

Handling HTTP Requests

SWE 432, Fall 2016

Design and Implementation of Software for the Web

Today

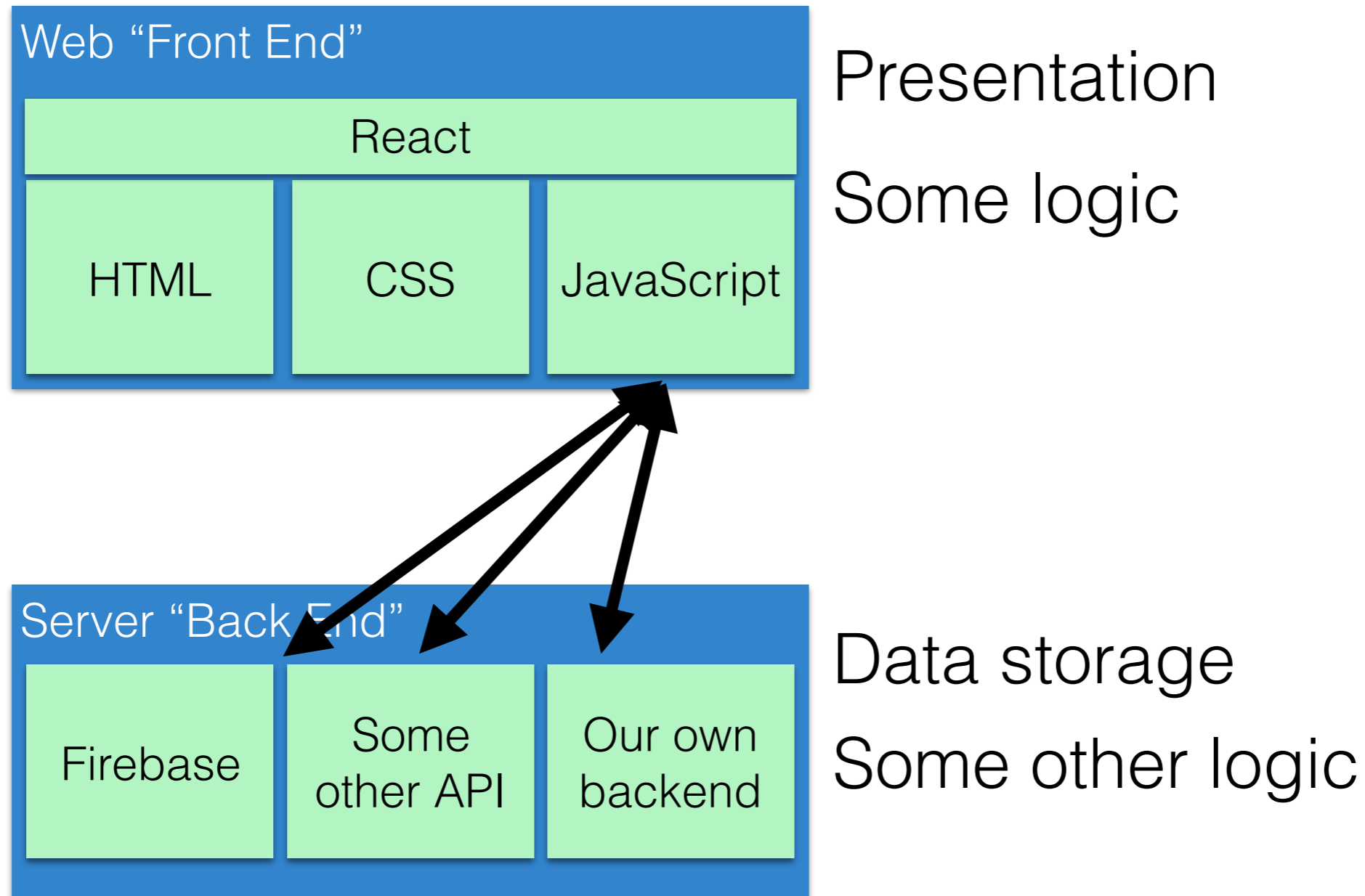
- Handling HTTP requests
- REST
 - What is it?
 - Why use it?
- Handling HTTP Requests with Express

