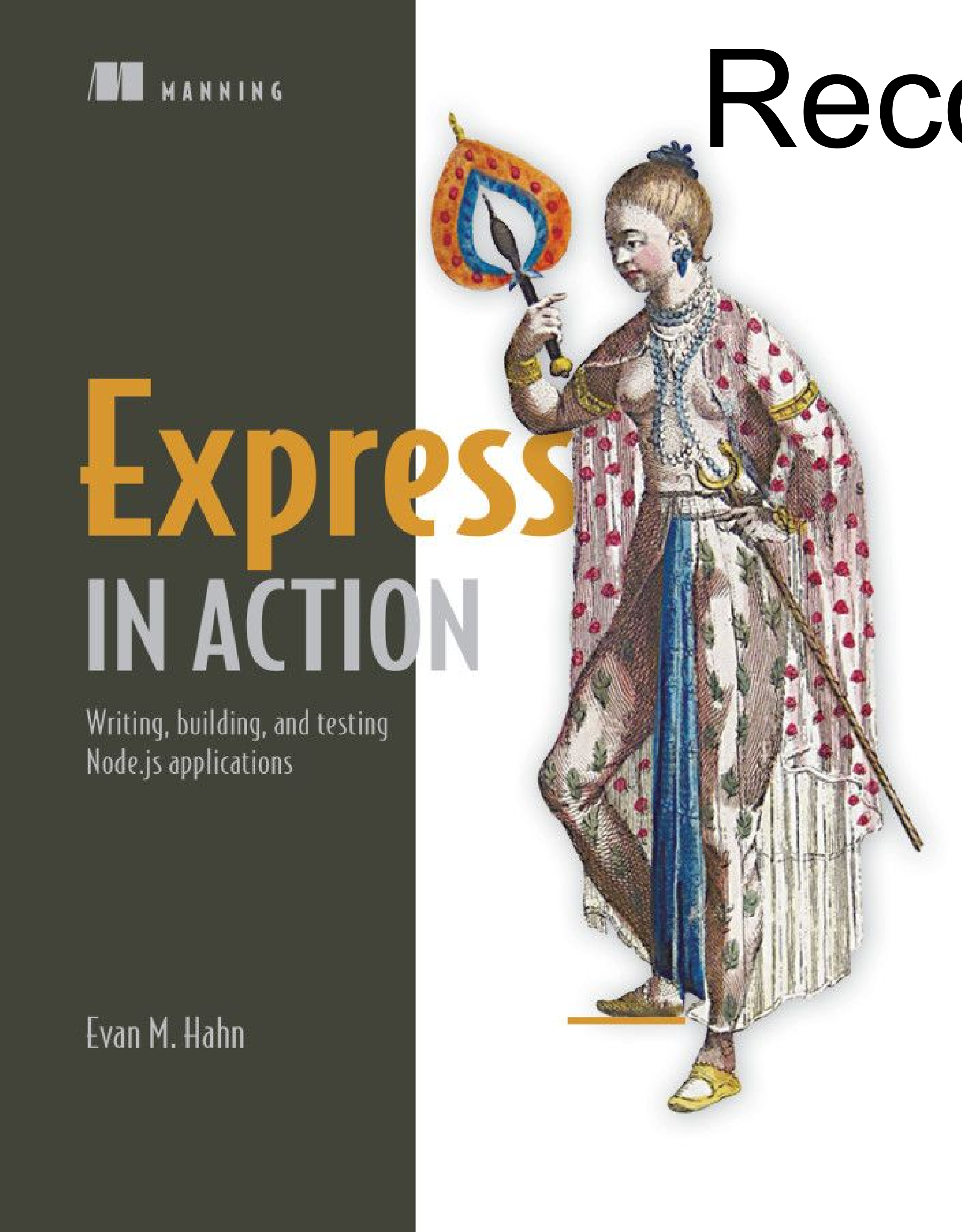
44-563: Unit 07

Developing Web Applications and Services

Includes

- Recommended text
- Nodemon (if not last week)
- Web services
- Middleware
- HTTP
- A03 & Git
- Workshop 3

Recommended



Express in Action: Writing, building, and testing Node.js applications

by Evan M. Hahn

https://www.manning.com/books/express-in-action

nodemon

- Use **nodemon** instead of node to run your app.
- nodemon **monitors and incorporates changes** (if you change a JavaScript file, you won't need to stop and restart node). :)

On Windows: use Open Command Window Here as Administrator

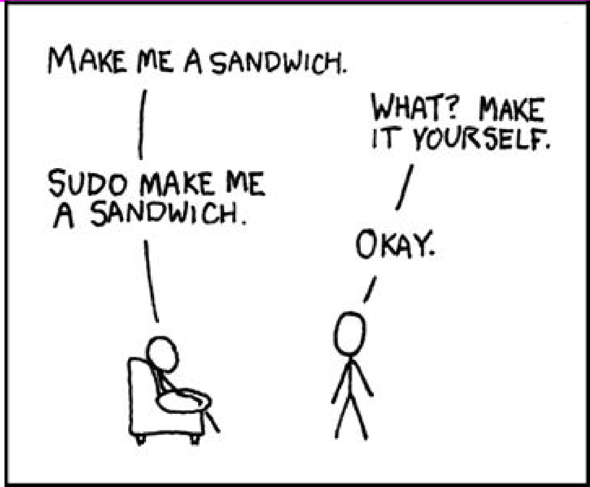
> npm install -g nodemon

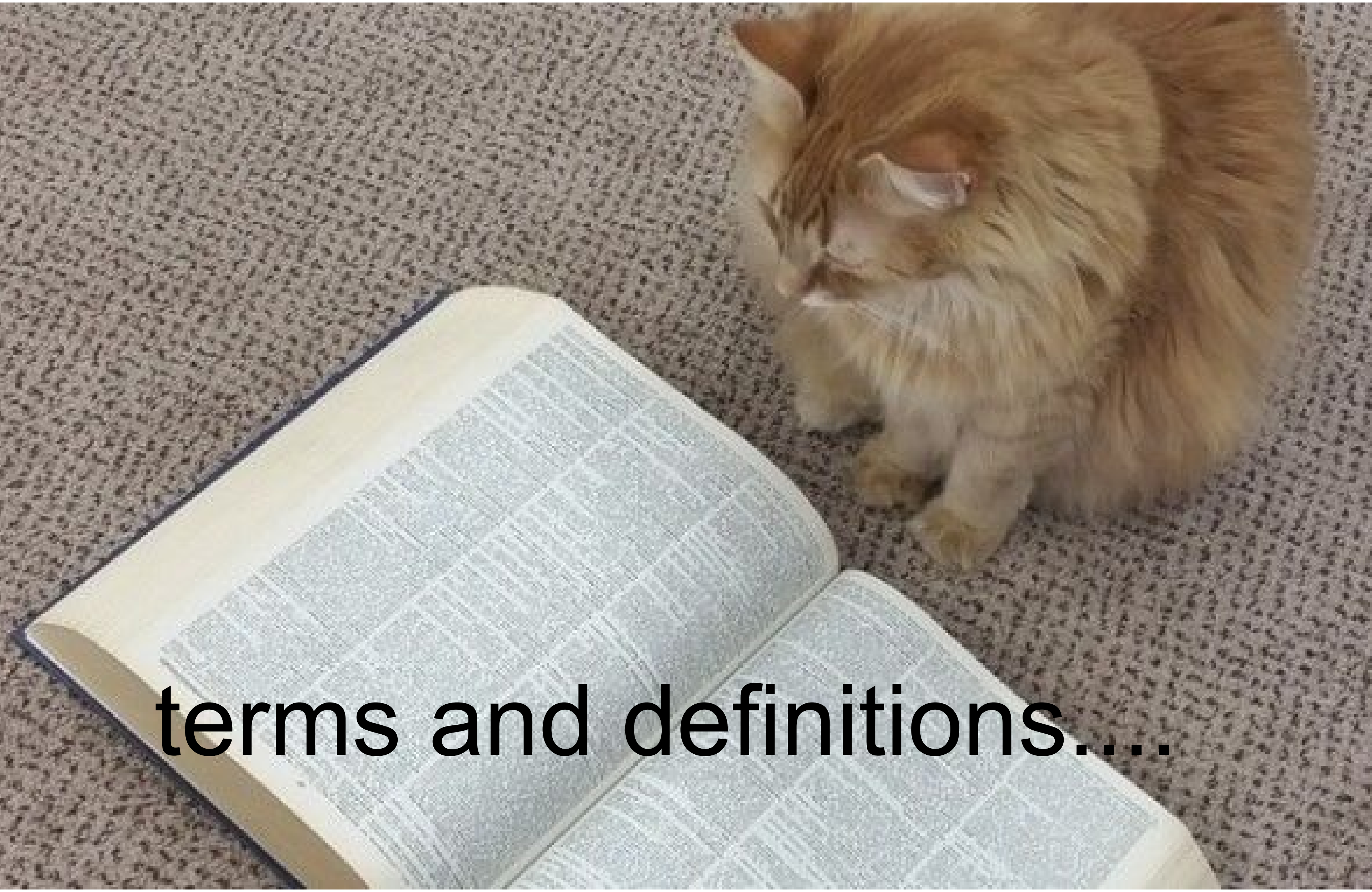
Mac/ Linux:

$ sudo npm install -g nodemon

Little bit of Linux:

- **sudo** means execute the following command as root.
- Use it only when the command comes from a trusted source.
- **-g** means install a package **globally**, not just locally in your module's folder
- nodemon is worth a global install





terms and definitions....

Web Services

Web server

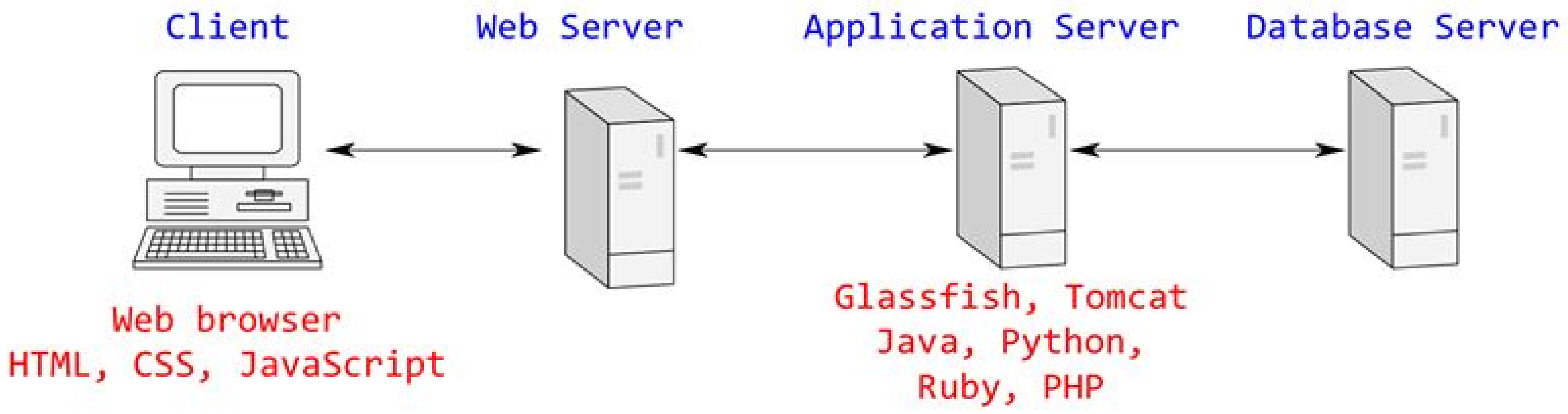
A **Web server** is a **computer system** that processes requests via HTTP to distribute information on the WWW.

Usage varies. The term may refer to:

- the whole **system** OR
- a specific **program** OR
- **hardware** - the dedicated computers and appliances that host the **Web servers**

Common servers in web apps

- **A web server** handles HTTP requests, serves web-based apps (servlets, JSP, ASP, etc)
- **An app server** handles business logic and communicates over various protocols (which may include HTTP).



Web service

A **Web service** is:

a **service** offered by an electronic device to another electronic device communicating via the WWW.

a software **system** designed to support interoperable **machine-to-machine** interaction over a network.[1]

By default, web services operate over port **80** or **443**. Alternate ports must be explicitly stated.

They typically transmit data (e.g.,JSON, XML) to be consumed by a client.

2 Web service classes

There are two major classes of Web services:

- **RESTful services**, in which the primary purpose of the service is to manipulate representations of **Web resources** using a **uniform** set of **stateless** operations. (They use standard HTTP operations (e.g., POST, GET, PUT, etc.), coupled with resources.)
- **Arbitrary services**, in which the service may expose an arbitrary or highly customized set of operations.[2]

The software system that requests data is called a ***service requester***, whereas the software system that handles the request and provides the data is called a ***service provider***.

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

Middleware

Middleware

Anything between the operating system kernel and user applications is considered **middleware**.

Middleware supports and simplifies complex distributed applications. It includes **web servers, application servers**, messaging and similar tools that support application development and delivery.

Middleware is especially integral to modern information technology based on XML, SOAP, Web services, and service-oriented architecture.

https://en.wikipedia.org/wiki/Middleware_(distributed_applications)

Middleware, Express Style

Middleware is used in Express apps. When any request comes in to Express(), **multiple functions** can be chained together to process the request/response. These functions form the middleware.

In Express, **app.use()** calls bind middleware to your application.

Each middleware function executes, then calls the next function. The order in which the methods appear dictates the sequence.

In this example, we use one middleware function, myLogger, before the app.get() method specifies the routing by matching a get call to '/' to a provided function.

