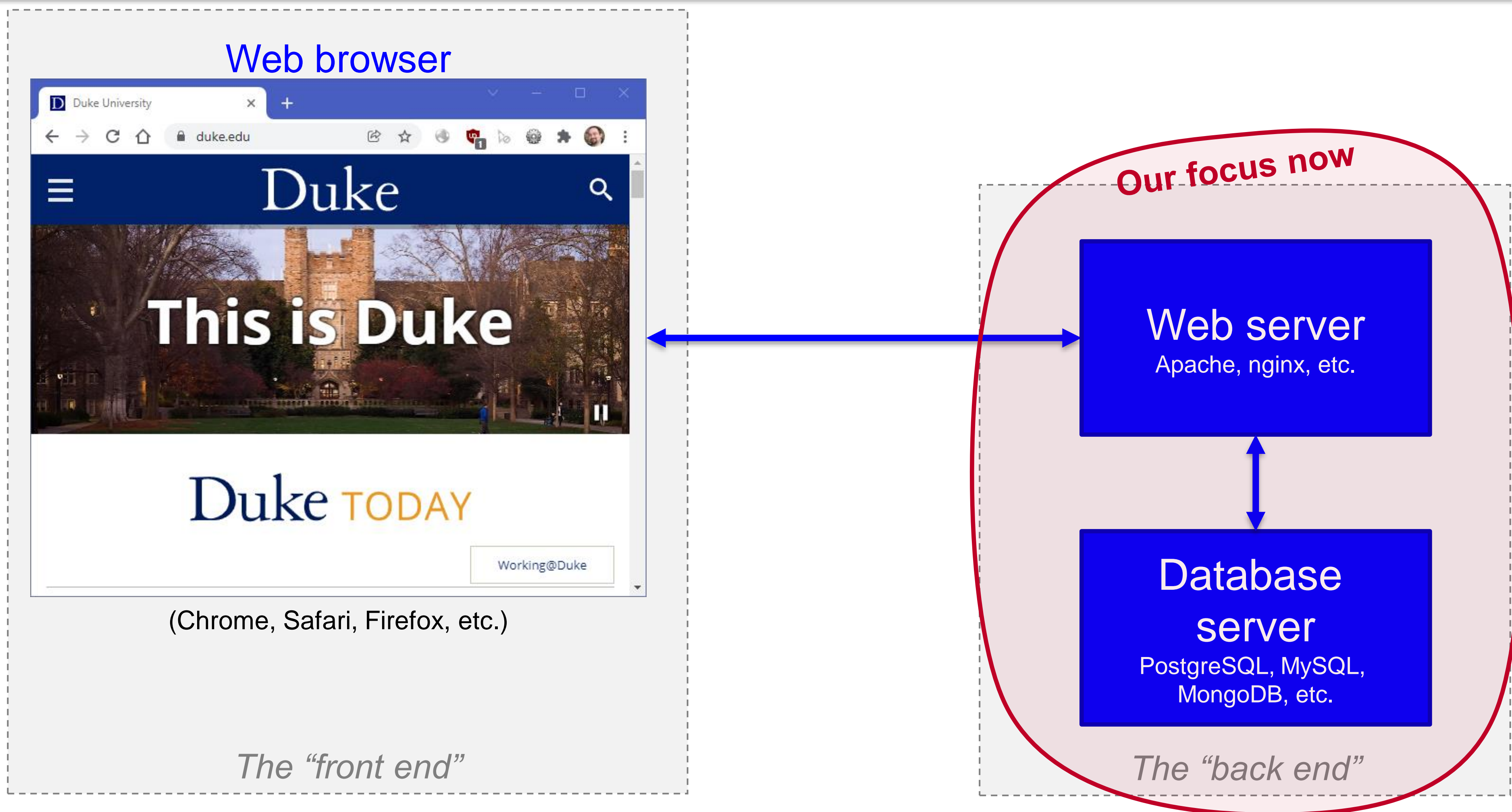


Asynchronous web programming

- For API-driven sites, much of interaction is **asynchronous**
  - Doesn't happen in sequential order, but rather as several requests, each with their own delays
- This can make code nasty. Common solutions:
  - **Callbacks**: Your code provides a function that another module calls back when something happens (such as data being ready)
  - **Promises**: An object representing a request in progress, will either resolve to the complete answer or be rejected as an error
  - **Async/await**: Javascript semantics to help with using promises; can make async code look more like regular sync code
- We won't go deeper now, but this article covers it well.