For further reading:

REST: <https://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf>

Express: <https://expressjs.com/>

Handling HTTP Requests



Handling HTTP Requests

Web "Front End"

Server "Back End"



HTTP Request

```
HTTP GET http://api.wunderground.com/api/  
3bee87321900cf14/conditions/q/VA/Fairfax.json
```

HTTP Response

```
HTTP/1.1 200 OK  
Server: Apache/2.2.15 (CentOS)  
Access-Control-Allow-Origin: *  
Access-Control-Allow-Credentials: true  
X-CreationTime: 0.134  
Last-Modified: Mon, 19 Sep 2016 17:37:52 GMT  
Content-Type: application/json; charset=UTF-8  
Expires: Mon, 19 Sep 2016 17:38:42 GMT  
Cache-Control: max-age=0, no-cache  
Pragma: no-cache  
Date: Mon, 19 Sep 2016 17:38:42 GMT  
Content-Length: 2589  
Connection: keep-alive
```

```
{  
  "response": {  
    "version": "0.1",  
    "termsOfService": "http://www.wunderground.com/weather/api/d/terms.html",
```

Key Design Questions

- API: What requests should server support?
- Identifiers: How are requests described?
- Errors: What happens when a request fails?
- Heterogeneity: What happens when different clients make different requests?
- Caching: How can server requests be reduced by caching responses?
- Versioning: What happens when the supported requests change?

REST: REpresentational State Transfer

- Defined by Roy Fielding in his 2000 Ph.D. dissertation
 - Used by Fielding to design HTTP 1.1 that generalizes URLs to URIs
 - http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
- “Throughout the HTTP standardization process, I was called on to defend the design choices of the Web. That is an extremely difficult thing to do... I had comments from well over 500 developers, many of whom were distinguished engineers with decades of experience. That process honed my model down to a core set of principles, properties, and constraints that are now called REST.”
- Interfaces that follow REST principles are called RESTful

Properties of REST

- Performance
- Scalability
- Simplicity of a Uniform Interface
- Modifiability of components (even at runtime)
- Visibility of communication between components by service agents
- Portability of components by moving program code with data
- Reliability

Principles of REST

- Client server: separation of concerns
- Stateless: each client request contains all information necessary to service request
- Cacheable: clients and intermediaries may cache responses.
- Layered system: client cannot determine if it is connected to end server or intermediary along the way.
- Uniform interface for resources: a single uniform interface (URIs) simplifies and decouples architecture

Uniform Interface for Resources

- Originally files on a web server
 - URL refers to directory path and file of a resource
- But... URIs might be used as an identity for any entity
 - A person, location, place, item, tweet, email, detail view, like
 - *Does not matter* if resource is a file, an entry in a database, retrieved from another server, or computed by the server on demand
 - Resources offer an *interface* to the server describing the resources with which clients can interact
 - Example: Firebase path

URI: Universal Resource Identifier

- Uniquely describes a resource
 - <https://mail.google.com/mail/u/0/#inbox/157d5fb795159ac0>
 - https://www.amazon.com/gp/yourstore/home/ref=nav_cs_ys
 - http://gotocon.com/dl/goto-amsterdam-2014/slides/StefanTilkov_RESTIDontThinkItMeansWhatYouThinkItDoes.pdf
 - Which is a file, external web service request, or stored in a database?
 - It does not matter
- As client, only matters what actions we can *do* with resource, not how resource is represented on server

Intermediaries

Web "Front End"

"Origin" server



The diagram illustrates the flow of an HTTP request and response between a Web Front End and an Origin server. A thick black arrow points from the Web Front End to the Origin server, and another thick black arrow points from the Origin server back to the Web Front End.