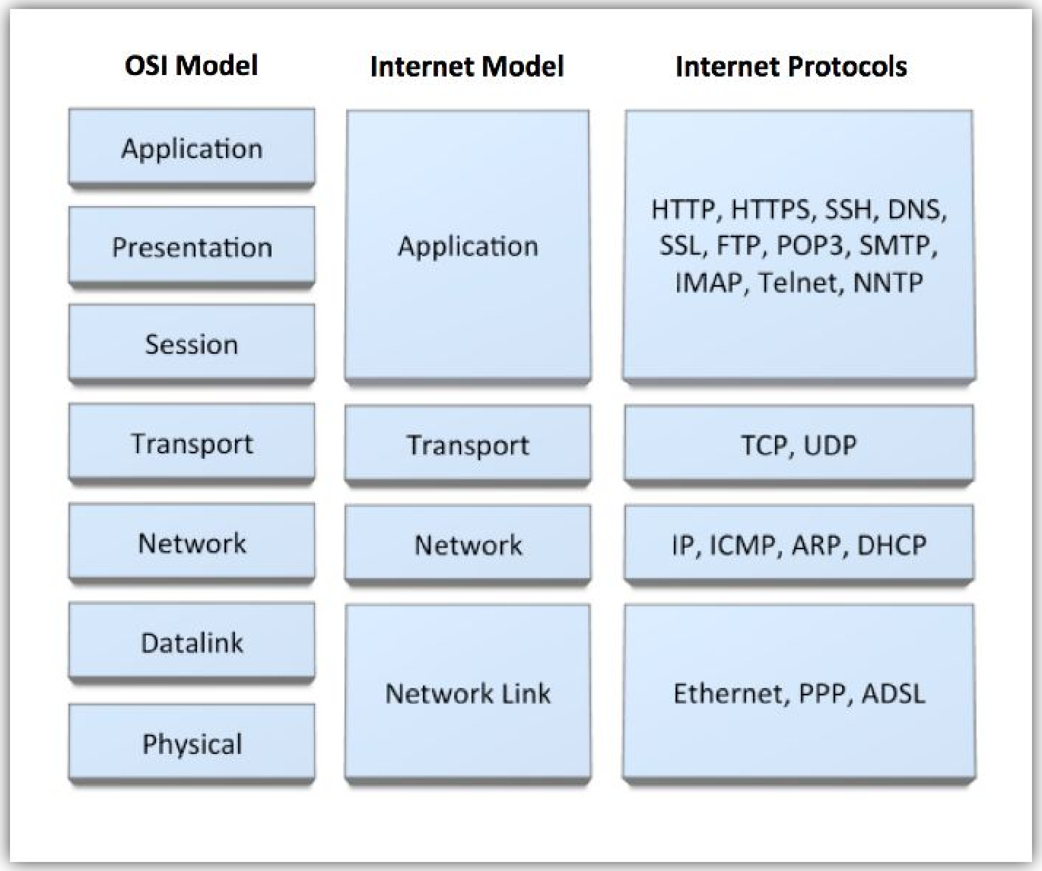
```
var express = require('express')
var app = express()

var myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}

app.use(myLogger)

app.get('/', function (req, res) {
  res.send('Hello World!')
})

app.listen(3000)
```

HTTP
Messages & Methods

What is a protocol?

- A **protocol** is a standard **procedure** for *defining and regulating communication*.
- There are many protocols, HTTP is just one, WebSockets is another.
- HTTP is the **set of rules** governing the *format and content* of the conversation between a Web client and server.
- The original rfc describing the HTTP protocol makes for interesting reading, as does the more recent version.

HTTP is a protocol in the Application Layer



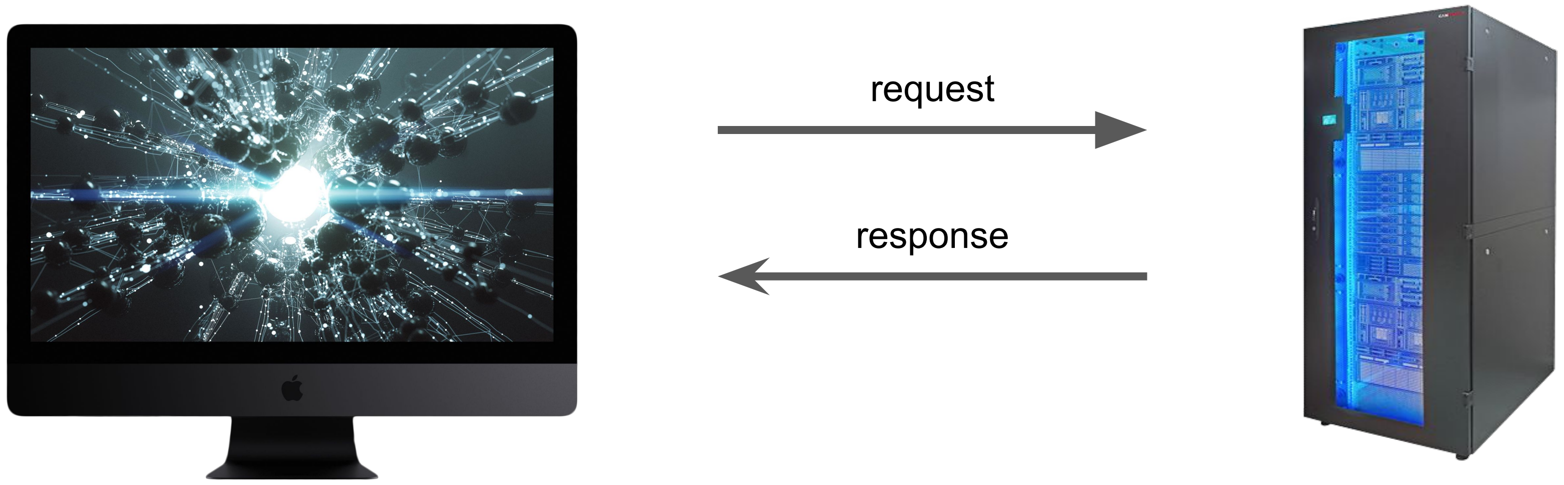
http://vichargrave.com/wp-content/uploads/2013/01/Network-Stack-Models1.png

- HTTP is an **abstraction layer** used in both TCP/IP and OSI
- protocol to exchange **hypertext** (text with links)
- Specifically, HTTP is an application protocol for *distributed collaborative hypermedia information systems*

HTTP Messages:

HTTP involves exchanging **messages** between client and server.

The client sends a **request** message, the server responds with a **response** message.



HTTP Message Parts:

Each message, whether a **request** or a **response**, has three parts:

a. The request or response **line**
b. The rest of the **header** section
c. The **body** of the message

HTTP Methods

HTTP defines **methods** (sometimes referred to as **verbs**) to indicate the desired action to be performed on the identified resource.

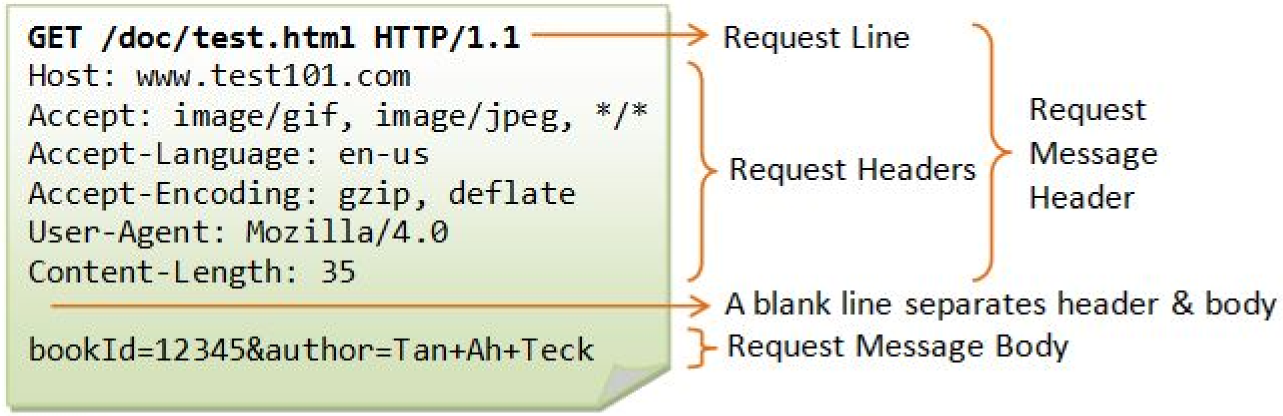
The **four main ones** we use are shown below (there are more).

Every HTTP request starts with one of these methods.

Method	Description
GET	Requests data from a specified resource
POST	Submits data to be processed to a specified resource
PUT	Uploads a representation of the specified URI
DELETE	Deletes the specified resource

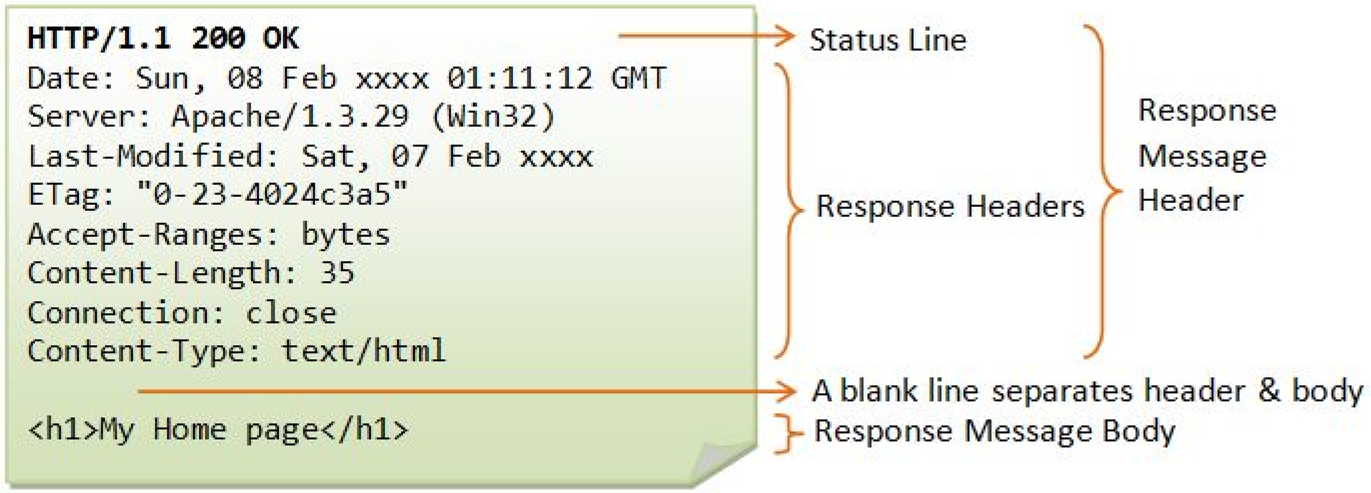
HTTP Message Examples

Request:



```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

Response:



```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:11:12 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 07 Feb xxxx
ETag: "0-23-4024c3a5"
Accept-Ranges: bytes
Content-Length: 35
Connection: close
Content-Type: text/html