Where does data live?

Client server topology



- Can get much fancier (middleware, clustering, proxies, etc.)

Storing data server-side

- Most data we care about is all about relationships
  - This ISBN refers to a book
  - This book is in this purchase order and that sales reconciliation
  - All these purchase orders of that book add up to this sum
  - Etc.

- Two main approaches:
  - **Relational database**: Tables of data expressing relationships between entities. Enforces constraints. Speaks **Structured Query Language (SQL)**.
  - **NoSQL database**: JSON documents expressing similar facts. May or may not enforce constraints.

Read more about the tradeoffs

SQL database example

Department Table

Did	Dname	Dacctno
4	human resources	528221
8	education	202035
9	accounts	709257
13	public relations	755827
15	services	223945

primary key (Did)

Employee Table

Ename	Did	Salarycode	Eid	Ephone
Robin	15	23	2345	6127092485
Neil	13	12	5088	6127092246
Jasmine	4	26	7712	6127099348
Cody	15	22	9664	6127093148
Holly	8	23	3054	6127092729
Robin	8	24	2976	6127091945
Smith	9	21	4490	6127099380

foreign key (Did) — primary key (Eid)

(a) Two tables in a relational database

Dname	Ename	Eid	Ephone
human resources	Jasmine	7712	6127099348
education	Holly	3054	6127092729
education	Robin	2976	6127091945
accounts	Smith	4490	6127099380
public relations	Neil	5088	6127092246
services	Robin	2345	6127092485
services	Cody	9664	6127093148

