# **CSC309** *Programming on the Web*

## **week 6:** http, rest, node

Amir H. Chinaei, Spring 2017

Office Hours: M 3:45-5:45 BA4222

ahchinaei@cs.toronto.edu
*http://www.cs.toronto.edu/~ahchinaei/*

# review

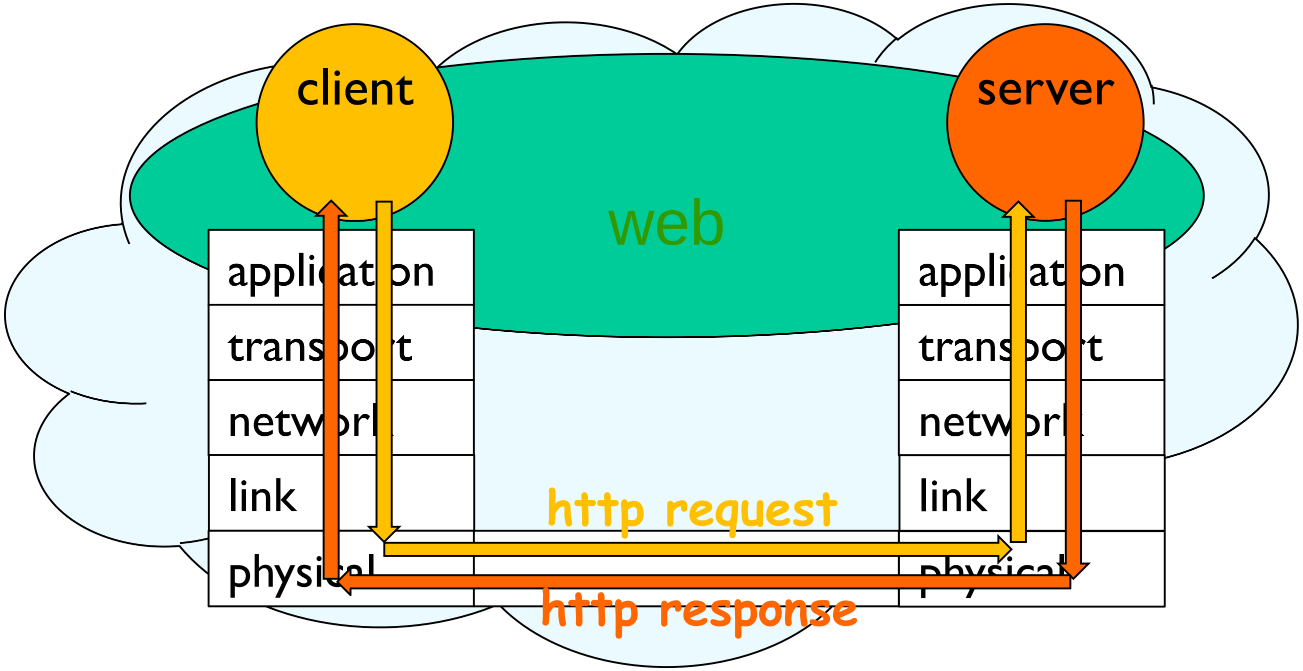
- **so far:**
    - **front-end**
        - **structure & semantic**, **appearance**, **behavior**
        - many design tips
    - **back-end**
        - **databases**
            - structured & semi-structured data
- **this week:**
    - front-end and back-end start **communication**
        - express, and sessions

# recall

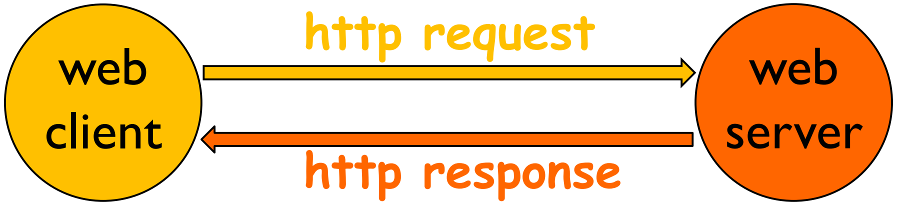
- web is an information space system—based on request & response—with the following features:
  - **HTML**: to describe (hypertext) documents/pages
  - **URL** : to uniquely locate a resource
  - **HTTP**: to describe how requests & responses operate.
  - **web server**: to respond to HTTP requests
  - **web browser**: to make HTTP requests from URLs and render/display the HTML document received

# recall

- client-server model
- communicate using **http** model
  - request-response



# http



- c&s establish a connection (details on csc358)
- client (e.g. browser) requests web content
- server responds with requested content
  - (if no error)
- c&s close the connection

- it's a stateless protocol

# static vs dynamic content

- **static**
  - content already stored in a resource
    - example: an html file, an image, etc.

dictionary1.com/content.html

- **dynamic**
  - content produced on-the-fly
    - example: an html file produced at run time by a program

dictionary2.com/search?word=content

both static and dynamic contents are stored in files (aka resources) before sending to the client .