HTTP Request

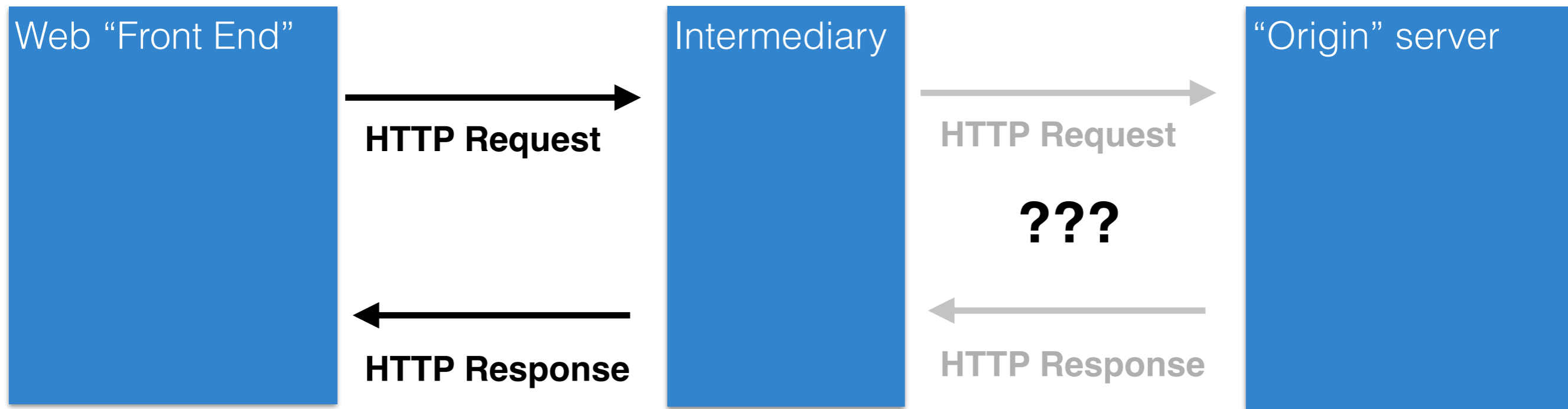
```
HTTP GET http://api.wunderground.com/api/  
3bee87321900cf14/conditions/q/VA/Fairfax.json
```

HTTP Response

```
HTTP/1.1 200 OK  
Server: Apache/2.2.15 (CentOS)  
Access-Control-Allow-Origin: *  
Access-Control-Allow-Credentials: true  
X-CreationTime: 0.134  
Last-Modified: Mon, 19 Sep 2016 17:37:52 GMT  
Content-Type: application/json; charset=UTF-8  
Expires: Mon, 19 Sep 2016 17:38:42 GMT  
Cache-Control: max-age=0, no-cache  
Pragma: no-cache  
Date: Mon, 19 Sep 2016 17:38:42 GMT  
Content-Length: 2589  
Connection: keep-alive
```

```
{  
  "response": {  
    "version": "0.1",  
    "termsOfService": "http://www.wunderground.com/weather/api/d/terms.html",
```

Intermediaries



- Client interacts with a resource identified by a URI
- But it never knows (or cares) whether it interacts with origin server or an unknown intermediary server
 - Might be randomly load balanced to one of many servers
 - Might be cache, so that large file can be stored locally
 - (e.g., GMU caching an OSX update)
 - Might be server checking security and rejecting requests

