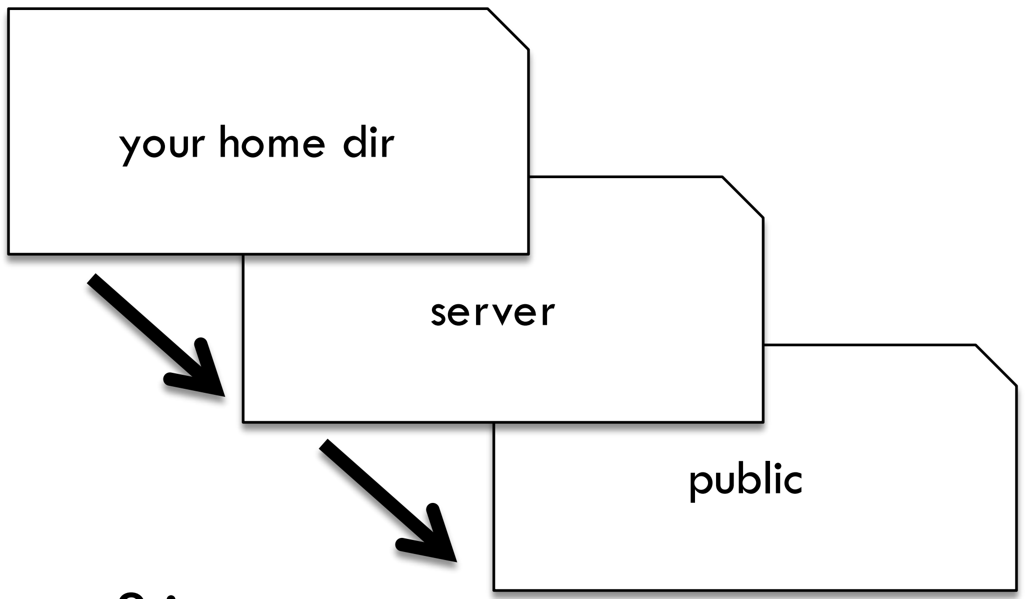
ECS 162

Web Programming

5/13

Directory structure for project



/server --- contains miniServer3.js, or whatever you want to call it

/server/public --- contains all static files including html, css, and js files that will be run on the browser. Your server should automatically serve any file in /public.

- Why is it a bad idea to have to change the server code every time you put a new file in /public?

- Why is it a bad idea to server files out of "." (the directory containing the server code)?

- Why is it a bad idea to have to change the server code every time you put a new file in /public?

  For a large Web site, you'd have to do this every day. It is better to have the server serve any file that gets put into /public automatically.

- Why is it a bad idea to server files out of "." (the directory containing the server code)?

  A malicious user could get the server code and any other code it uses, and look for security holes.

Last time: Initialize database

To make a new table: CREATE TABLE flashcards (

user INT, english TEXT, korean TEXT, seen INT, correct INT)

To delete a table: DROP TABLE flashcards

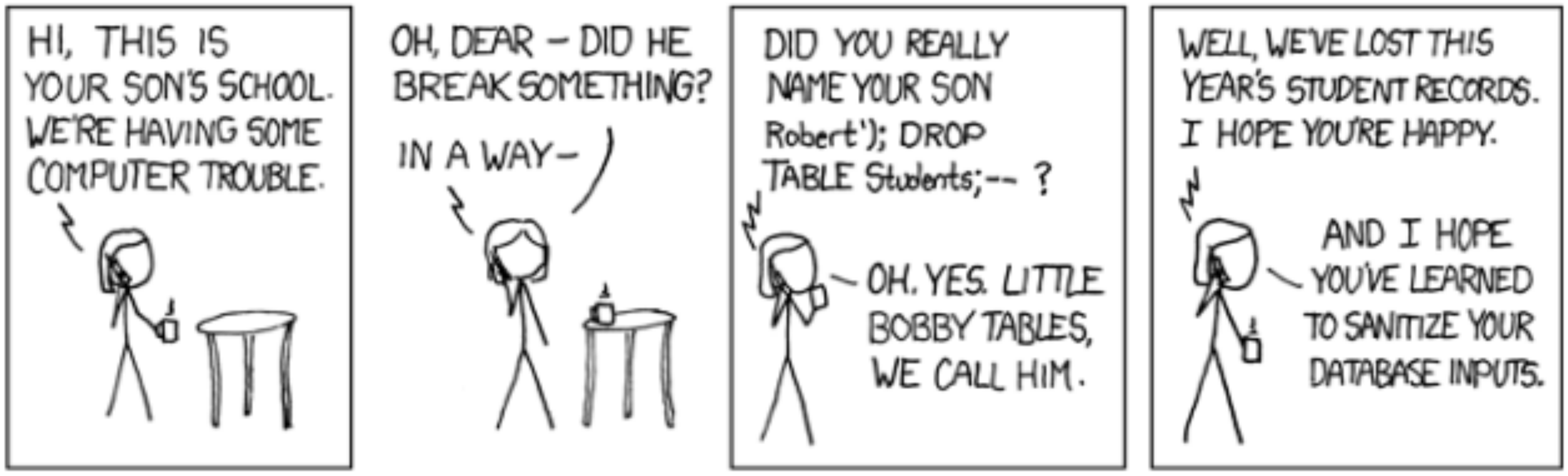
Rowid	**User**	**English**	**Korean**	**times seen**	**times correct**
1	1	Excuse me…	실례합니다…	0	0
2	1	Where is the train station?	기차역은 어디 있습니까?	0	0