# requests

- an http request consists of a *request line*
  - optionally followed by *request headers*
- *request line*

  <method> <uri> <version>

*request header*

<name>: <value>

- example:

```
GET   /    HTTP1.1
Host: utoronto.ca
```

- popular http methods:
  - GET get a static/dynamic resource from the server
  - POST get a dynamic resource from the server
  - PUT create a resource on server
  - DELETE delete a resource from server

# responses

- an http response consists of a *response line*
  - optionally followed by *response headers*
- *response line*

  <version> <status code> <status message>
- example:

```
HTTP1.1   302    Found
Content-Type: text/html
```

- some status codes:

| | |
|---|---|
| 200 | OK |
| 302 | Found |
| 403 | Forbidden |
| 404 | Not Found |

# rest

- motivation: an architectural style
- why it's called **rest**?
- "**re**presentational **s**tate **t**ransfer is intended to evoke an image of how a well-designed web application behaves:
  - a network of web pages (a virtual state-machine),
  - where the user progresses through an application by selecting links (state transitions),
  - resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

Roy Fielding

# examples

- to get all words in a dictionary web service, the client would request the following uri:
  - dictionary.com/words
- to get the word “content”, the client would request the following uri:
  - dictionary.com/word/content
- or,
  - dictionary.com/word/content?flavor=xml
- response

```
<?xml version=“1.0”?>
<word>
        <name>content</name>
        <definition>satisfied</definition>
        <example>She is content with her job</example>
</word>
```

# best practices

- identify all resources
- provide a uri for each resource
- logical uri is preferred
  - dictionary.com/word/content

  is preferred over
  - dictionary.com/word/content.html

  as it's transparent to client how the server generates it
- use nouns (not verbs) for uri
- do not change a resource by GET method
- use hypertext in your responses to facilitate next requests
- for complex queries, use a gradual unfolding approach
- provide documentation

# node.js

- backend runtime environment
  - javascript running on the server-side
  - **event-driven**
- **asynchronous io**
  - no-blocking
    - perform operation x asynchronously
    - continue other tasks
    - when op x is completed, send the response

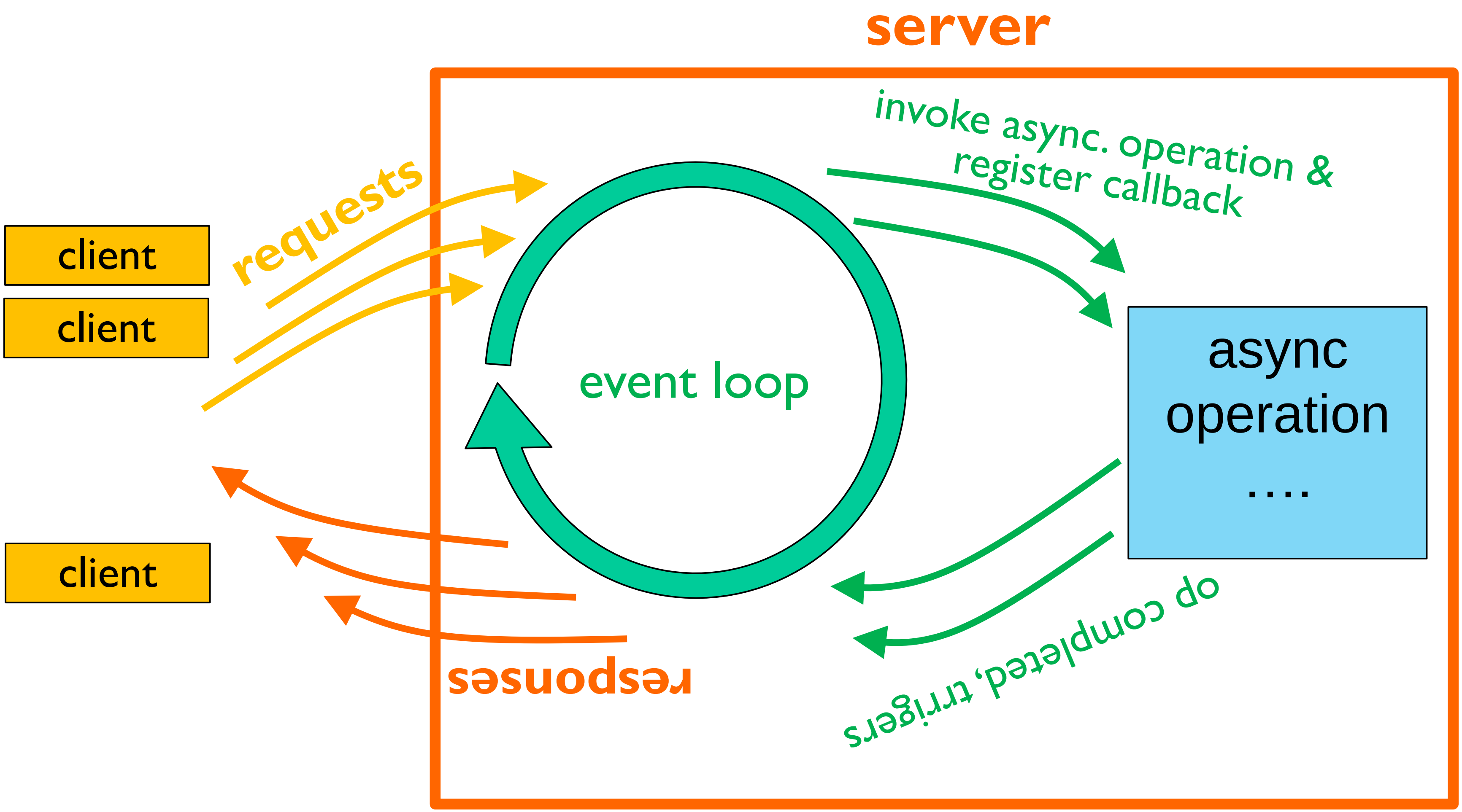
```
fs.readFile( “some.txt”, readCompletedCallback);
//do other tasks …

function readCompletdCallback( error, dataBuffer) {
   console.log(dataBuffer);
});
```