Challenges with intermediaries

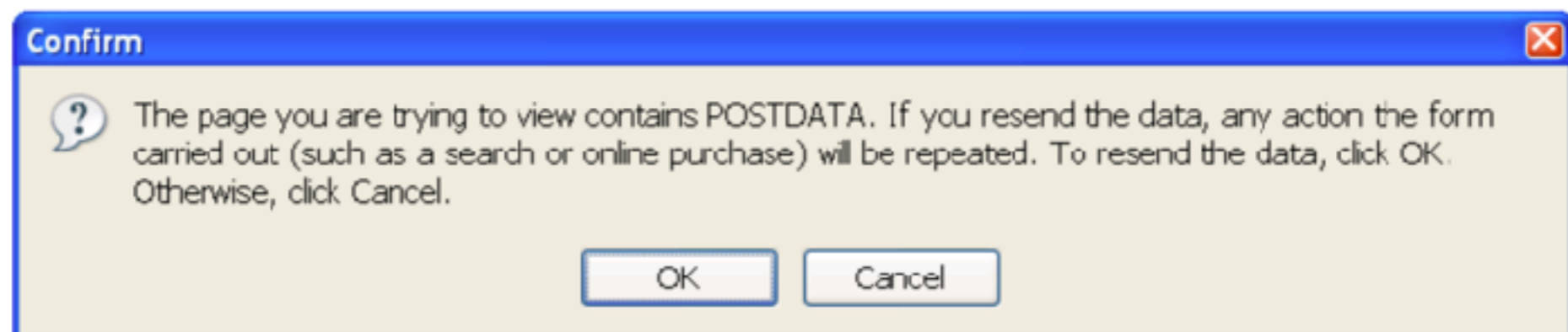
- But can all requests really be intercepted in the same way?
 - Some requests might produce a change to a resource
 - Can't just cache a response... would not get updated!
 - Some requests might create a change every time they execute
 - Must be careful retrying failed requests or could create extra copies of resources

HTTP Actions

- How do intermediaries know what they can and cannot do with a request?
- Solution: HTTP Actions
 - Describes what will be done with resource
 - GET: retrieve the current state of the resource
 - PUT: modify the state of a resource
 - DELETE: clear a resource
 - POST: initialize the state of a new resource

HTTP Actions

- GET: safe method with no side effects
 - Requests can be intercepted and replaced with cache response
- PUT, DELETE: idempotent method that can be repeated with same result
 - Requests that fail can be retried indefinitely till they succeed
- POST: creates new element
 - Retrying a failed request might create duplicate copies of new resource



Specifying HTTP Actions w/ jQuery

- method field of \$.ajax can be used to specify method
 - “GET”, “PUT”, “DELETE”, “POST”

```
$.ajax({  
  url: “http://webservice.com/resource/235”,  
  method: “PUT”,  
  data: { “name”: “Best resource ever!” }  
});
```


Versioning

- Your web service just added a great new feature!
 - You'd like to expose it in your API.
 - But... there might be old clients (e.g., websites) built using the old API.
 - These websites might be owned by someone else and might not know about the change.
 - Don't want these clients to throw an error whenever they access an updated API.

Cool URIs don't change

- In theory, URI could last forever, being reused as server is rearchitected, new features are added, or even whole technology stack is replaced.
- “What makes a cool URI?
A cool URI is one which does not change.
What sorts of URIs change?
URIs don't change: people change them.”
 - <https://www.w3.org/Provider/Style/URI.html>
 - Bad:
 - <https://www.w3.org/Content/id/50/URI.html> (What does this path mean? What if we wanted to change it to mean something else?)
- Why might URIs change?
 - We reorganized our website to make it better.
 - We used to use a cgi script and now we use node.JS.

URI Design

- URIs represent a contract about what resources your server exposes and what can be done with them
- Leave out **anything that might change**
 - Content author names, status of content, other keys that might change
 - File name extensions: response describes content type through MIME header not extension (e.g., .jpg, .mp3, .pdf)
 - Server technology: should not reference technology (e.g., .cfm, .jsp)
- Endeavor to make all changes backwards compatible
 - Add new resources and actions rather than remove old
- If you must change URI structure, support old URI structure **and** new URI structure

Describing Responses

- What happens if something goes wrong while handling HTTP request?
 - How does client know what happened and what to try next?
- HTTP offers response status codes describing the nature of the response
 - 1xx Informational: Request received, continuing
 - 2xx Success: Request received, understood, accepted, processed
 - 200: OK
 - 3xx Redirection: Client must take additional action to complete request
 - 301: Moved Permanently
 - 307: Temporary Redirect

