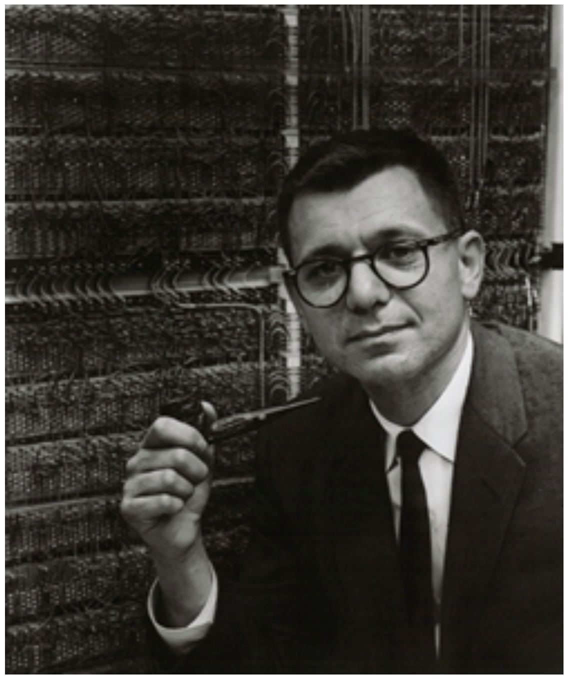

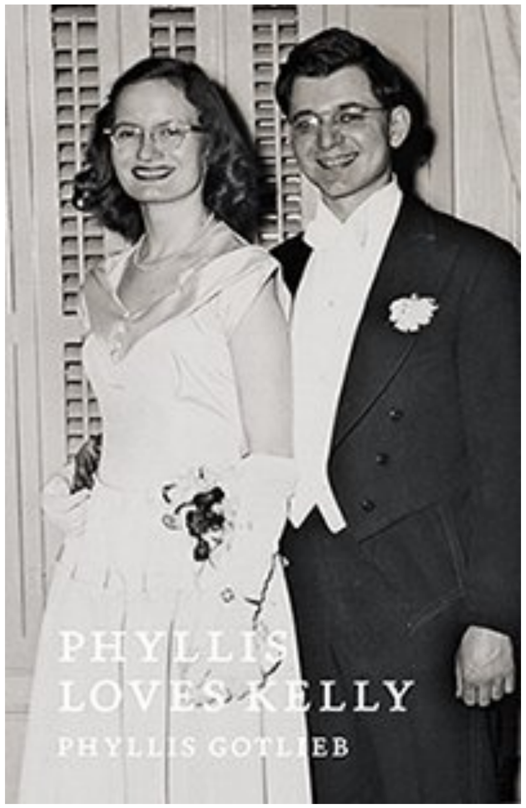

Kelly Gotlieb (1921-2016)

Computing pioneer, co-founder of the Department of Computer Science, University of Toronto

REST, NODE, Express

REST

- Representational State Transfer is an architectural style for using HTTP to provide resources over the web.
- Focus on roles and actions (HTTP verbs)
- Uniquely access resources through URLs
- Use GET, POST, PUT, DELETE
- Use a standard data format: HTML, XML, JSON
- Stateless protocol

Benefits

- Performance (lightweight)
- Scalability due to client-server separation
- Simplicity (nouns and verbs)
- Visibility of communication
- Portability (platform independent)
- Reliability at the system level

REST verbs

- GET : Read a specific resources (by identifier)
- PUT: Update or create a specific resource (by identifier
- DELETE: Removes a specific resource
- POST: Creates/updates a resource

- We need two basic URLS per resource:
    - One for a collection of items in the resource:
        - /collection/pokemon
    - One to select a particular resource:
        - /collection/pokemon/2

- You can even pass parameters:
    - /collection/pokemon/search?q=char&field=name

Best practices for RESTful API design

- Use logical URLs that are human-understandable and don't point to a particular file
- If dealing with a lot of data provide a paging mechanism
- Document everything and provide instructions
- Use POST (not GET) to make a change
- Provide multiple output data formats, e.g. JSON, XML, CSV, RSS, HTML
- Use authentication if your API allows change/deletion/adding

Example: https://developers.google.com/google-apps/calendar/v3/reference/

Server-side

- The server side of the web application processes HTTP requests, and outputs a combination of HTML, JSON, JavaScript to the client

- Node.js is an event-driven, I/O model-based runtime environment and library for developing server-side web apps using JS.