# non-blocking vs blocking

- example:
  - req1 at server: at time 1
  - req2 at server: at time 1
  - req1 initial process: 1 unit of time
  - req1 readFile: 5 units of time
  - req1 final process: 1 unit of time
- **non-blocking:**
  - `fs.readFile( "some.txt", readCompletedCallback);`
  - req2 initial process starts what time?
- **blocking:**
  - `fs.readFileSync( "some.txt", readCompletedCallback);`
  - req2 initial process starts at what time?

# event loop



server

client

client

client

requests

event loop

invoke async. operation &
register callback

async
operation
….

op completed, triggers

responses

# node.js example

- example:
  - create a server that listens to port 3000 on localhost
  - and to all requests, responds as

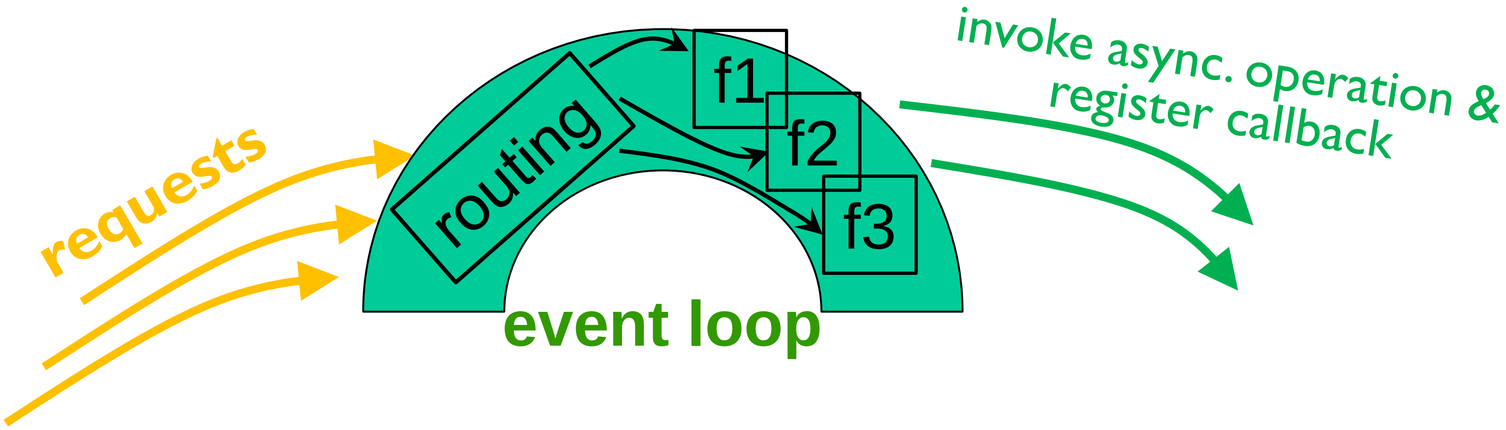
    **Hello World.**

    **My first webserver works fine**

```
var http = require('http');
http.createServer(function (req, res) {
          res.writeHead(200, {'Content-Type': 'text/plain'});
          res.end('Hello World.\nMy first webserver works fine :)');
         }).listen(3000, "127.0.0.1");
```

# express.js

- a thin layer on node.js
  - robust routing



# express.js

- example:
  - create a server that listens to port 3000
  - and to all requests, responds as

    **This is my first webserver using express ^-^**

```
var express = require('express');
var expressApp = express();

expressApp.get('/', function (httpRequest, httpResponse)
{
    httpResponse.send('This is my first webserver using express ^-^');
});
expressApp.listen(3000);
```