<h1>My Home page</h1>
```

https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/images/HTTP_RequestMessageExample.png
https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/images/HTTP_ResponseMessageExample.png

HTTP Req/Res Line

- Every request line and response line includes the HTTP **version**.
- The response line includes a **status code** & short **description** Example: *HTTP/1.1 404 Not Found*
- Status Codes
  - 100-199 Informational
  - 200-299 Request was successful
  - 300-399 Request was redirected
  - 400-499 Request failed (e.g. 404)
  - 500-599 Server error occurred

HTTP Message Parts

1. **Headers**
2. **Request Body**
3. **Response Body**

Match the message parts :)

Can you match the message part to what it contains?

A. Includes instructions, e.g. the type of response, authorization

B. Includes data sent with the request, e.g. for POST

C. Includes the resource information requested by the client e.g. with GET

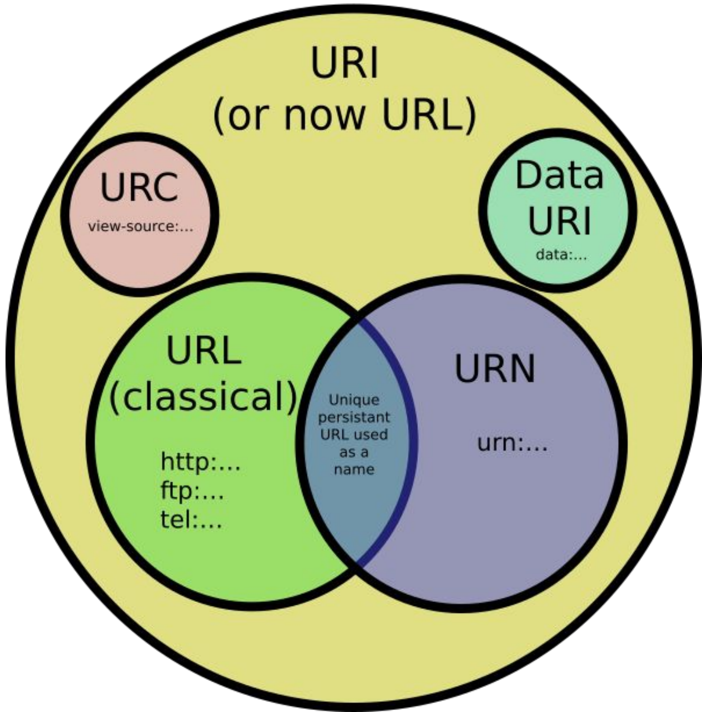
URI/URL

1. Uniform Resource Identifier (URI) identifies
2. Uniform Resource Locator (URL) identifies & locates
3. Current W3C: use these interchangeably, URI is preferred
4. May include **query parameters** (or not)
5. Globally **unique** and **persistent** even when the resource ceases to exist or becomes unavailable.
6. Prefixes include http:, ftp:, tel:, urn:, data:
7. **We will use URI & URL interchangeably,** and wait for the industry to figure out what it wants to do.

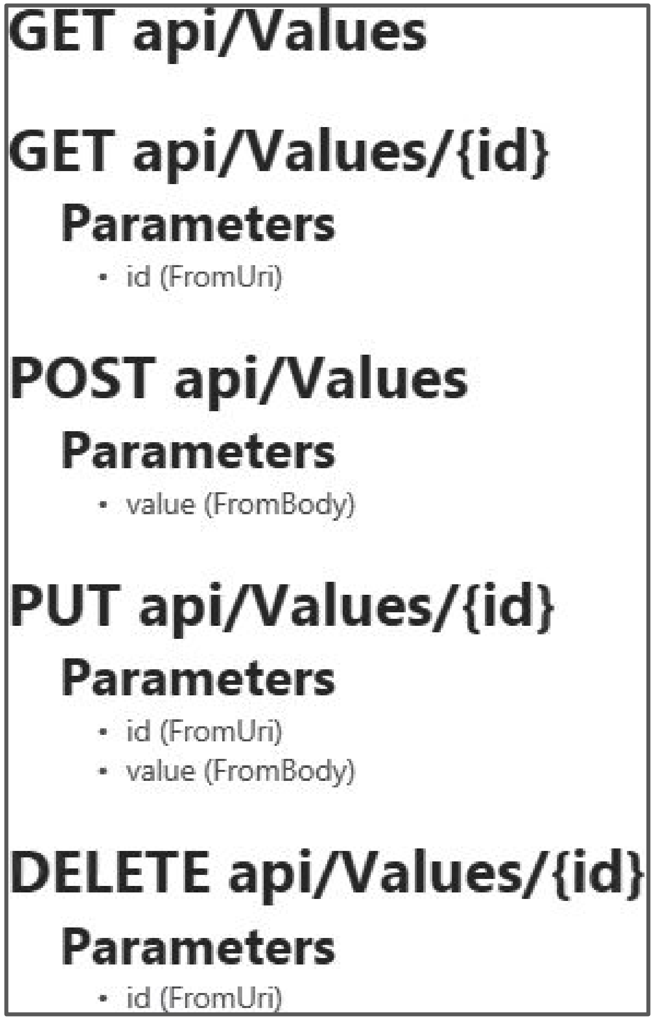
A REST-ful URI is a URI that identifies a ***domain resource*** *(like a book or a shelf or a book loan in a library application) rather than an application resource (like a web page or a form in your application or website).*

- *http://blog.2partsmagic.com/restful-uri-design/*

Venn diagram of URIs as defined by the W3C



Example of a Web API



GET api/Values

GET api/Values/{id}

Parameters

- id (FromUri)

POST api/Values

Parameters

- value (FromBody)

PUT api/Values/{id}

Parameters

- id (FromUri)
- value (FromBody)

DELETE api/Values/{id}

Parameters

- id (FromUri)

Sometimes confused:

Put = update

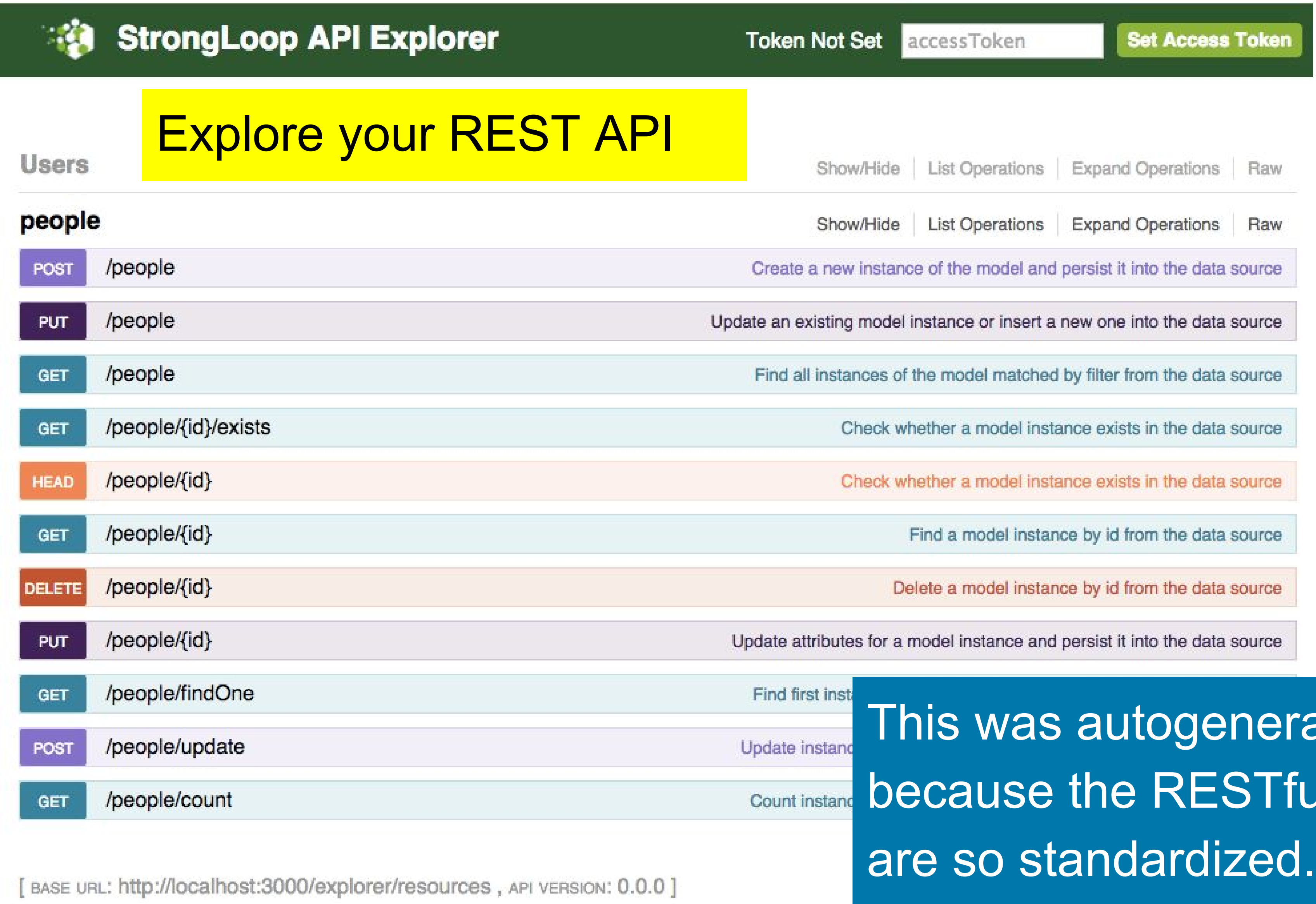
Post = **insert**

https://everyoneneedsanalgonquin.files.wordpress.com/2010/03/erpost.jpg

For every resource there will be at least these 5 calls. The api represents the host and port; Values represents the particular resource being accessed.

http://blogs.msdn.com/b/martinkearn/archive/2015/01/05/introduction-to-rest-and-net-web-api.aspx
https://en.wikipedia.org/wiki/Web_API

An Autogenerated API



Explore your REST API

people Show/Hide | List Operations | Expand Operations | Raw

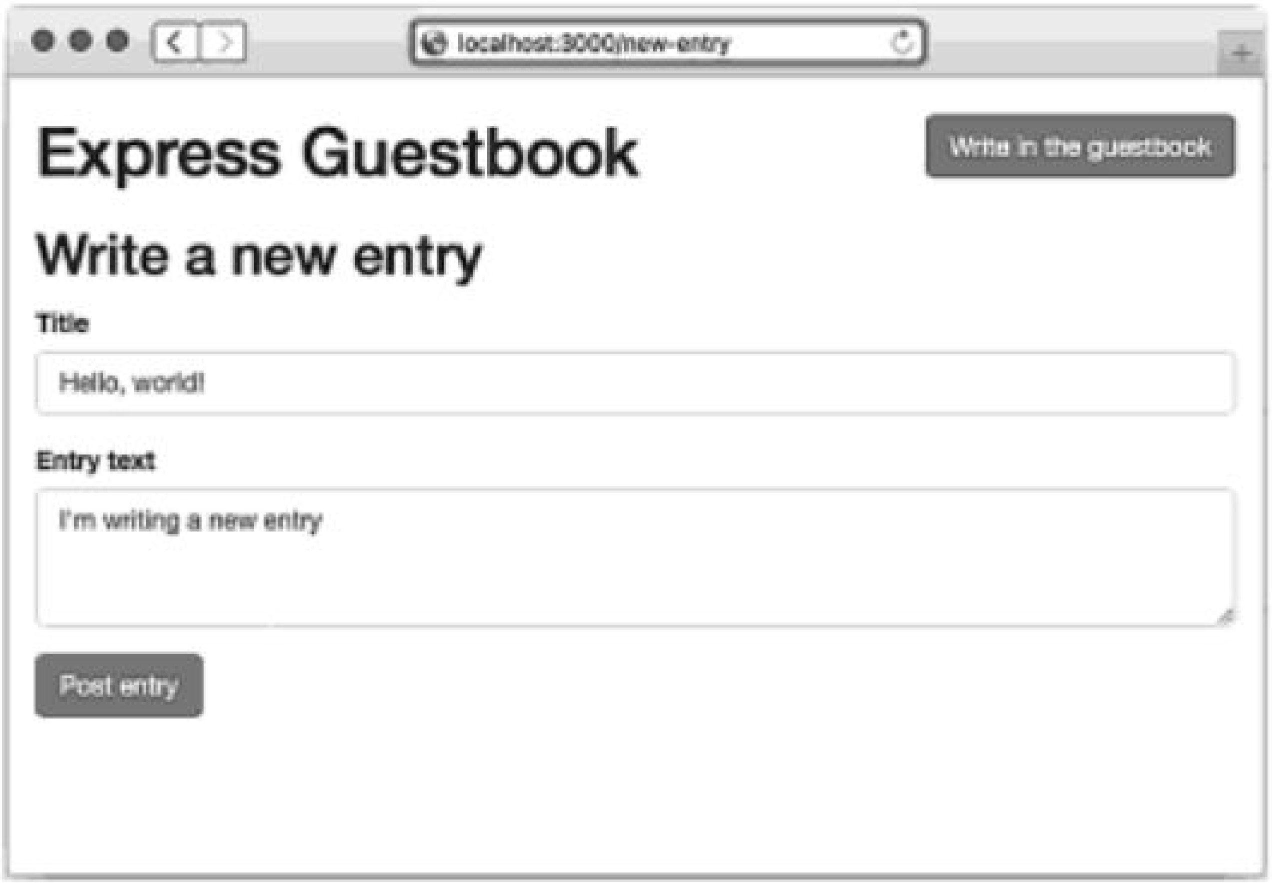
POST	/people	Create a new instance of the model and persist it into the data source
PUT	/people	Update an existing model instance or insert a new one into the data source
GET	/people	Find all instances of the model matched by filter from the data source
GET	/people/{id}/exists	Check whether a model instance exists in the data source
HEAD	/people/{id}	Check whether a model instance exists in the data source
GET	/people/{id}	Find a model instance by id from the data source
DELETE	/people/{id}	Delete a model instance by id from the data source
PUT	/people/{id}	Update attributes for a model instance and persist it into the data source
GET	/people/findOne	Find first inst
POST	/people/update	Update instanc
GET	/people/count	Count instanc

[BASE URL: http://localhost:3000/explorer/resources , API VERSION: 0.0.0]

This was autogenerated, because the RESTful APIs are so standardized.

Let's make a Guest Book to add to our site

Copyrighted Material


localhost:3000/new-entry

Express Guestbook

Write a new entry

Title

Hello, world!

Entry text

I'm writing a new entry

Figure 3.5 **The page to write a new entry in the guestbook**

https://www.amazon.com/Express-Action-Writing-building-applications/dp/1617292427

Let's Design It

Requirements:

1. Users can **write** new entries.
2. Users can **browse** others’ entries.

Requirements should always be **specific** and **numbered**.

Let's Design It

Two pages:

- A **guestbook page** that lists entries.
- A **new page** to add a new entry.

Also:

- Log all requests.
- Provide a 404 page.
- Use Express to **route** URL requests (home, add new, post)
- Use Express & EJS to **render pages**.

What is a 404 page?

EJS templates

- Design principle: *"Write once, use many (times)".*
- **Avoid duplicating** HTML.
- EJS stands for <% Embedded JavaScript %>
- JavaScript between **<% %>** is **executed**.
- JavaScript between **<%= %> adds HTML** to the result.
- In many multi-page apps, we have reusable page parts, e.g., header, footer.
- .ejs file extension

```
<% include header %>
<h2>404! Page not found.</h2>
<% include footer %>
```

<% include header %>

causes header.ejs to be included into the html at that location.

Let's Design It

For the server, we'll need:

- ❑ **package.json** (dependencies & meta information)
  - **Express** since it's a web app
  - **Morgan** logger middleware to **log all** HTTP requests
  - **Body-parser** middleware to **parse** incoming HTTP request bodies
  - **Ejs** (Embedded JavaScript) client-side view templating engine

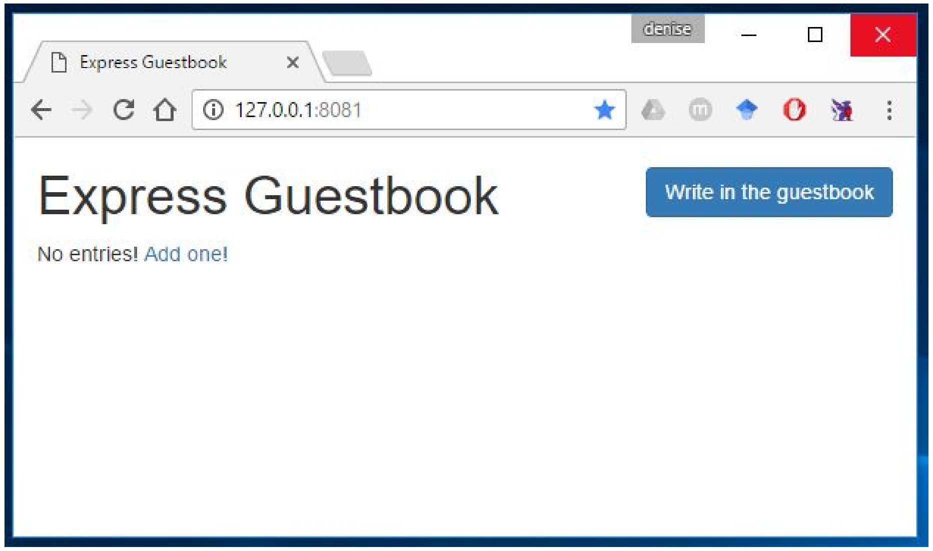
- ❑ **gbapp.js** (our server-side app)
  - Manages entries
  - GET home page request that comes to "/"
  - GET new-entry page request that comes to "/new-entry"
  - POST a new-entry (insert)
  - Show 404 if page not found

Let's Design It

For the client, create a **views** folder with 5 views:

- ❑ 404.ejs (page not found)
- ❑ footer.ejs (used by both index & new-entry)
- ❑ header.ejs (used by both index & new-entry)
- ❑ index.ejs (view for default or home page)
- ❑ new-entry.ejs (view for creating an entry)

EJS views are a **server-side** construct. The view uses a combination of **HTML and code** to dynamically generate content that will be rendered in the browser.



Express Guestbook

No entries! Add one!

Let's Build It (M07)

Create C:\44563\m07 folder. In this root folder,

1. Create empty **.gitignore** (hint: use **git bash** touch)
2. Create empty **README.md**.
3. Create empty **package.json**.
4. Create empty **gbapp.js**.
5. Create a **views** subfolder. In here, create:
6. 404.ejs
7. footer.ejs
8. header.ejs
9. index.ejs
10. new-entry.ejs

1. Create the files.
2. Right-click folder & *Open with Code.*

.gitignore (always)

```
node_modules
```

Why do we not commit our dependencies?

README.md (always)

~~~
# M07 Guestbook Example

A simple guest book using Node, Express, BootStrap, EJS

## How to use

Open a command window in your c:\44563\m07 folder.

Run npm install to install all the dependencies in the package.json file.