Putting stuff into the database

- Text ultimately comes from the user, who could be malicious; on the Web anybody can go to our Web site and try to break it.
- Never paste user input (or any untrusted input) into an SQL command, or any command that is going to be executed; it basically lets someone run any code that they want to on your server.
- We had the same issue with using “innerHTML”; html is a language that gets executed by the browser.

Protecting the database

- To get this XKDC comic, we need to know that "DROP TABLE" is the SQL opposite of "CREATE TABLE"; it is how we delete a table.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Sanitizing inputs

INSERT into Flashcards (user, english, korean, seen, correct) VALUES (1, @0, @1, 0, 0)

- This is a template for an insertion command.
- The list of values goes into the corresponding list of columns
- The parameters @0 and @1 will contain the English and Korean text
- Sqlite3 automatically checks that values supplied for the parameters have the correct type, no forbidden characters
- This is called sanitization

Running the SQL from Javascript

```
const cmdStr = 'INSERT into Flashcards (user, english, korean,  seen, correct) VALUES (1, @0, @1, 0, 0)'
db.run(cmdStr, eng, kor, insertCallback);
```

- Just like before, put the SQL command in a string, and call db.run on the string.
- You can specify parameters @0, @1 in the db.run command, eg. from the data returned by Google Translate

Better version…

```
db.run(cmdStr, eng, kor, insertCallback);


function insertCallback(err) {
        if (err) { console.log(err); }
}
```

- Database code is hard to debug, always try to catch error messages in callback
- In this case, callback should also return response to browser to indicate flashcard has been stored.

Comic

- Where is the insert command in the comic?
- And where is the callback function?

Getting output

```
SELECT *
FROM Flashcards
WHERE user = 1
```

- Returns all rows in data base with user 1

Select statement

SELECT columns FROM table WHERE Boolean

- Handy example:

SELECT * FROM Flashcards

- Dumps the whole table. The * means all columns, and omitting the WHERE gets all rows.

More WHERE expressions

WHERE seen < 3

WHERE seen < 3 and correct < 1

- Handy when we are looking for cards the user has not seen much yet

WHERE googleid = 587302830

- We'll need this one when we add a table of users to connect a user to her data when she logs in

Callbacks for data

```
 db.get( ‘SELECT * FROM Flashcards WHERE user = 1',
   dataCallback);


function dataCallback(err, rowData) {
     if (err) { console.log("error: ",err); }
     else { console.log("got: ",rowData,"\n");   }
}
```

- rowData is an object containing data from one row.
- If more than one row matches, we get only the first.

Gets an array of rows

```
db.all( ‘SELECT * FROM Flashcards WHERE user =
   1',  arrayCallback);


function arrayCallback(err, arrayData) {
   if (err) { console.log("error: ",err,"\n");
   } else { console.log("array: ",arrayData,"\n");  }}
```

- arrayData contains an array of objects, each object contains one row.

Limiting number of rows

```
db.all(‘SELECT * FROM Flashcards WHERE user = 1
   LIMIT 12', arrayCallback);
```

- Could be many rows that have a particular tag.
- We won't want to send hundreds down to the browser; limit number chosen.

Changing a row

- We could always re-write an entire row to change it. But better to just do specific cells:

```
UPDATE Flashcards  SET seen = 1 WHERE rowid = 73
```

- The WHERE clause selects the row…or rows! Always safe to choose by rowid since that is the unique primary key.

Changing a row

UPDATE Flashcards SET seen = 1 WHERE rowid = 1

- Warning! Omitting WHERE changes all the selected column in all the rows!

Use = not == in both SET and WHERE.

Database is asynchronous

- Commands are not necessarily done in the order we issue them.

  db.run(‘UPDATE Flashcards SET seen = 1 WHERE rowid = 1’), errorCallback);

db.get('SELECT seen from Flashcards WHERE rowid =1', dataCallback);

- Sometimes the SELECT commands sees seen = 1, sometimes seen = 0 – it depends on whether the UPDATE finished before the SELECT occurred.

