ECS 189
WEB PROGRAMMING

5/15

Photobooth

- A photo storage site that knows something about what the photos are of.
- This week: upload photos, put into database, edit tags interactively.
- Work in groups of up to three; due Mon 22.

Photobooth

- We looked on Friday at using the browser's XMLHttpRequest and formData objects to upload image files.
- Today we look at the server side, then creation of database.

Our Server does 3 things

GET request, http://138.68.25.50 :???/pathname

- gives a static file, pathname is based at /public, or 400 message if no such file exist

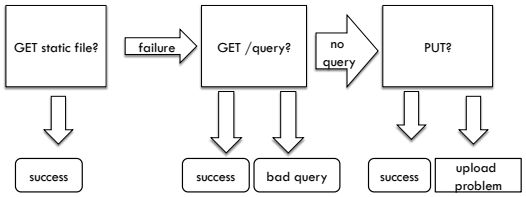
GET request, http://138.68.25.50.???/query?img=hula

- returns labels associated with image named hula.jpg or hula.jpeg, or 400 message if no such image on server

PUT request to http://138.68.25.50.???/

- uploads an image to /public and responds with a 201 message if successful or 500 message if not

Pipeline so far



- May need to add some error handling

A server using express

- Express gives us an alternative syntax for writing a server:

```
var express = require('express');
var formidable = require('formidable');
  // for using forms
var app = express();
```

- app is now an express server object

Static server in express

```
app.use(express.static('public'));
```

- The 4 rather dense lines we had in Node are encapsulated into one nice library function
- "use" in express puts the function inside the parens into a pipeline of functions that are called on the (request, response) pair for a given HTTP request
- Pipeline functions are called "middleware"
- static exits if it finds the file, but not all middleware functions do

Pipeline stage for queries

- This is our own query handling bit. Also exits if it succeeds. answer and sendCode are our functions.

```
app.get('/query', function (request, response)
  {   console.log("query");
  query = request.url.split("?")[1]; // get query string
  if (query) {
      answer(query, response);   }
  else { sendCode(400, response,"bad query");   }});
```

Pipeline stage for PUT requests

```
app.post('/', function (request, response){
  var form = new formidable.IncomingForm();
  form.parse(request);   // get file handle, name
  // two anonymous callback functions
  form.on('fileBegin', function (name, file)
      {   file.path = __dirname + '/public/' +
      file.name;   });   // direct to file in /public
  form.on('end', function ()
      {  console.log('success');
        sendCode(201,response,'recieved file'); }); });
```

What to do with files?

- Since we stored the files in /public, they are available to the browser (and anyone else…) already:

  http://138.68.25.50:???/hula.jpg

- So photobooth can display any uploaded image just by setting the "src" field on an image element of the DOM

Faded image

- When uploading and processing the image, which will take a bit of time, Dani's design calls for us to display a faded version of the image.
- But it's not yet available at the URL on the server!
- We have the file name on the client machine that is running the browser. We can't just set "src" to that file name. Why not?

Faded image

- We have the file name on the client machine that is running the browser. We can't just set "src" to that file name. Why not?
- In general, the browser does not have access to files on a user's machine.
- For instance, I don't want to give every Website I look at access to your grades. This would be a huge security hole.
- So what to do?

Faded image

- The user gave us access to a file handle for this particular file.
- File handles are what let us read data.
- So let's read in the data!
- Again, access Javascript functionality via an object
- Again, file reading is an asynchronous action (going to disk takes time, although orders of magnitude less than going over the internet), so we use a callback function

See fader.html

```
var selectedFile =
  document.getElementById('fileSelector').files[0];
var image = document.getElementById('theImage');
var fr = new FileReader();

// anonymous callback run when file load is complete
fr.onload = function ()
  { image.src = fr.result; };
fr.readAsDataURL(selectedFile); // begin reading
```

Photobooth once we have pictures

- Once the user has uploaded some pictures, Photobooth should remember that they are there
- When it comes up, it should display all the photos it has
- Needs to remember a list of image files on server, and ultimately also their labels and whether they are favorites or not
- Time for a database!

Databases

- A database is
  - A file, or collection of files, storing data, usually on a server's disks
  - Software for interfacing to that data
- Files are stored on disks and (when not in a database) are read from beginning to end
- Say we're looking for something near the end, or in the middle, even. This can take forever.
- Databases generally have an in index to help you find things quickly