Threads vs Events

```
req = readRequest(socket);
reply = processRequest(req);
sendReply(socket, reply);
```

Implementation:

Thread switching (i.e. blocking) and scheduler

```
readRequest(socket,function(request) {
       processRequest(request,
             function (reply) {
               sendReply(socket, reply);
       });
});
```

Implementation:

Event queue processing

Event Queue

- Inner loop

```
while(true) {
  if(!eventQueue.notEmpty()) {
      eventQueue.pop().call();
  }
```

- Never wait/block in event handler
  - Example `readRequest (socket)`

    1. `launchReadRequest(socket); // returns immediately`
    2. When read finishes:
       `eventQueue.push(readDoneEventHandler)`

Node.js

- Use a JavaScript engine from a browser (Chrome's V8 engine)
  - Get the same JavaScript on browser and server
  - Don't need the DOM on the server
- Add events and an event queue
  - Everything runs as a call from event loop
- Make event interface to all OS operations
  - Wrap all OS blocking calls (file and network I/O)
  - Add some data handling support
- Add a proper module system
  - Each module gets its own scope

require is a Node module call

OS `read()` is synchronous but Node's `fs.readFile` is asynchronous

```
var fs = require(“fs”);

fs.readFile("smallFile", readDoneCallback); // Start read

function readDoneCallback(error, dataBuffer) {
 if (!error)  {
   console.log("smallFile contents”,
                     dataBuffer.toString());
 }
}
```

Node callback convention: First argument is JavaScript Error object

dataBuffer is a special Node Buffer object

Node Modules

- Import using require()
  - System module: require(“fs”); // Looks in node_module
  - From a file: require(“./mod.js”); // reads specified file
  - From a directory: require(“./myModule”); // reads myModule/index.js
- Modules have a private scope
  - Require returns what is assigned to module.exportss

Node Modules

- Many standard Node modules
- Huge library of modules (npm)
- We will use:
    - Express - “Fast, unopinionated, minimalist framework”
    - Mongoose - mongoldb object modelling

npm

`npm init`

will create a **`package.json`** file

Then for any new modules that you want to install, use

`npm install module --save`

```
{
  "name": "lab4",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "dependencies": {
    "body-parser": "^1.15.2",
    "ejs": "^2.5.2",
    "express": "^4.14.0",
    "express-validator": "^2.20.10"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test
specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "",
  "license": "ISC"
}
```

Git and installing modules

- You should not store generated files in git
  - wastes space, and can lead to confusion
- Add a `.gitignore` file to ignore files or directories that are generated.
- Example `.gitignore` file from lab4:

```
node_modules
npm-debug.log
```

Programming with Events/ Callbacks

- Key difference
  - Threads: Blocking/waiting is transparent
  - Events: Blocking/waiting requires callback
- Mental model
  - If code doesn’t block: same as thread programming
  - If code does block (or needs to block): Need to set up a callback
  - Often what was a return statement becomes a function

Example: Three step process

Threads

```
r1 = step1();

console.log('step1 done', r1);

r2 = step2(r1);

console.log('step2 done', r2);

r3 = step3(r2);

console.log('step3 done', r3);

console.log('All Done!');
```

Callbacks

```
step1(function(r1) {
    console.log('s1 done', r1);
    step2(r1, function (r2) {
        console.log('s2 done', r2);
        step3(r2, function (r3) {
            console.log(‘s3done',r3);
        });
    });
});
console.log(‘All Done!’); //WRONG
```

Listener/emitter pattern

- When programming with events a listener/emitter pattern is used.
- Listener - Function to be called when the event is signalled.
    - Same idea as DOM programming (addEventListener)
- Emitter - Signal that an event has occurred
    - Emit an event causes all the listener functions to be called

EventEmitter

```
var events = require('events');
var myEmitter = new events.EventsEmitter();
```

- Listen with on() and signal with emit()

```
myEmitter.on('myEvent', function(param1, param2) {
 console.log('myEvent occurred with ' + param1 +
            'and' + param2 + '!');
});
myEmitter.emit('myEvent', 'arg1', 'arg2');
```

- On emit call listeners are called synchronously and in the order the listeners were registered
- If no listener then `emit()` is a no op.