Run node gbapp.js to start the server.  (Hit CTRL-C to stop.)

```
> npm install
> node gbapp.js
```

Point your browser to `http://localhost:8081`.
~~~

Check your Markdown online at http://dillinger.io/

package.json

```
{   "name": "M07",
    "version": "0.0.1",
    "description": "simple guestbook app",
    "main": "gbapp.js",
    "dependencies": {
        "express": "latest",
        "morgan": "latest",
        "body-parser": "latest",
        "ejs": "latest"
    },
    "author": "Denise Case",
    "homepage": "https://bitbucket.org/professorcase/m07",
    "repository": {
        "type": "git",
        "url": "https://bitbucket.org/professorcase/m07"
    },
     "license": "Apache-2.0"
}
```

- What is the main server program?
- What modules are included?
- Customize this file to reflect your information.

gbapp.js

Start with a basic shell

```
var path = require("path")
var express = require("express")
var logger = require("morgan")
var bodyParser = require("body-parser")

var app = express()  // make express app
var server = require('http').createServer(app) // inject app into the server

// set up the view engine
// manage our entries
// set up the logger
// GETS
// POSTS
// 404

// Listen for an application request on port 8081
server.listen(8081, function () {
  console.log('Guestbook app listening on http://127.0.0.1:8081/')
})
```

https://www.amazon.com/Express-Action-Writing-building-applications/dp/1617292427

views/404.ejs

```
<% include header %>
<h2>404! Page not found.</h2>
<% include footer %>
```

This template **view** will be combined and rendered by the EJS view engine.

https://www.manning.com/books/express-in-action

views/footer.ejs

```
</body>
</html>
```

Boring footer.
Add your name and copyright.. Maybe some style.

views/header.ejs

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>M07 Guestbook</title>
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/
bootstrap.min.css">
</head>
<body class="container">
<h1>
Express Guestbook
<a href="/new-entry" class="btn btn-primary pull-right">
Write in the guestbook
</a>
</h1>
```

Include css links in your header. Include header in every full page view.

Google to get current links for BootStrap CDN and jQuery CDN

https://www.manning.com/books/express-in-action

Let's Build It (M07)

C:\44563\M07 folder

- ❑ Created **.gitignore**
- ❑ Created **README.md**.
- ❑ Created **package.json**.
- ❑ Created **gbapp.js**.
- ❑ Created views (to use later)

Open cmd window in your app folder:

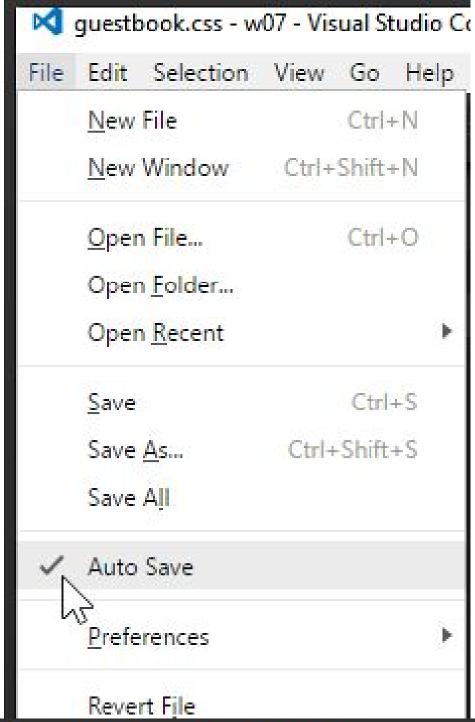
```
> npm install
> node gbapp.js
```

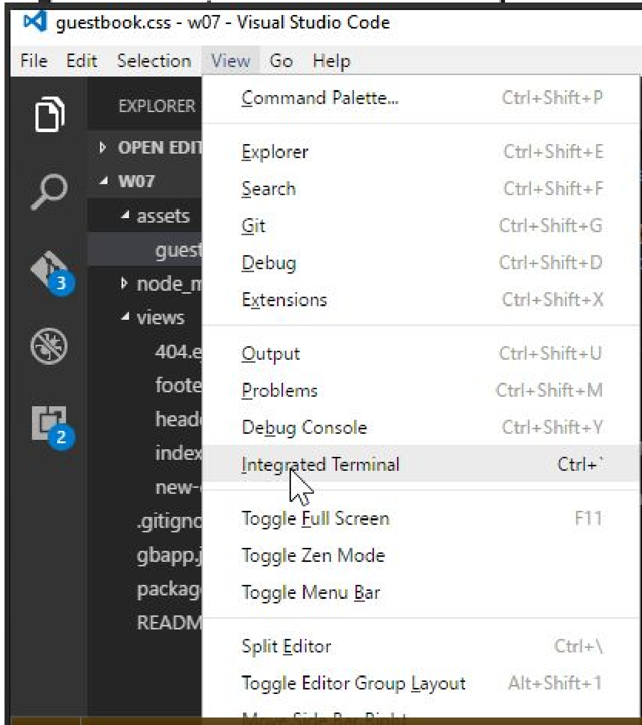
1. What command do we use to install the dependencies into our new app?
2. What file lists the dependencies?
3. What command do we use to start our app?

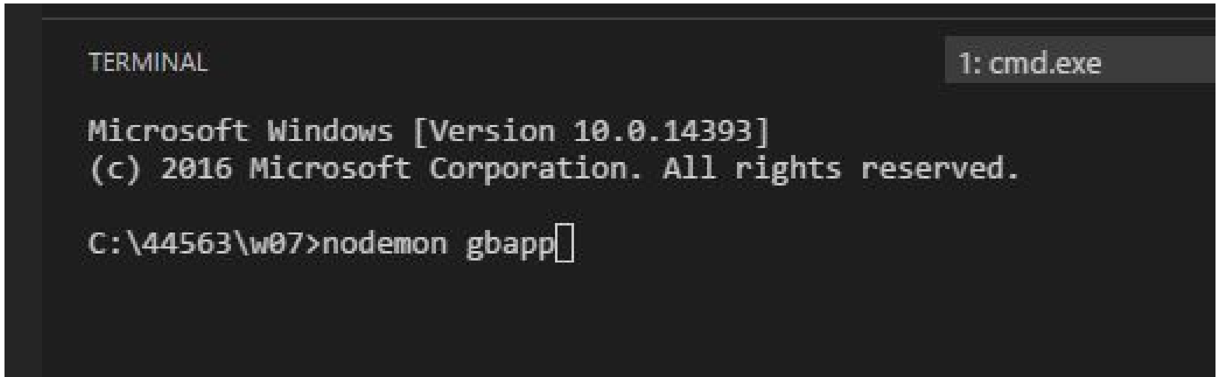
VS Code ProTips

Right-click on folder in File Explorer (e.g. C:\44563\M07)

- ❑ Select **"Open with Code"**.
- ❑ Turn on **File / Auto Save**.
- ❑ Open **View / Integrated Terminal.**
- ❑ Type > **nodemon gbapp.js**





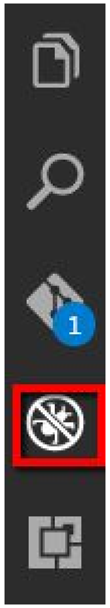


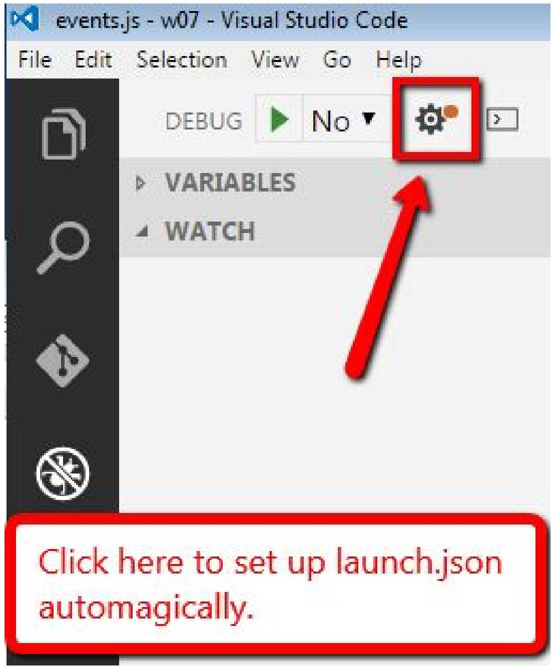
```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\44563\w07>nodemon gbapp
```

VS Code Debugger

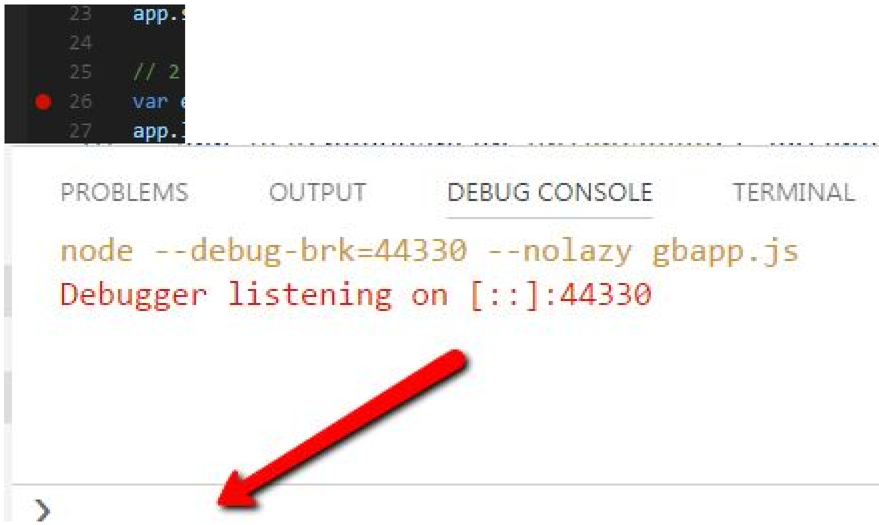
Right-click on folder in File Explorer (e.g. C:\44563\M07)

- ❑ Select **"Open with Code"**.
- ❑ In VS Code, click the **Debug** Icon
- ❑ In Debug, click Settings icon.
- ❑ It automagically creates launch.json.
- ❑ Set breakpoints (left of line #'s)
- ❑ Click the green arrow to run.
- ❑ In Debug console at the bottom, enter variables to inspect.





Click here to set up launch.json automagically.



```
app.
// 2
var e
app.
```

```
node --debug-brk=44330 --nolazy gbapp.js
Debugger listening on [::]:44330
```

Let's finish our Guest Book

Let's Finish It (w07)

Create C:\44563\w07 folder.

1. Copy your content.
2. Update the README.md to w07.
3. Review the design.
4. Looks like we still need **two more views**.
5. Then we'll finish **gbapp.js**

README.md:

- Typically capitalized
- Google **markdown** to format your file.

We'll finish laying out the views.
And then complete the server-side app.

views/index.ejs

```
<% include header %>
<% if (entries.length) { %>
	<% entries.forEach(function(entry) { %>
	<div class="panel panel-default">
		<div class="panel-heading">
			<div class="text-muted pull-right">
			<%= entry.published %></div>
			<%= entry.title %></div>
		<div class="panel-body">
		<%= entry.body %>
		</div>
	</div>
	<% }) %>
<% } else { %>
	No entries! <a href="/new-entry">Add one!</a>
<% } %>
<% include footer %>
```

Where did entries come from?

1. Follow good engineering practices and "***Don't Repeat Yourself***":
   - Reuse/include a shared header you write once.
   - Include a shared footer you write once.

2. Display in a loop, based on number of entries.

https://www.manning.com/books/express-in-action

views/new-entry.ejs

```
<% include header %>
<h2>Write a new entry</h2>
<form method="post" role="form">
      <div class="form-group">
      <label for="title">Title</label>
      <input type="text" class="form-control" id="title"
        name="title" placeholder="Entry title" required></div>

      <div class="form-group">
      <label for="content">Entry text</label>
      <textarea class="form-control" id="body" name="body"
      placeholder="Love Express! It’s a great tool for
      building websites." rows="3" required></textarea></div>

      <div class="form-group">
      <input type="submit" value="Post entry" class="btn btn-primary"></div>
</form>
<% include footer %>
```

New-entry needs a **heading** and a **form** with 3 parts.

Copyrighted Material
Summary

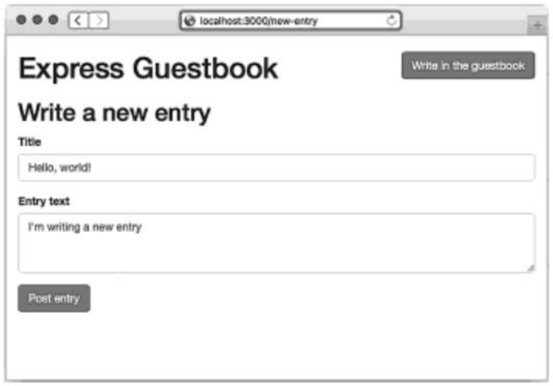


Figure 3.5 The page to write a new entry in the guestbook

https://www.manning.com/books/express-in-action

gbapp.js

```
var path = require("path")
var express = require("express")
var logger = require("morgan")
var bodyParser = require("body-parser") // simplifies access to request body

var app = express()  // make express app
var http = require('http').Server(app)  // inject app into the server

// 1 set up the view engine
// 2 manage our entries
// 3 set up the logger
// 4 handle valid GET requests
// 5 handle valid POST request
// 6 respond with 404 if a bad URI is requested

// Listen for an application request on port 8081
http.listen(8081, function () {
  console.log('Guestbook app listening on http://127.0.0.1:8081/')
})
```

https://www.manning.com/books/express-in-action

gbapp.js

```
// 1 set up the view engine
app.set("views", path.resolve(__dirname, "views")) // path to views
app.set("view engine", "ejs") // specify our view engine

// 2 create an array to manage our entries
var entries = []
app.locals.entries = entries // now entries can be accessed in .ejs files

// 3 set up an http request logger to log every request automagically
app.use(logger("dev"))     // app.use() establishes middleware functions
app.use(bodyParser.urlencoded({ extended: false }))
```

1. __dirname is where the local module resides; path.resolve() joins them to create an entire path.
2. You can use app.get() & app.set() to get and set variables -- since app is a JS object you can add variables as you wish -- but there are some specific variable/values pairs that are used by Express, including ‘views’ and ‘views engine’
3. app.locals is a convenient place to store values that will persist as long as the app is running.

https://www.manning.com/books/express-in-action

gbapp.js

```
// 4 handle http GET requests (default & /new-entry)
app.get("/", function (request, response) {
  response.render("index")
})
app.get("/new-entry", function (request, response) {
  response.render("new-entry")
})
```

Set up routing to GET our two pages:

1) /
2) /new-entry

https://www.manning.com/books/express-in-action

gbapp.js

```
// 5 handle an http POST request to the new-entry URI
app.post("/new-entry", function (request, response) {
  if (!request.body.title || !request.body.body) {
    response.status(400).send("Entries must have a title and a body.")
    return
  }
  entries.push({  // store it
    title: request.body.title,
    content: request.body.body,
    published: new Date()
  })
  response.redirect("/")  // where to go next? Let's go to the home page :)
})
```

Set up the routing to POST a new entry.

https://www.manning.com/books/express-in-action

gbapp.js

Configure our 404 page

```
// if we get a 404 status, render our 404.ejs view
app.use(function (request, response) {
  response.status(404).render("404")
})

// Listen for an application request on port 8081 & notify the developer
//http.listen(8081, function () {
 // console.log('Guestbook app listening on http://127.0.0.1:8081/')
//})
```

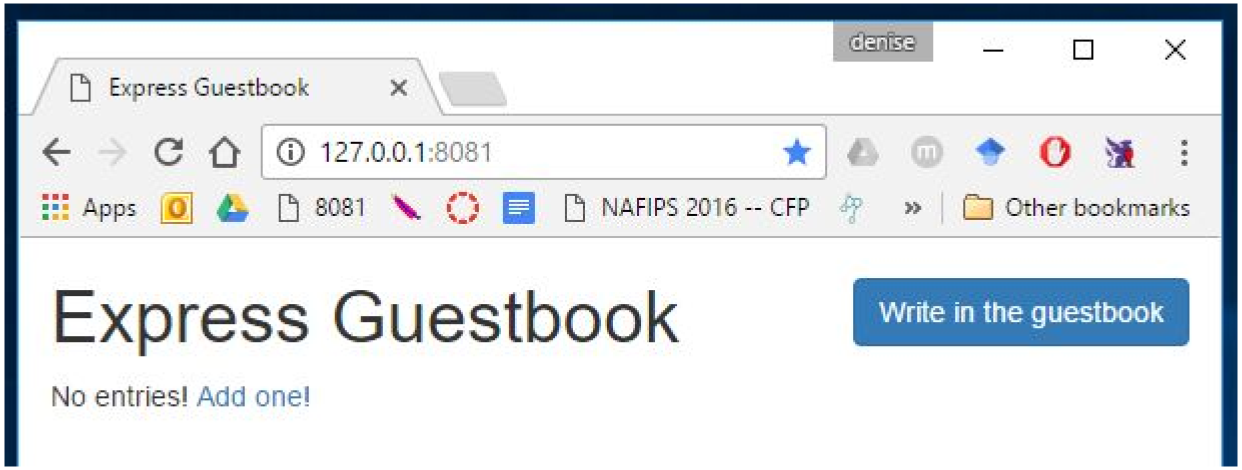
Let's Run It (W07)

Go to C:\44563\w07 folder

Right-click & Open command window here as administrator.

```
> npm install
> node gbapp
```

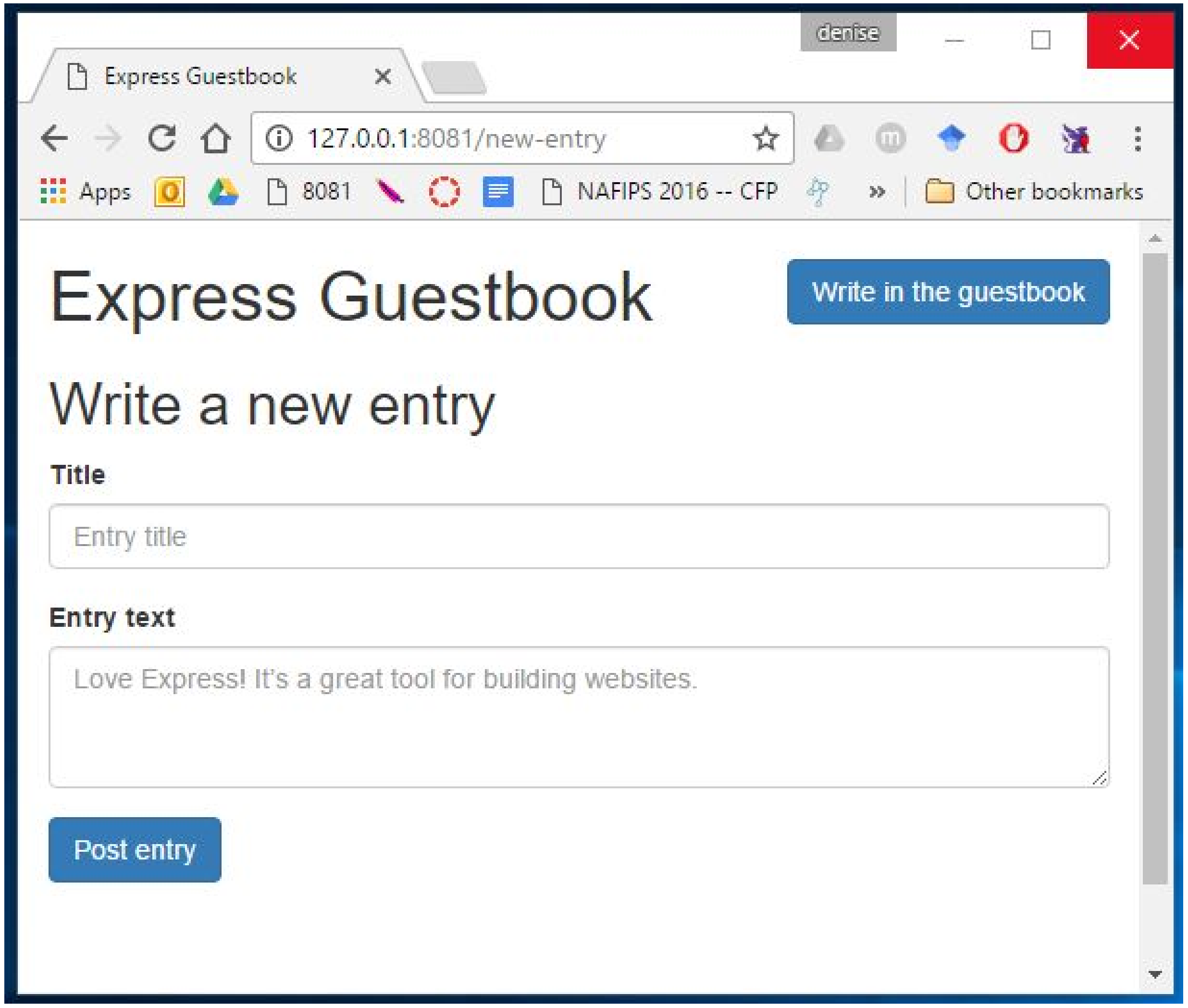
1. How are we **storing** the entries?
2. Is that scalable? Could we become the next FB?



Express Guestbook

No entries! Add one!

Let's Run It (W07)



denise

Express Guestbook

127.0.0.1:8081/new-entry

Apps 8081 NAFIPS 2016 -- CFP » Other bookmarks

Express Guestbook

Write in the guestbook

Write a new entry

Title

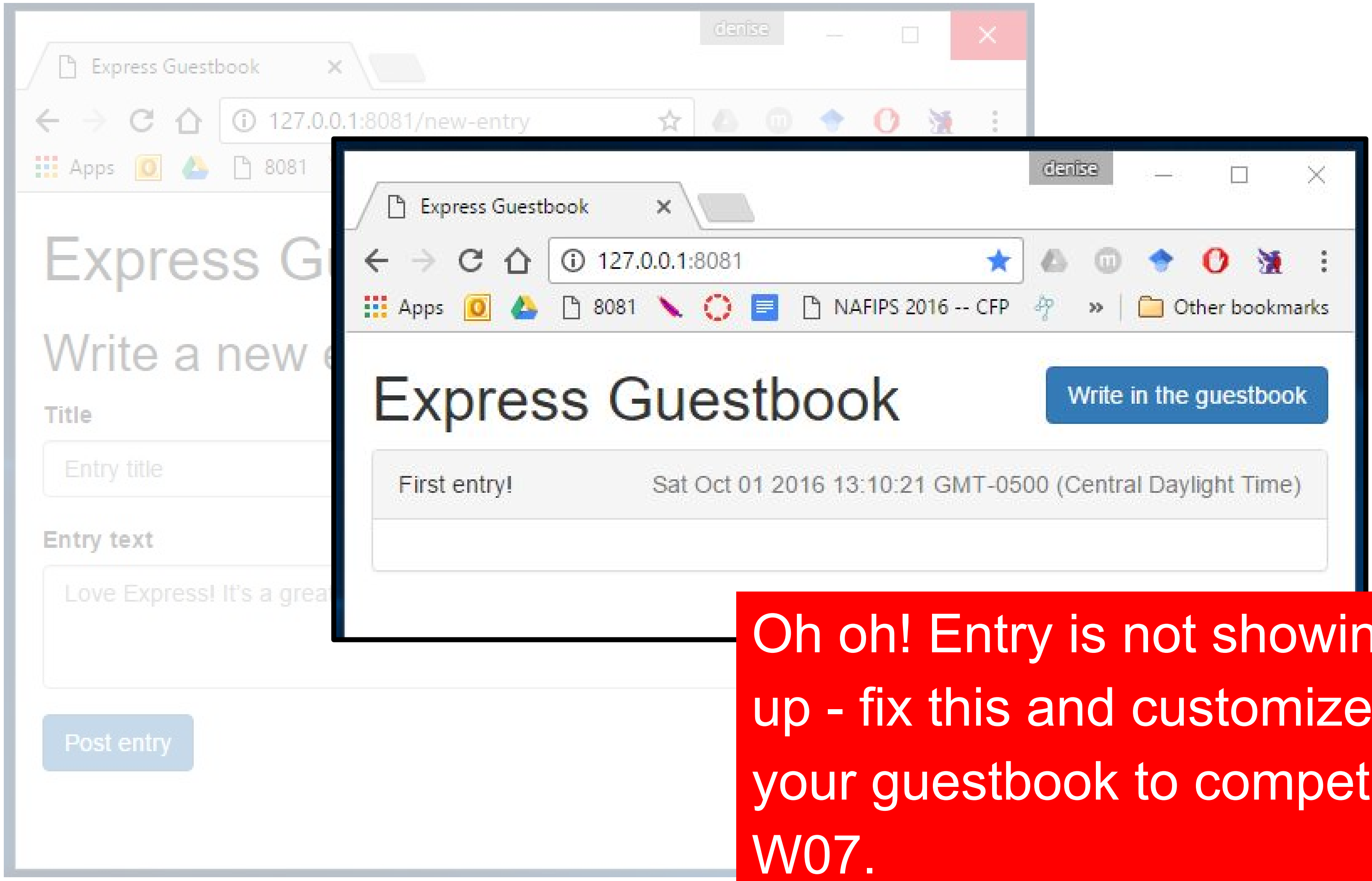
Entry title

Entry text

Love Express! It's a great tool for building websites.

Post entry

Let's Run It (W07)


Express Guestbook

127.0.0.1:8081/new-entry

Express Guestbook

Write a new entry

Title

Entry title

Entry text

Love Express! It's a great

Post entry

127.0.0.1:8081

Express Guestbook

Write in the guestbook

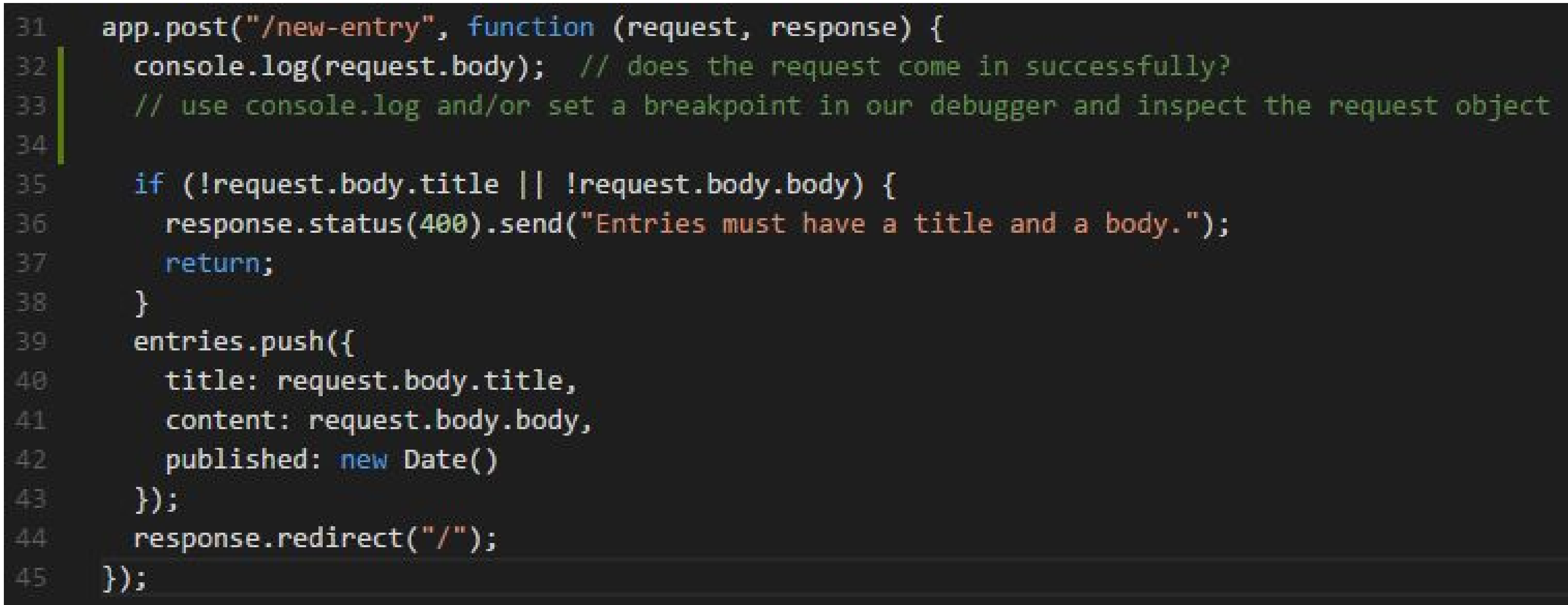
First entry! Sat Oct 01 2016 13:10:21 GMT-0500 (Central Daylight Time)

Oh oh! Entry is not showing up - fix this and customize your guestbook to compete W07.

Debug

How do we figure out where it went wrong?