Enforcing ordering

- Sometimes we don't care if commands are executed in order, eg. insertion of three rows.
- Sometimes we do care, eg. INSERT before UPDATE, UPDATE before SELECT.
- To enforce ordering, use the callbacks.
- Example: Issue the SELECT command in the callback function for the INSERT.

Order commands with callbacks

```
cmdStr = 'INSERT into Flashcards (user, english,
korean,  seen, correct) VALUES (1, @0, @1, 0, 0) ';
db.run(cmdStr,eng, kor, insertCallback);


function insertCallback(err) {
   if (err) { console.log("insert error!", err); }
   else {
        lookAtRowid();  // function that issues SELECT
   }
```

Other ways people build DBs

- We could construct the database using sqlite3 directly (note! NOT THE SAME as sqlite!)
- We can access it from the sqlite3 command line:

```
amenta@cs162:~/server$ sqlite3
sqlite> attach database ”Flashcard.db" as db;
sqlite> select user from Flashcards where
                                        rowid = 1;
sqlite> detach database db;
```

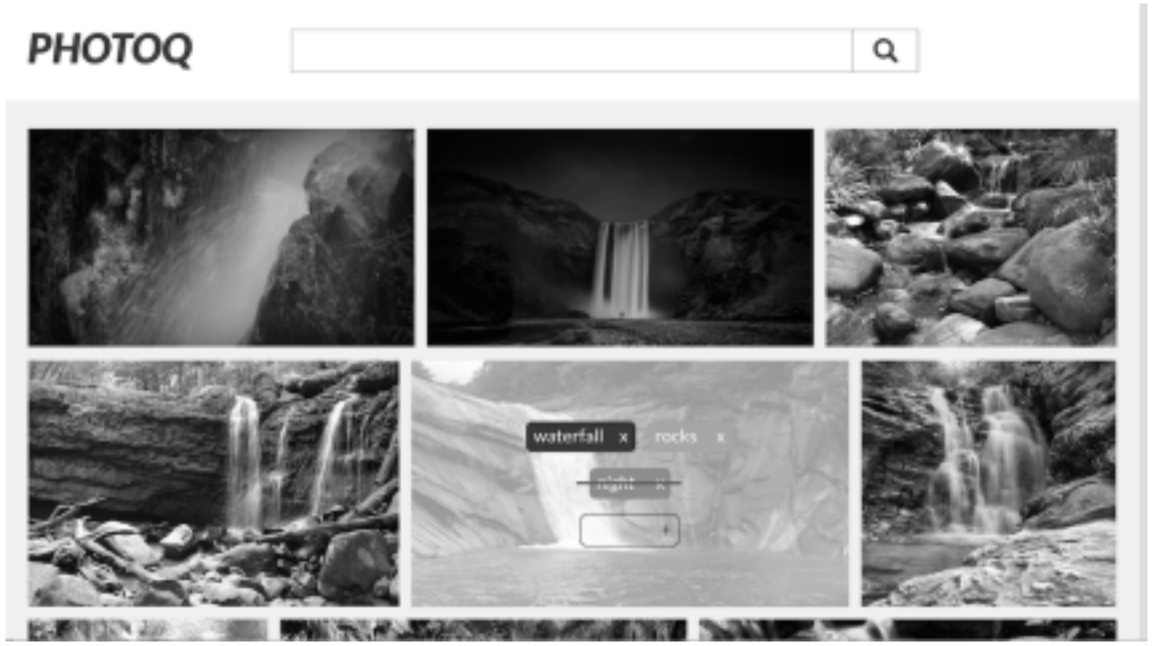
Next topic: React

- Moving from Server to Browser

- React is a front-end user interface library
- Developed by Facebook. Very popular.
- Helps in several ways
- First, enforces modular design of UI code by organizing UI into components.

React photo gallery component

- Gallery component contains image tiles
- Image tile component contains picture and controls
- Controls contains tags, add-tag box, close button
- Tag contains text, delete button
- Lowest level components contain HTML elements



Components as "virtual elements"

- Components and real HTML elements can be combined in a hierarchy to build up Web pages
- Components have properties like elements

  eg. a Tag has a "text" property, just like an img has a src property.
- Putting pieces of UIs into these "virtual elements" lets us write modular software

Modularity allows code reuse

- We re-used this photo gallery!
- I found it in an article called "15 Awesome React Components", also mentioned elsewhere…
- It was written by a developer called Sandra Gonzalez; I got it off her github.
- This year I think we'll re-use someone's flipping card component for the flashcard review section.

Virtual DOM

- Second advantage: Programmer’s illusion that the entire DOM is re-constructed at every event (eg. user clicks button, React re-builds entire DOM).
- User actions and other events change basic state variables, and then React generates the DOM based on new state variables.
- Always show same display in same state, whatever path through the controls took you there.