A big database

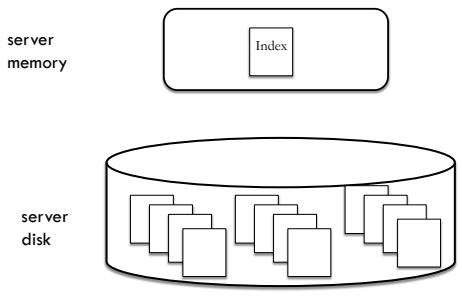


Photo upload

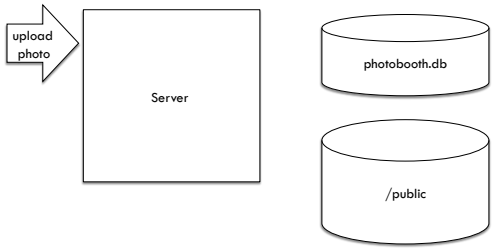
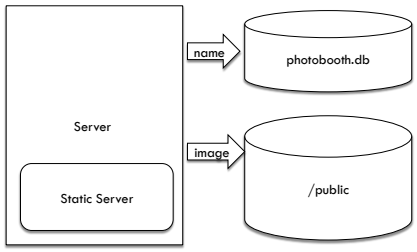
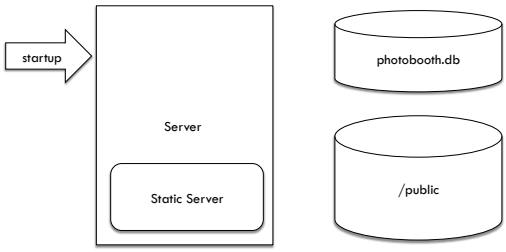


Photo upload

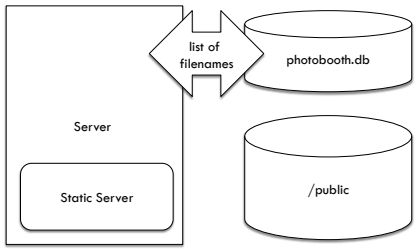


Startup



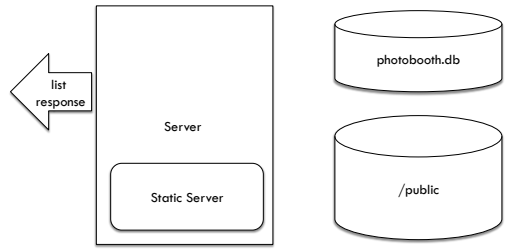
Startup

When photobooth starts, needs a list of photos

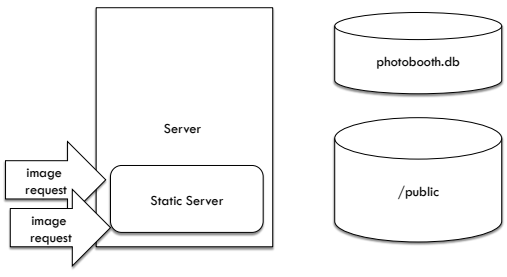


Startup

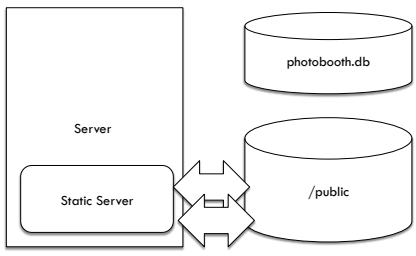
When photobooth starts, needs a list of photos



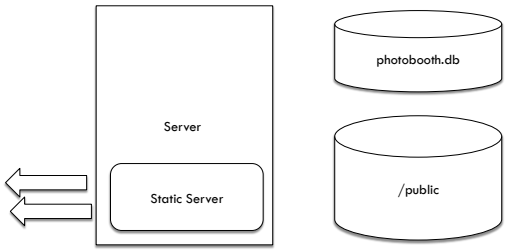
Photos requested as static files



Photos requested as static files



Photos requested as static files



Database

- A database is made up of tables
- A table is similar to a spreadsheet
- Columns can contain arrays or strings as well as numbers

imageFile	labels	favorite
hula.jpg	Dance, Performing Arts, Sports, Entertainment	1
eagle.jpg	Bird, Beak, Bird Of Prey, Eagle	0

SQL

- Most databases support SQL as the API for the software interface between the user and the data
- SQL is a declarative language – you specify what you want, not what computations the database should use to get it for you.
- Let the clever database programmers figure out the best way to get the data
- Loose standard, many variants

sqlite3

- sqlite3 is a node database module
- It implements the database as a single file, although it uses fancy file access techniques to be able to pick out records in the middle, using an index
- Node module for sqlite3 gives us an SQL interface we can use in our Javascript code
- We'll only need to initialize the database once, so we write a stand-alone node program to do this
- Our server later will open the database and make SQL queries to add photos, add labels, etc.

Connecting with sqlite

- Install sqlite3 (npm install squlite3)
- Require it at the top of the file
- Open a database file:

  var dbFile = "photos.db";

  var db = new sqlite3.Database(dbFile);
- db is now a variable that has methods to run SQL commands.
- Stuff written to the database by our program will be stored in the file "photos.db"

Making a table in the database

- Our database will contain one table
- SQL CREATE command makes a table, defines its columns:

CREATE TABLE PhotoLabels (fileName TEXT , labels TEXT, favorite INT)

Issuing SQL commands

- In node.js, we put the command into a string, and we pass the string to the db object:

```
var cmdStr = "CREATE TABLE Photos (fileName TEXT ,
  labels TEXT, favorite INT)";

db.run(cmdStr);
```

(Javascript is pink and SQL is green)

Primary key

- Rows correspond to photos
- This will be the primary way we look up data
- Insist each row in this column has to be unique; we don't want two rows for same file
- Insist it has to not be null (present in all records)
- Define it as the primary key = easiest thing to access via the index

```
fileName TEXT UNIQUE NOT NULL PRIMARY KEY
```

Wednesday: more database