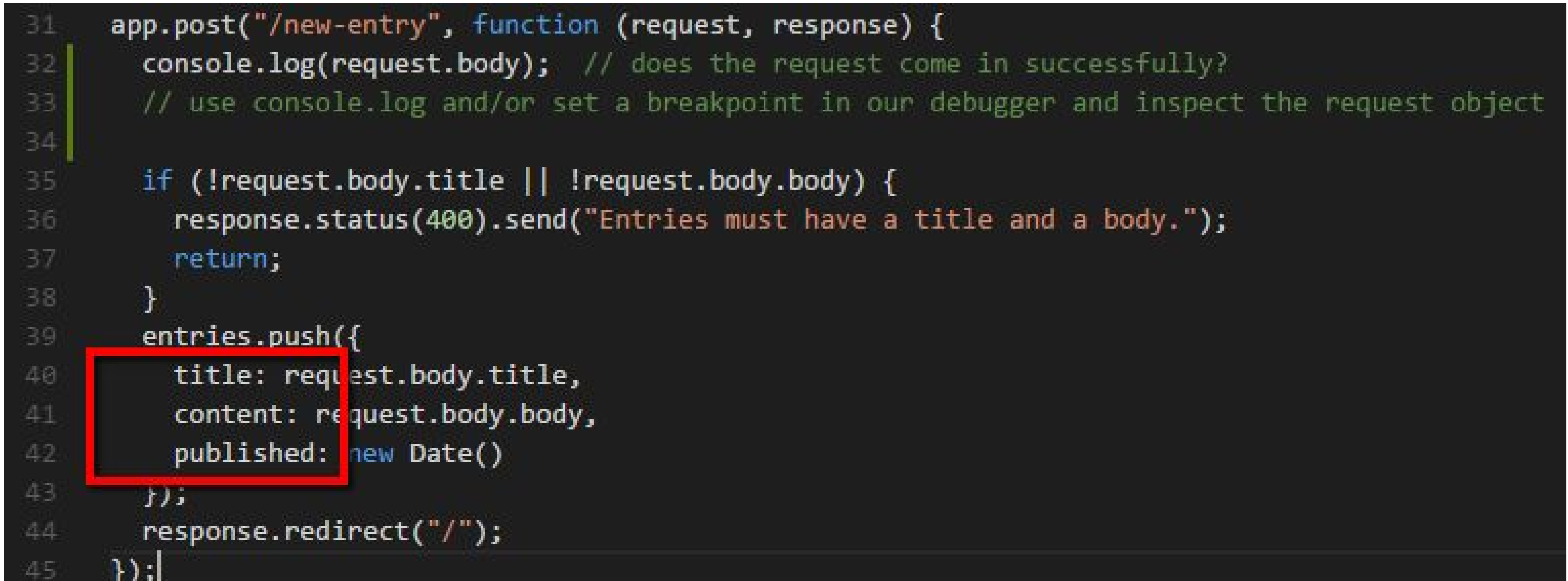
1. Right-click in browser and inspect. Is the missing content just not appearing? *Nope.. there is no entry.*
2. Back up. Did we successfully post an entry? What event triggers our post? app.**post** + **/new-entry** URI. Log it or set a breakpoint. *Yep... the request come through fine...*



```
app.post("/new-entry", function (request, response) {
  console.log(request.body);  // does the request come in successfully?
  // use console.log and/or set a breakpoint in our debugger and inspect the request object

  if (!request.body.title || !request.body.body) {
    response.status(400).send("Entries must have a title and a body.");
    return;
  }
  entries.push({
    title: request.body.title,
    content: request.body.body,
    published: new Date()
  });
  response.redirect("/");
});
```

Debug

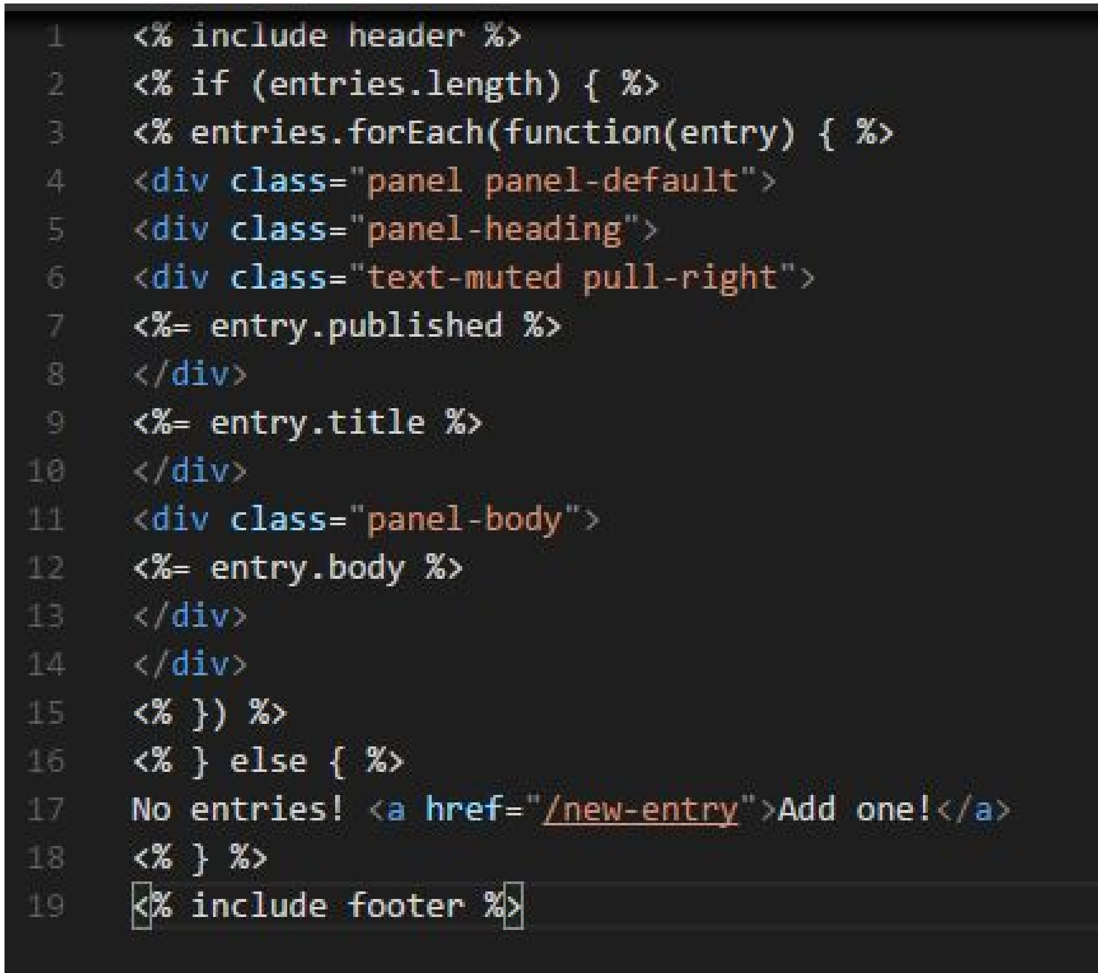
3. In app.post, we pushed a new entry. We assigned info in the request body to our entry ***title***, ***content***, *and* ***published***.
4. But when we display our entries in index.ejs... only ***title*** and ***published*** appear, but the ***content*** doesn't. *Let's look at index.ejs - why do only two of the three appear?*



```
app.post("/new-entry", function (request, response) {
  console.log(request.body);  // does the request come in successfully?
  // use console.log and/or set a breakpoint in our debugger and inspect the request object

  if (!request.body.title || !request.body.body) {
    response.status(400).send("Entries must have a title and a body.");
    return;
  }
  entries.push({
    title: request.body.title,
    content: request.body.body,
    published: new Date()
  });
  response.redirect("/");
});
```

Debug

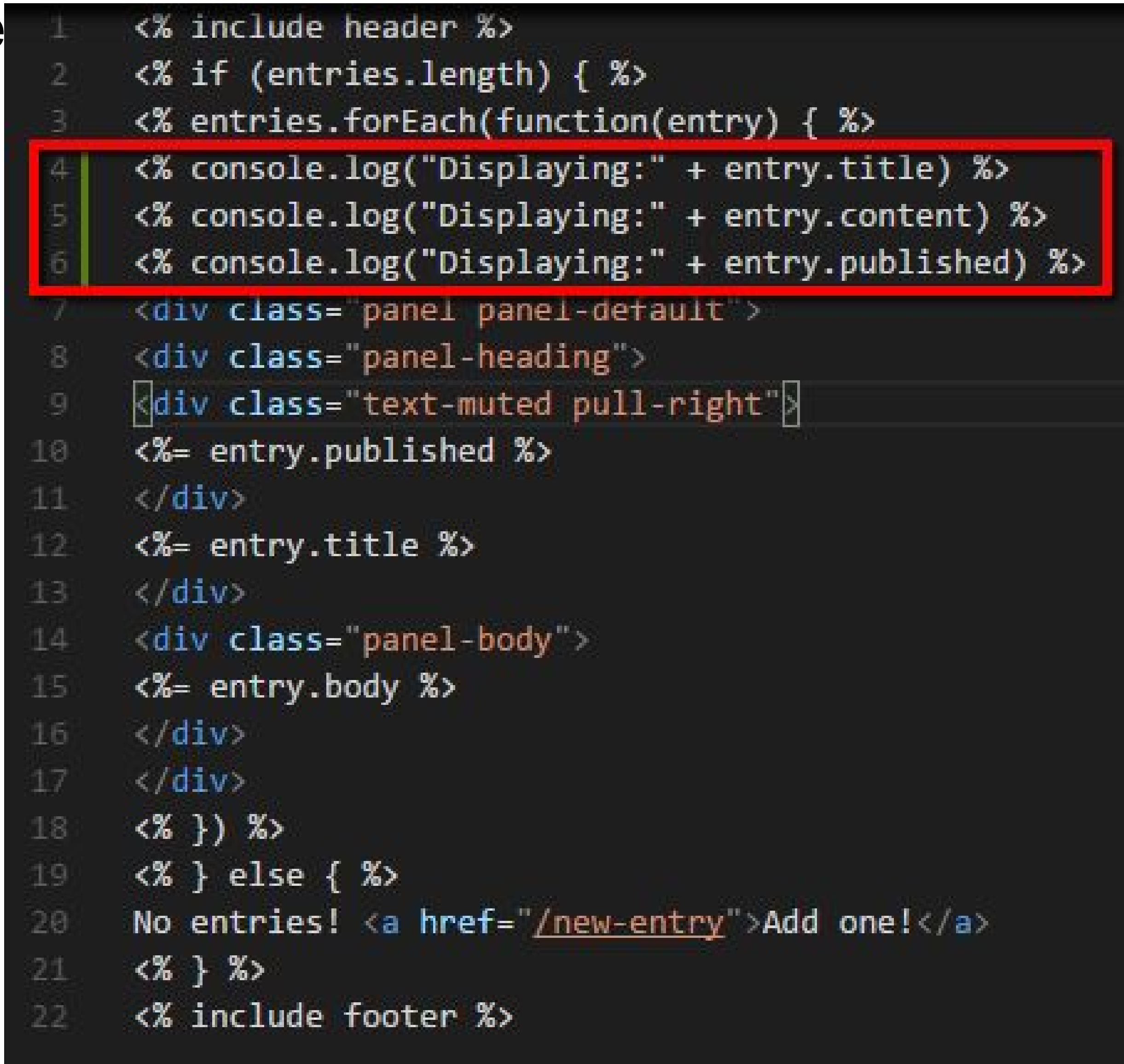
5. In index.ejs, we want to display our entry.***title****,* entry.***content****, and* entry.***published****. Why do only two of the three appear? Can you figure it out?*



```
<% include header %>
<% if (entries.length) { %>
<% entries.forEach(function(entry) { %>
<div class="panel panel-default">
<div class="panel-heading">
<div class="text-muted pull-right">
<%= entry.published %>
</div>
<%= entry.title %>
</div>
<div class="panel-body">
<%= entry.body %>
</div>
</div>
<% }) %>
<% } else { %>
No entries! <a href="/new-entry">Add one!</a>
<% } %>
<% include footer %>
```

Debug

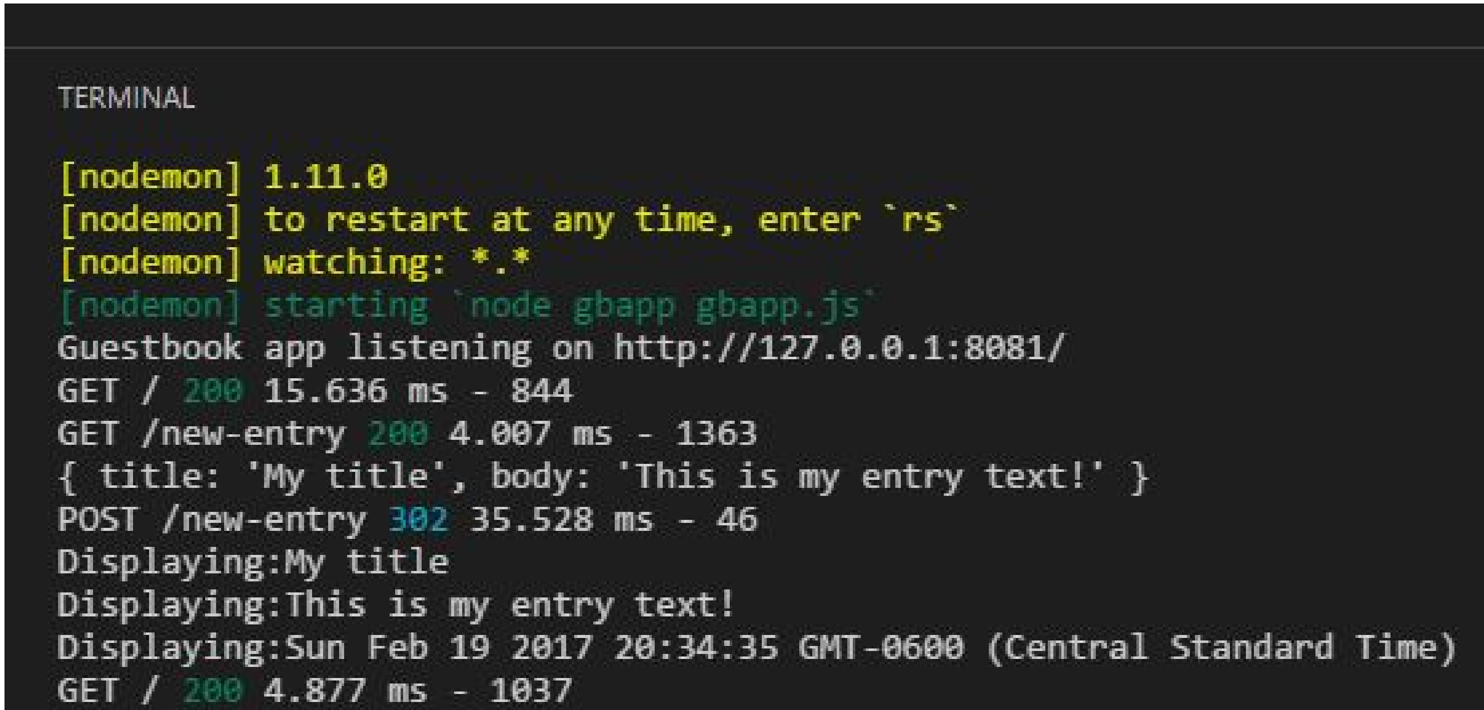
6. We can write code in our ejs files... so we can add console.log statements.



```
<% include header %>
<% if (entries.length) { %>
<% entries.forEach(function(entry) { %>
<% console.log("Displaying:" + entry.title) %>
<% console.log("Displaying:" + entry.content) %>
<% console.log("Displaying:" + entry.published) %>
<div class="panel panel-default">
<div class="panel-heading">
<div class="text-muted pull-right">
<%= entry.published %>
</div>
<%= entry.title %>
</div>
<div class="panel-body">
<%= entry.body %>
</div>
</div>
<% }) %>
<% } else { %>
No entries! <a href="/new-entry">Add one!</a>
<% } %>
<% include footer %>
```

Debug

7. All three are there and available in index.ejs. *We must be very close!*
8. Go back and verify we are displaying all three of our entry attributes correctly. *Can you debug the error now?*



```
TERMINAL

[nodemon] 1.11.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node gbapp gbapp.js`
Guestbook app listening on http://127.0.0.1:8081/
GET / 200 15.636 ms - 844
GET /new-entry 200 4.007 ms - 1363
{ title: 'My title', body: 'This is my entry text!' }
POST /new-entry 302 35.528 ms - 46
Displaying:My title
Displaying:This is my entry text!
Displaying:Sun Feb 19 2017 20:34:35 GMT-0600 (Central Standard Time)
GET / 200 4.877 ms - 1037
```

CSS

Express needs to know about **static files**.