Typical EventEmitter patterns

- Have multiple different events for different state or actions

```
myEmitter.on('conditionA', doConditionA);
myEmitter.on('conditionB', doConditionB);
myEmitter.on('conditionC', doConditionC);
myEmitter.on('error', handleErrorCondition);
```

- Handling 'error' is important - Node exits if not caught!

```
myEmitter.emit('error', new Error('Ouch!'));
```

Thinking asynchronously

```
listen( "click", function handler(evt){
    setTimeout( function request(){
        ajax( “/url", function response(text){
            if (text == "hello") {
                handler();
            } else if (text == “world”) {
                request();
            }
        } );
    }, 500) ;
} );
```

Thinking asynchronously

```
listen( "click", handler );

function handler() {
    setTimeout( request, 500 );
}

function request(){
    ajax( “/url, response );
}

function response(text){
    if (text == "hello") {
        handler();
    } else if (text == "world") {
        request();
    }
}
```

Excerpt From: Kyle Simpson. “You Don’t Know JS: Async & Performance.” iBooks.

Order?

```
doA( function() {
    doB();
    doC( function() {
        doD();
    })
    doE();
});

doF();
```

Example where things can go mysteriously wrong

Track a sale. Analytics provided by third party…

```
analytics.trackPurchase( purchaseData,
                         function(){
    chargeCreditCard();
    displayThankyouPage();
} );
```

Turns out client was charged 5 times!

Excerpt From: Kyle Simpson. “You Don’t Know JS: Async & Performance.” iBooks.

Potential fix

```
var tracked = false;

analytics.trackPurchase( purchaseData, function(){
    if (!tracked) {
        tracked = true;
        chargeCreditCard();
        displayThankyouPage();
    }
} );
```

Excerpt From: Kyle Simpson. “You Don’t Know JS: Async & Performance.” iBooks.

Questions?

- What happens if they never call the callback? (If they can call it 5 times, why not 0?)

- How could the utility misbehave?
  - Call the callback too early
  - Call the callback too late
  - Call the callback too few or too many
  - Fail to pass along environment or parameters
  - Swallow errors/exceptions
  - …

Express.js

- Relatively thin layer on top of the base Node.js functionality
- What does a web sever implementor need?
  - Speak HTTP: Node's HTTP module does this
  - Routing: Map URLS to the web server function
  - Middleware support: Allow request processing layers to be added. Custom support for sessions, cookies, security, compression, etc.

```
var express = require(‘express’);

var expressApp = express();
```

- expressApp object has methods for:
  - Routing HTTP requests
  - ○ Rendering HTML (e.g. run a preprocessor like Jade templating engine)
  - ○ Configuring middleware and preprocessors

```
expressApp.get('/', function (httpRequest, httpResponse)
{
      httpResponse.send('hello world');
});
expressApp.listen(3000);
```

Express routing

- By HTTP method:

```
expressApp.get(urlPath, requestProcessFunction);
expressApp.post(urlPath, requestProcessFunction);
expressApp.put(urlPath, requestProcessFunction);
expressApp.delete(urlPath,
                  requestProcessFunction);
expressApp.all(urlPath, requestProcessFunction);
```

- Many others less frequently used methods
- urlPath can contain parameters (e.g. ‘/user/:user_id')

httpRequest object

```
expressApp.get('/user/:user_id',
              function (httpRequest, httpResponse) ...
```

- Object with large number of properties
- Middleware (like JSON body parser, session manager, etc.) can add properties

`request.params` - Object containing url route params (e.g. user_id)

`request.query` - Object containing query params (e.g. &foo=9 ⇒ {foo: '9'})

`request.body` - Object containing the parsed body

`request.get(field)` - Return the value of the specified HTTP header field

httpResponse object

```
expressApp.get('/user/:user_id',
                function (httpRequest, httpResponse) ...
```

- Object with a number of methods for setting HTTP response fields

`response.write(content)`	- Build up the response body with content
`response.status(code)`	- Set the HTTP status code of the reply
`response.set(prop, value)`	- Set the response header property to value
`response.end()`	- End the request by responding to it
`response.send(content)`	- Do a write and end