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Describing Errors

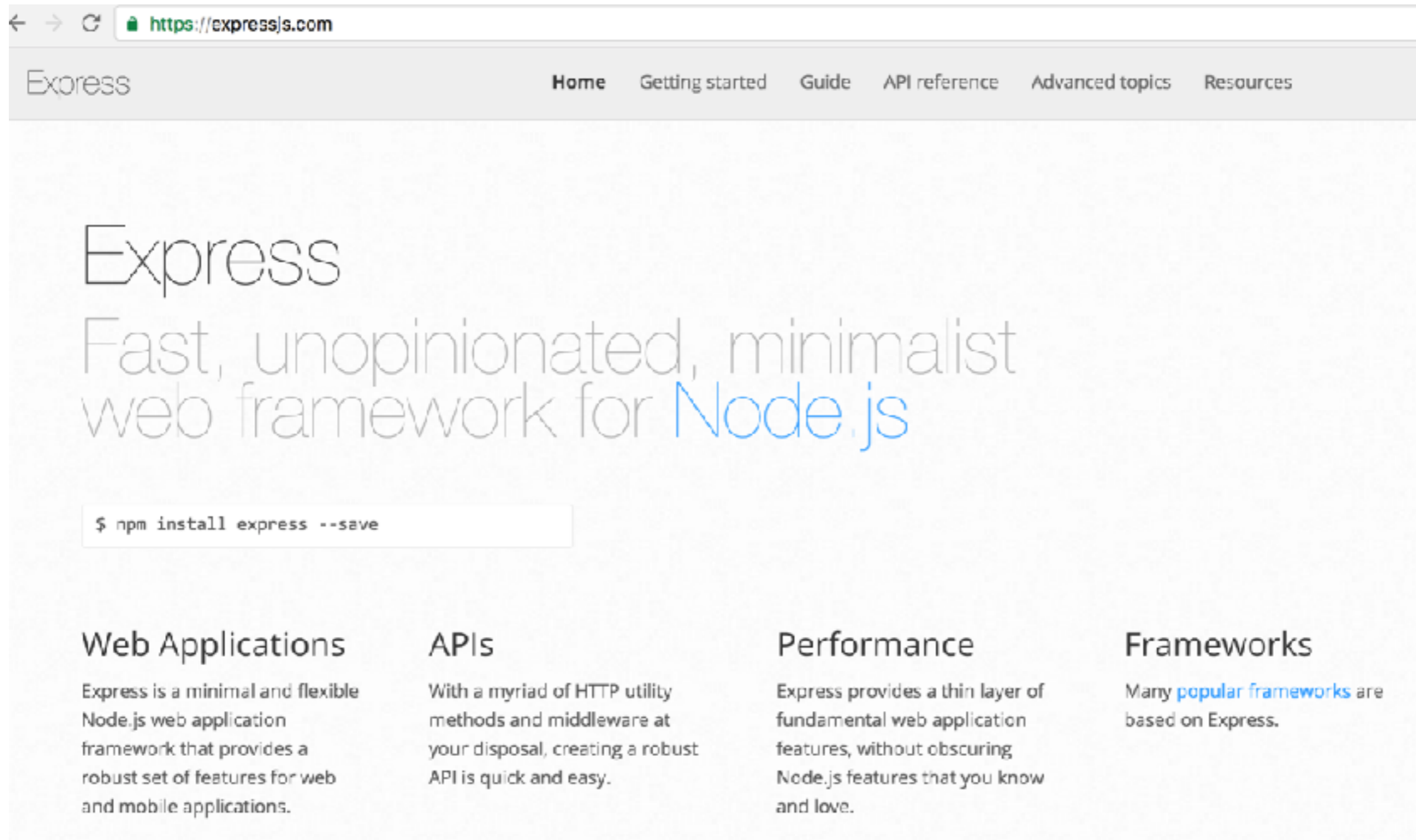
- 4xx Client Error: client did not make a valid request to server. Examples:
 - 400 Bad request (e.g., malformed syntax)
 - 403 Forbidden: client lacks necessary permissions
 - 404 Not found
 - 405 Method Not Allowed: specified HTTP action not allowed for resource
 - 408 Request Timeout: server timed out waiting for a request
 - 410 Gone: Resource has been intentionally removed and will not return
 - 429 Too Many Requests