Option:

If you want to add **css**, css files should go in the "**assets**" folder. In your web app, you'll have to tell Express to include the client-side assets found in this folder.

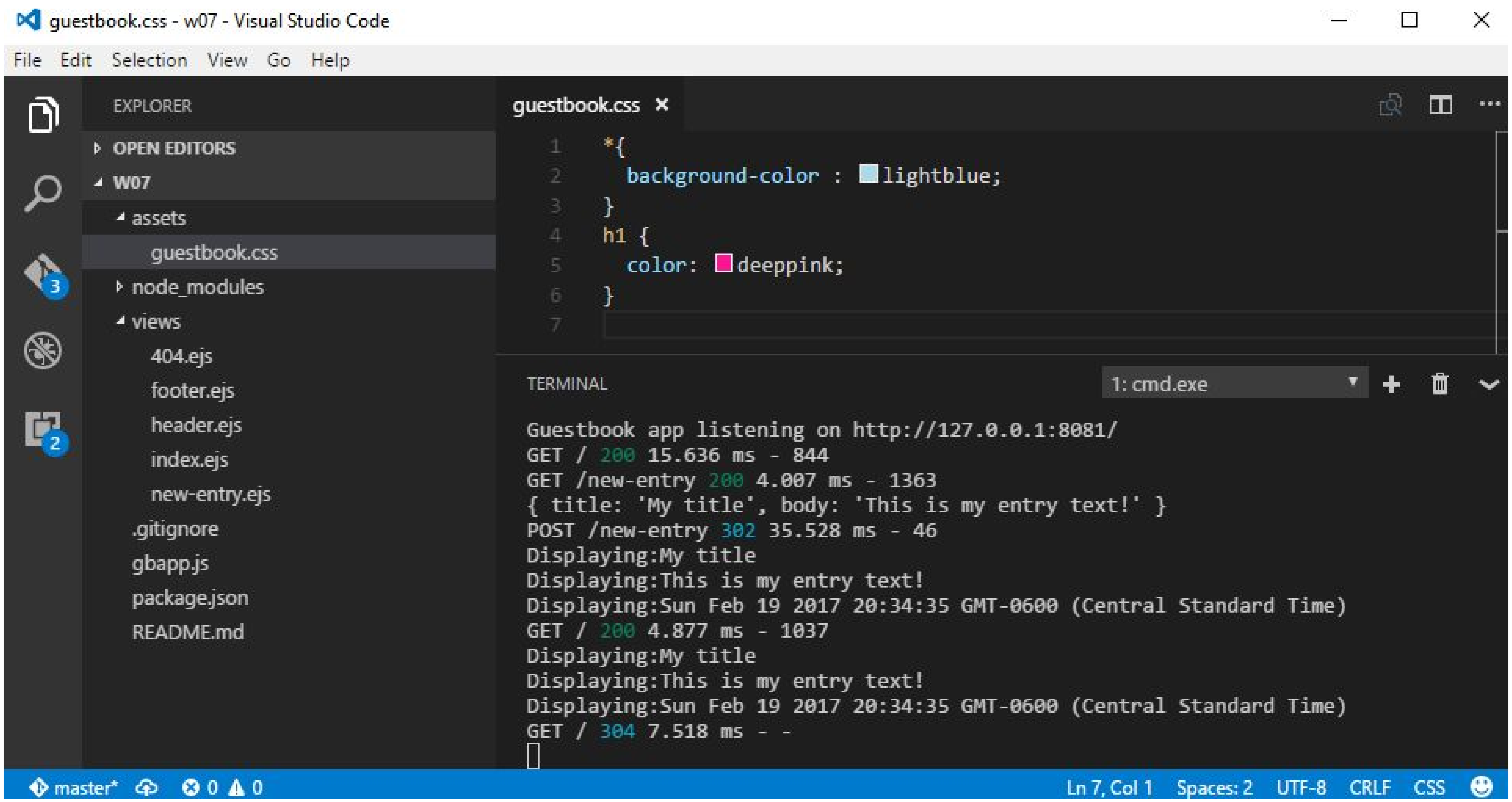
```
app.use(express.static(__dirname + '/assets'))
```

For example, if guestbook.css is in the assets folder, you can now link to it with href="`/guestbook.css`" because /assets is included the places Express will check.

More info on express.static() is available [in the docs.]()

CSS

What will happen if we add these styles?



```css
*{
  background-color : lightblue;
}
h1 {
  color: deeppink;
}
```

```
Guestbook app listening on http://127.0.0.1:8081/
GET / 200 15.636 ms - 844
GET /new-entry 200 4.007 ms - 1363
{ title: 'My title', body: 'This is my entry text!' }
POST /new-entry 302 35.528 ms - 46
Displaying:My title
Displaying:This is my entry text!
Displaying:Sun Feb 19 2017 20:34:35 GMT-0600 (Central Standard Time)
GET / 200 4.877 ms - 1037
Displaying:My title
Displaying:This is my entry text!
Displaying:Sun Feb 19 2017 20:34:35 GMT-0600 (Central Standard Time)
GET / 304 7.518 ms - -
```

Git

Common git steps to save your local work in your cloud space.

Create C:\44563\w07 folder.

1. Right-click "create repository here" & OK
2. Right-click "TortoiseGit / Add / Select All / OK"
3. Click "Commit"
4. Add message, e.g. "initial commit", click Commit & Close.
5. Right-click "TortoiseGit / Settings / Git / Remote.
6. Remote: origin
7. URL: your URL, e.g. mine is: https://bitbucket.org/professorcase/w07, OK, Yes, OK. Close.
8. Right-click "TortoiseGit / Push", OK. Close.

Git Bash

Gitting started with git, at the command line 😍

Create C:\44563\w07 folder. Populate it as needed.

1. Right-click, choose Git Bash Here
2. git add .
3. git commit -m "initial commit"
4. git remote add origin "https://bitbucket.org …"
5. git push

Notes:

Use git status to see what’s up with your repo

Use git --help to get help at any time.

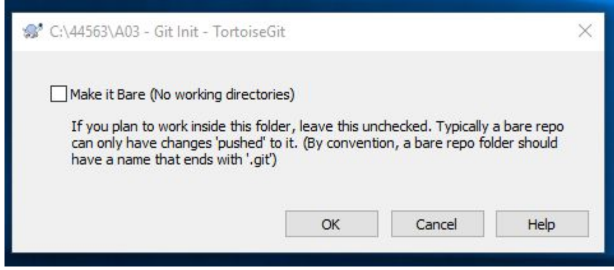
Repeat steps

A03 & Git

A03 - set up

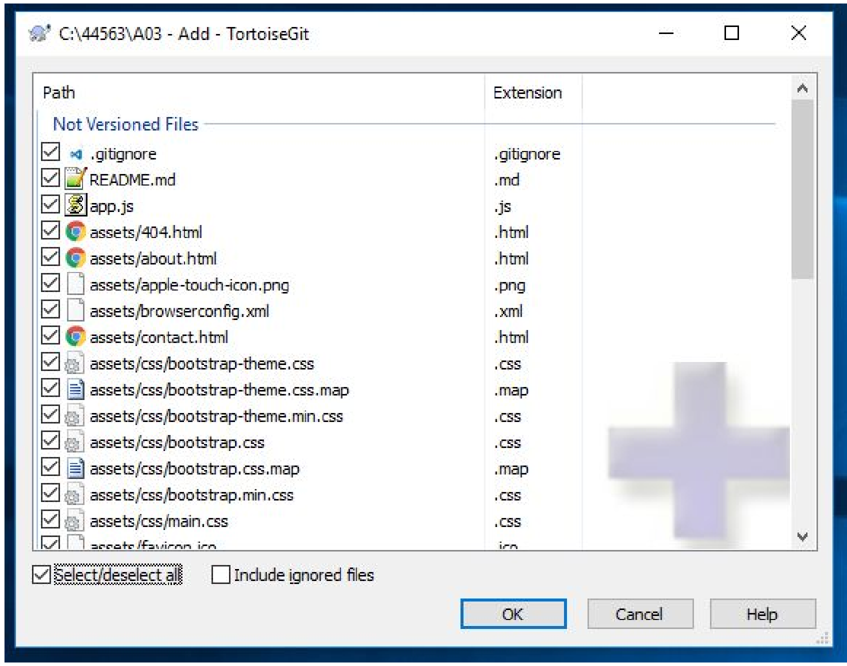
1. Install **Git** and **TortoiseGit**.
2. Create C:\44563\A03Lastname.
3. Copy in A02 code (later, adjust paths as needed).
4. **Create a git repo** in A03 folder.
5. Git **add** all your files.
6. Git **commit** with message "initial commit".

Using TortoiseGit, right-click on your folder and select "Git create repository here" to make the hidden git folder. Right-click and say git Add, select all. Click OK, click Commit. Add your message. Git bash fans: 1) Right-click on the folder, 2) Git Bash Here, 3) git init, 4) git add ., 5) git commit -m.


C:\44563\A03 - Git Init - TortoiseGit

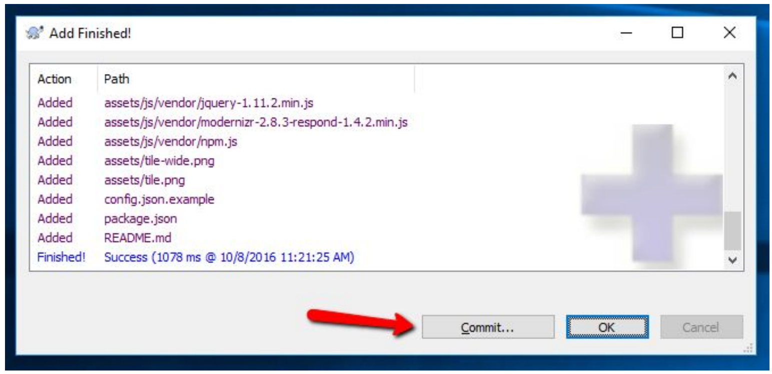
☐ Make it Bare (No working directories)

If you plan to work inside this folder, leave this unchecked. Typically a bare repo can only have changes 'pushed' to it. (By convention, a bare repo folder should have a name that ends with '.git')


C:\44563\A03 - Add - TortoiseGit

Path	Extension
Not Versioned Files	
☑ .gitignore	.gitignore
☑ README.md	.md
☑ app.js	.js
☑ assets/404.html	.html
☑ assets/about.html	.html
☑ assets/apple-touch-icon.png	.png
☑ assets/browserconfig.xml	.xml
☑ assets/contact.html	.html
☑ assets/css/bootstrap-theme.css	.css
☑ assets/css/bootstrap-theme.css.map	.map
☑ assets/css/bootstrap-theme.min.css	.css
☑ assets/css/bootstrap.css	.css
☑ assets/css/bootstrap.css.map	.map
☑ assets/css/bootstrap.min.css	.css
☑ assets/css/main.css	.css
☑ assets/favicon.ico	.ico