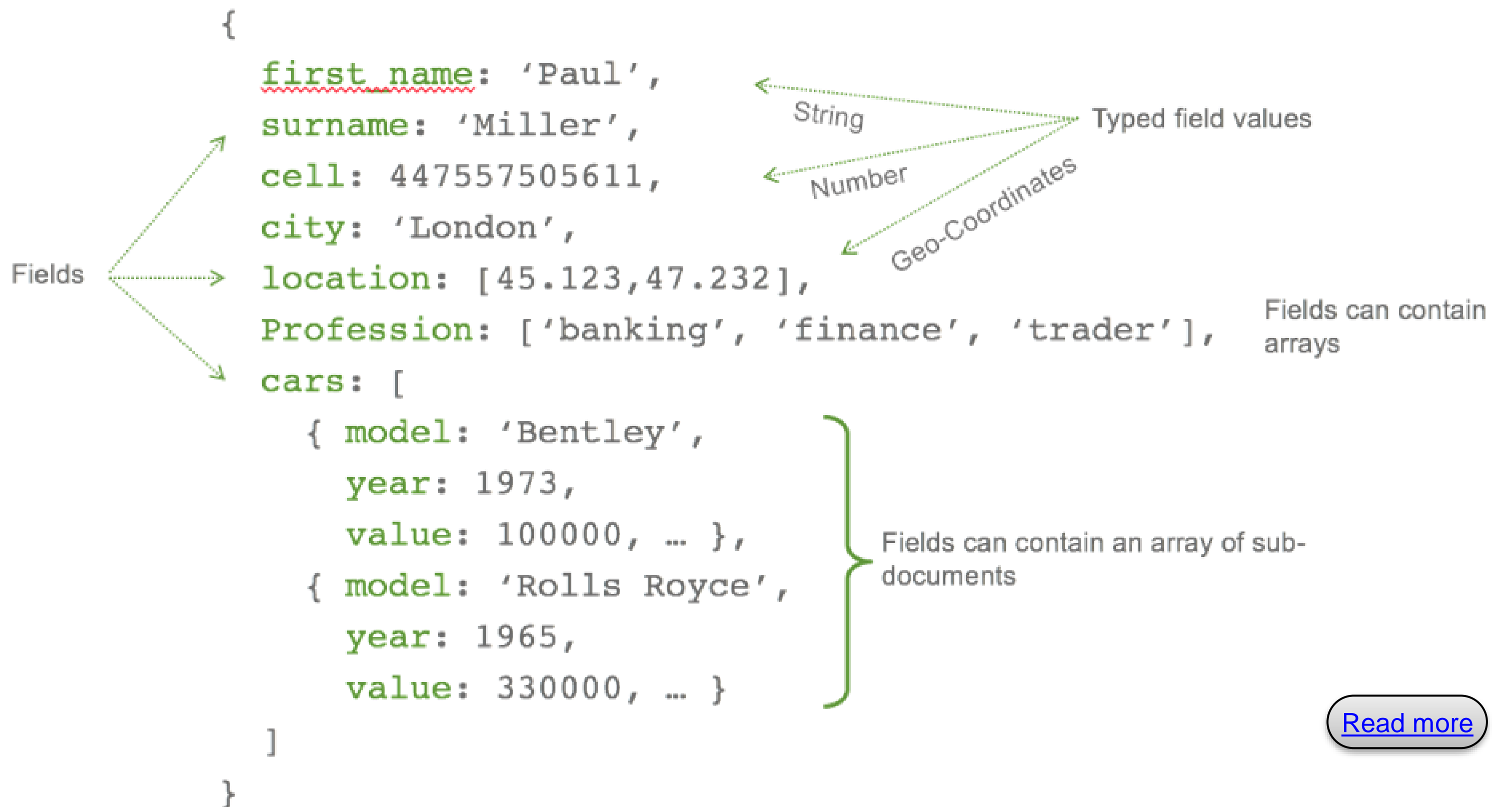
(b) A view derived from the database

- Tables often have **primary keys** (a column that uniquely identifies each entry)
- Records can refer to primary keys of other tables, this is a **foreign key**
- Using these relationship, can describe whole situation with no data duplication
- When tables are created, relationships and constraints are expressed.

Read more

NoSQL database example: MongoDB

- Stores JSON **Documents** in **Collections**
- Not focused on relationships
- Faster setup, more flexibility
- Can usually add constraints with add-on middleware



Read more

Object Relational Mapper (ORM)

- DO NOT USE DATABASES DIRECTLY! No need for that in the modern age!

- Use a framework with an **Object Relational Mapper (ORM)**
  - Automatically converts object-oriented operations to database operations
  - Applicable to both SQL and NoSQL

- Example: Django's ORM

Define the database

```
from datetime import date

from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

    def __str__(self):
        return self.name
```

Make a new database entry

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

Update it

```
>>> b5.name = 'New name'
>>> b5.save()
```

Query for stuff

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

Amount of SQL written: zero!

Bring it all together

Big picture

- Most of this stuff you will **not** worry about day-to-day!
- Why? You will use a **framework** that abstracts most of it

- Ideal development mindset: only program the **unique** things about your problem, avoid boilerplate (stock common code)

Example stacks

- Classic Python web app:
  - Back end: **Python** + **Flask** framework + **PostgreSQL** database
  - Front end: Custom HTML+CSS and standalone Javascript modules
- Modern Python web app:
  - Back end: **Python** + **Django** framework + **Django-REST** module + **PostgreSQL** database
  - Front end: **Javascript** + **React** UI framework
- Modern Javascript web app:
  - "MERN Stack": MongoDB, Express, React, NodeJS
  - Back end: **NodeJS** + **Express** framework + **MongoDB** database
  - Front end: **Javascript** + **React** UI framework
- Modern Ruby web app:
  - Back end: **Ruby** + **Rails** framework + **PostgreSQL** database
  - Front end: **Javascript** + **Angular** UI framework

NodeJS is server-side Javascript

IMPORTANT!!

These are just examples!
Other combos are possible!
There are fine technologies not listed!
Do your own research!

Questions?