Describing Errors

- 5xx Server Error: The server failed to fulfill an apparently valid request.
 - 500 Internal Server Error: generic error message
 - 501 Not Implemented
 - 503 Service Unavailable: server is currently unavailable

Handling HTTP Requests in Express

- Node.js package for expressing rules about how to handle HTTP requests



The image shows a screenshot of the Express.js website homepage. The browser address bar displays `https://expressjs.com`. The navigation menu includes [Home](#), [Getting started](#), [Guide](#), [API reference](#), [Advanced topics](#), and [Resources](#). The main heading is "Express", followed by the tagline "Fast, unopinionated, minimalist web framework for Node.js". A code block shows the command `$ npm install express --save`. Below this, there are four columns of text describing the framework's features: Web Applications, APIs, Performance, and Frameworks.

Express

Fast, unopinionated, minimalist web framework for Node.js

```
$ npm install express --save
```

Web Applications
Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

APIs
With a myriad of HTTP utility methods and middleware at your disposal, creating a robust API is quick and easy.

Performance
Express provides a thin layer of fundamental web application features, without obscuring Node.js features that you know and love.

Frameworks
Many popular frameworks are based on Express.

Hello World from Last Time

```
var express = require('express');
var app = express();
var port = process.env.port || 3000;
app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.get('/goodbye', function (req, res) {
  res.status(500);
});

app.listen(port, function () {
  console.log('Example app listening on port' + port);
});
```


Core concept: Routing

- The definition of end points (URIs) and how they respond to client requests.
 - `app.METHOD(PATH, HANDLER)`
 - METHOD: all, get, post, put, delete, [and others]
 - PATH: string
 - HANDLER: call back

```
app.post('/', function (req, res) {  
  res.send('Got a POST request');  
});
```

Route paths

- Can specify strings, string patterns, and regular expressions
 - Can use ?, +, *, and ()
- Matches request to root route

```
app.get('/', function (req, res) {  
  res.send('root');  
});
```

- Matches request to /about

```
app.get('/about', function (req, res) {  
  res.send('about');  
});
```

- Matches request to /abe and /abcde

```
app.get('/ab(cd)?e', function (req, res) {  
  res.send('ab(cd)?e');  
});
```

Route parameters

- Named URL segments that capture values at specified location in URL
 - Stored into `req.params` object by name
- Example
 - Route path `/users/:userId/books/:bookId`
 - Request URL `http://localhost:3000/users/34/books/8989`
 - Resulting `req.params`: `{ "userId": "34", "bookId": "8989" }`

```
app.get('/users/:userId/books/:bookId', function(req, res) {  
  res.send(req.params);  
});
```

Request object

- Enables reading properties of HTTP request
 - `req.body`: JSON submitted in request body (*must* define body-parser to use)
 - `req.ip`: IP of the address
 - `req.query`: URL query parameters

Response object

- Enables a response to client to be generated
 - `res.send()` - send string content
 - `res.download()` - prompts for a file download
 - `res.json()` - sends a response w/ JSON header
 - `res.redirect()` - sends a redirect response
 - `res.sendStatus()` - sends only a status message
 - `res.sendFile()` - sends the file at the specified path

```
app.get('/users/:userId/books/:bookId', function(req, res) {  
  res.json({ "id": req.params.bookID });  
});
```

Error handling

- Express offers a default error handler
- Can specify error explicitly with status
 - `res.status(500);`

Demos

- <https://github.com/expressjs/express/tree/master/examples>
- <https://github.com/tlatoza/ProgrammingStudies/blob/master/server.js>
- Examples from last lecture updated with more routing and params tricks