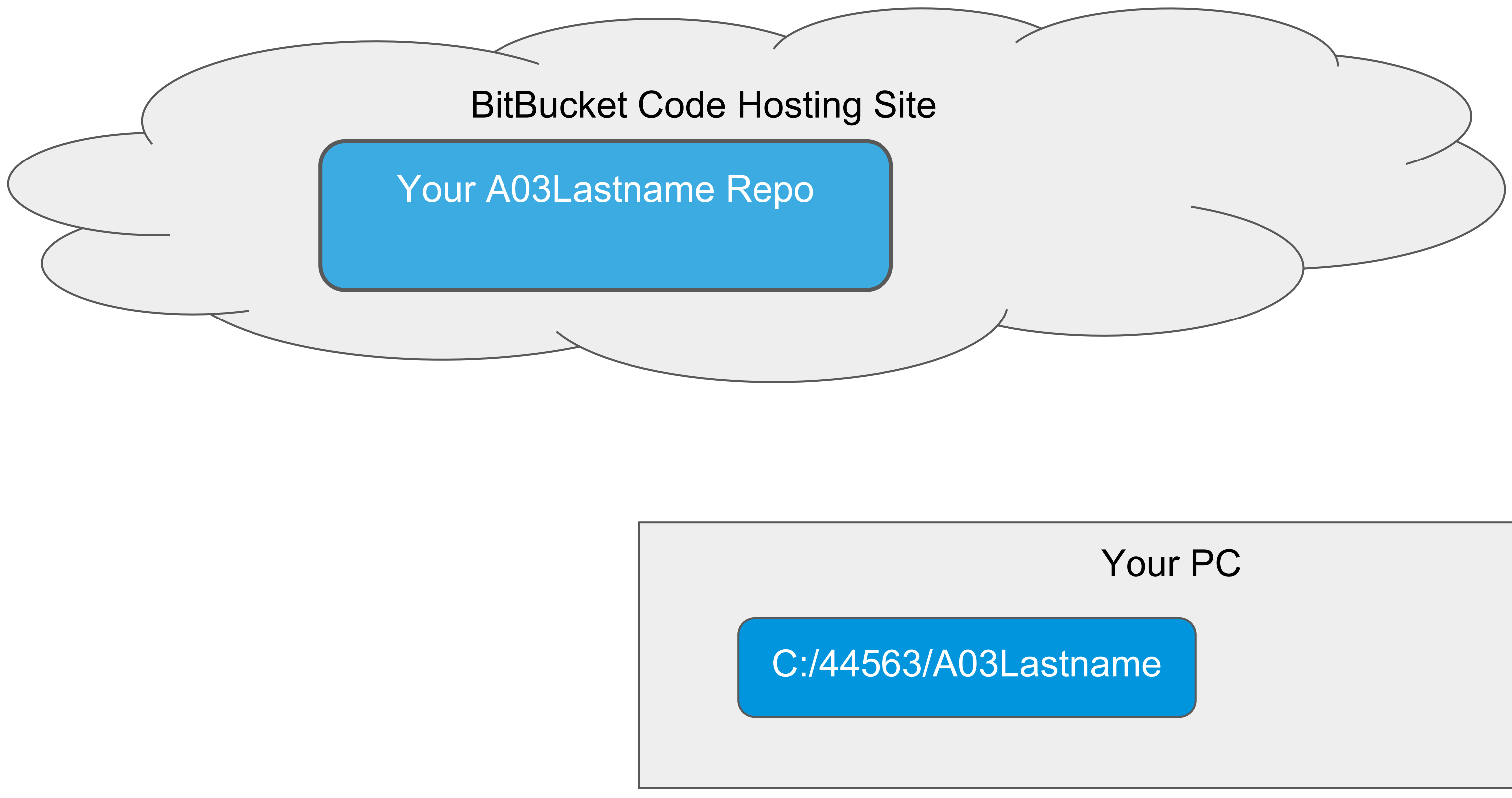
☑ Select/deselect all ☐ Include ignored files


Add Finished!

Action	Path
Added	assets/js/vendor/jquery-1.11.2.min.js
Added	assets/js/vendor/modernizr-2.8.3-respond-1.4.2.min.js
Added	assets/js/vendor/npm.js
Added	assets/tile-wide.png
Added	assets/tile.png
Added	config.json.example
Added	package.json
Added	README.md
Finished!	Success (1078 ms @ 10/8/2016 11:21:25 AM)

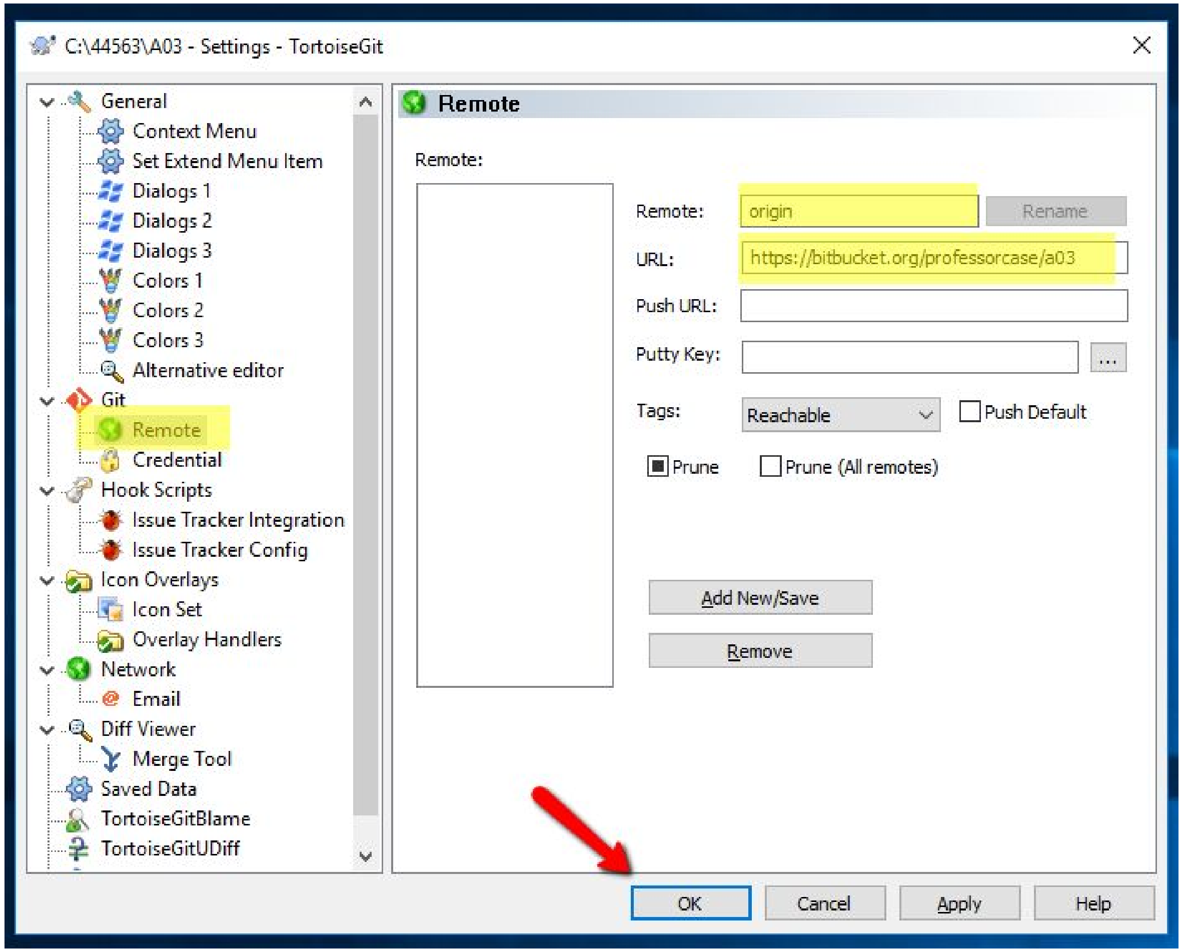
A03 - create cloud repo

1. Open your BitBucket profile.
2. Create a repo named a03Lastname.
3. See instructions under "I have an existing project"



A03 - origin alias

1. Assign your cloud repo URL to alias **origin**. Use git bash to execute given command OR use TortoiseGit /Settings /Remote.


C:\44563\A03 - Settings - TortoiseGit

Remote

Remote:

Remote: origin

URL: https://bitbucket.org/professorcase/a03

Push URL:

Putty Key:

Tags: Reachable

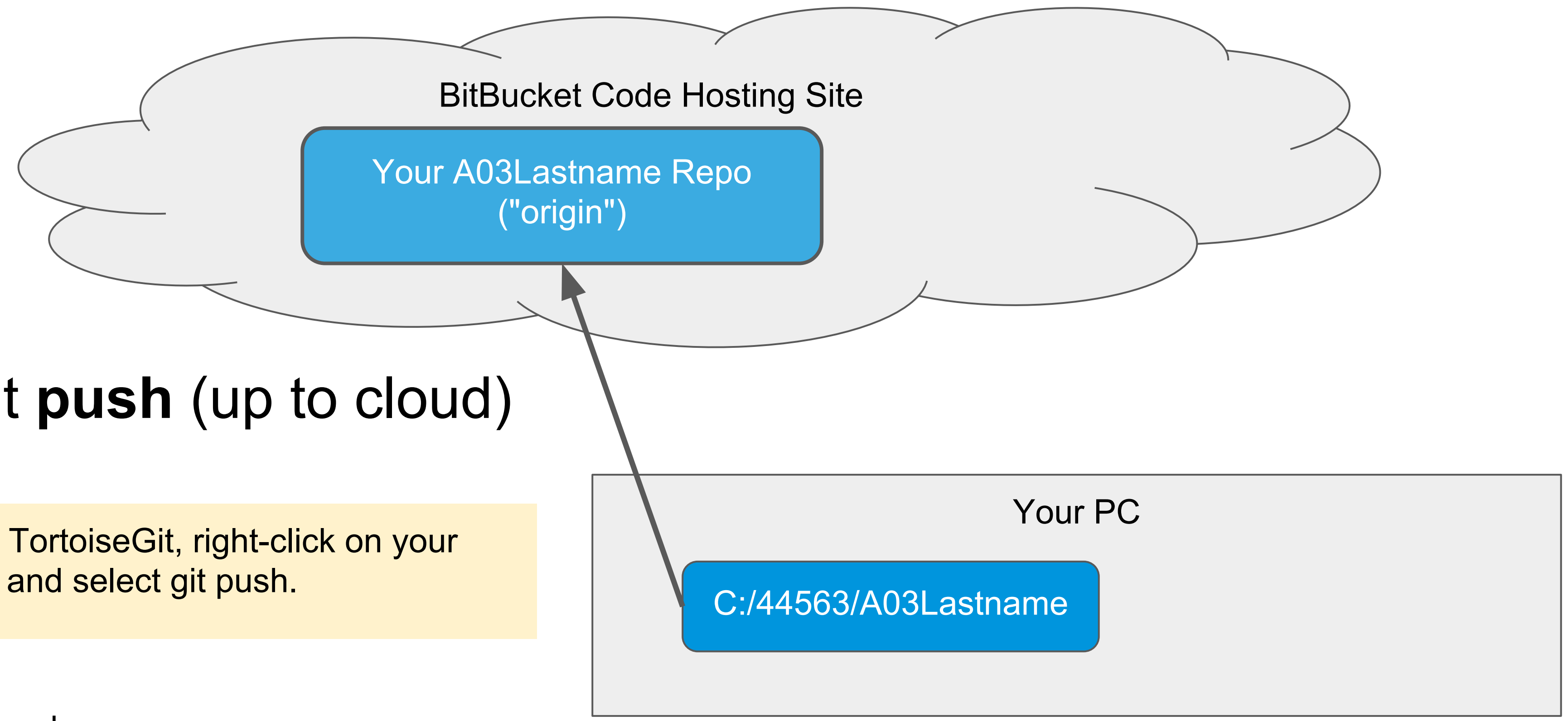
☐ Push Default

■ Prune ☐ Prune (All remotes)

$ git remote add origin "http://bitbucket.org/ …"

A03 - push to cloud

1. From your local folder, git **push** up to your cloud.
2. Sign in with your BitBucket creds as needed.



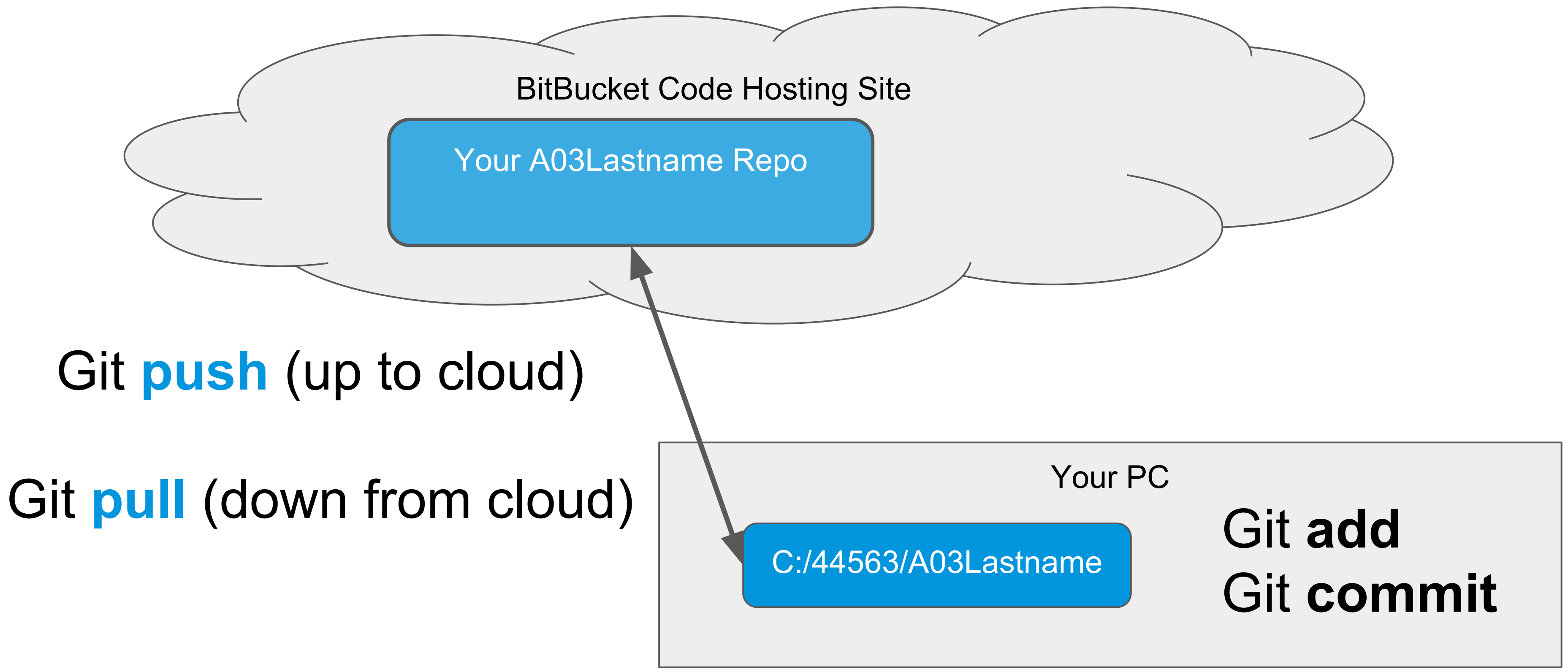
Git **push** (up to cloud)

Using TortoiseGit, right-click on your folder and select git push.

```
$ git push
```

A03 - git commands

1. Use origin as an alias for a cloud repo.
2. Add and commit locally.
3. Push and pull from cloud.



Git

Common git steps to save your local work in your cloud space.

Create C:\44563\a03lastname folder.

1. Right-click "create repository here" & OK
2. Right-click "TortoiseGit / Add / Select All / OK"
3. Click "Commit"
4. Add message, e.g. "initial commit", click Commit & Close.
5. Right-click "TortoiseGit / Settings / Git / Remote.
6. Remote: ***origin***
7. URL: your URL, e.g. ***https://bitbucket.org/youraccount/a03lastname***, OK, Yes, OK. Close.
8. Right-click "TortoiseGit / Push", OK. Close.