UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 11

IMPLEMENTING WEB APPS WITH NODE.JS

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

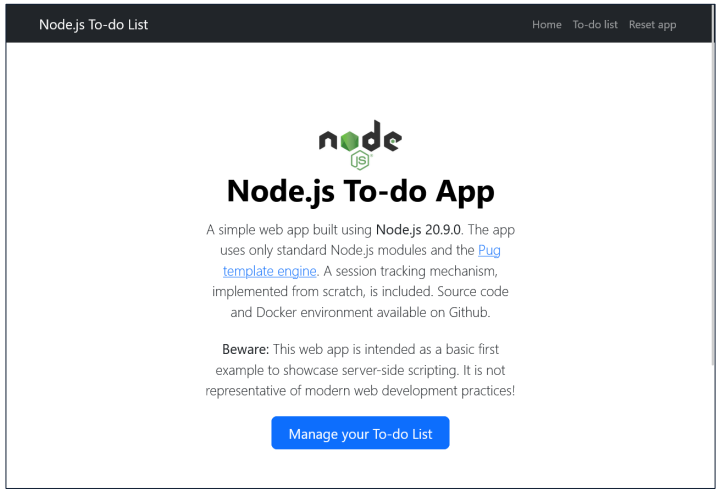
https://luistar.github.io

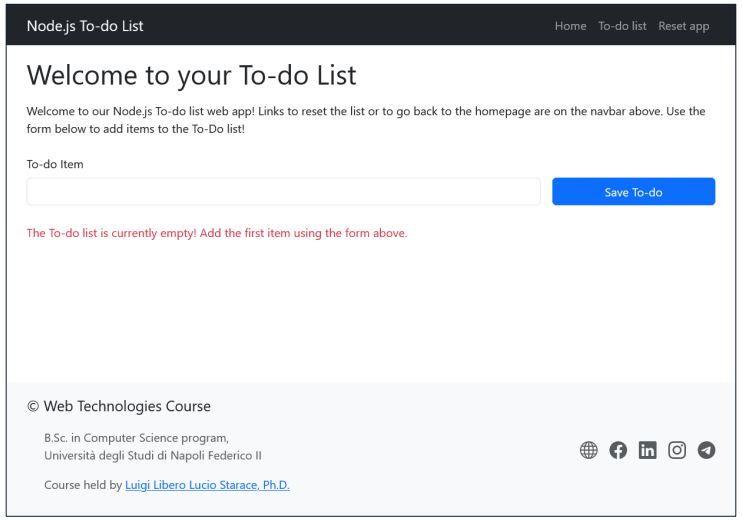
https://docenti.unina.it/luigiliberolucio.starace

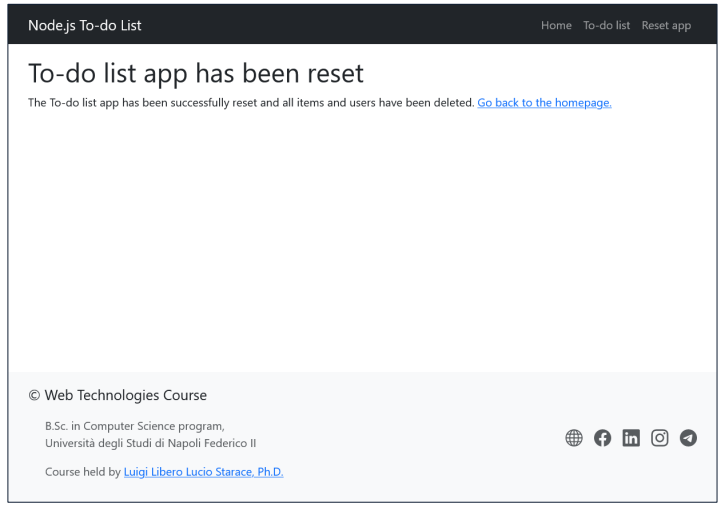
TO-DO LIST WEB APP

Will this be less painful than the infamous CGI+Bash example?

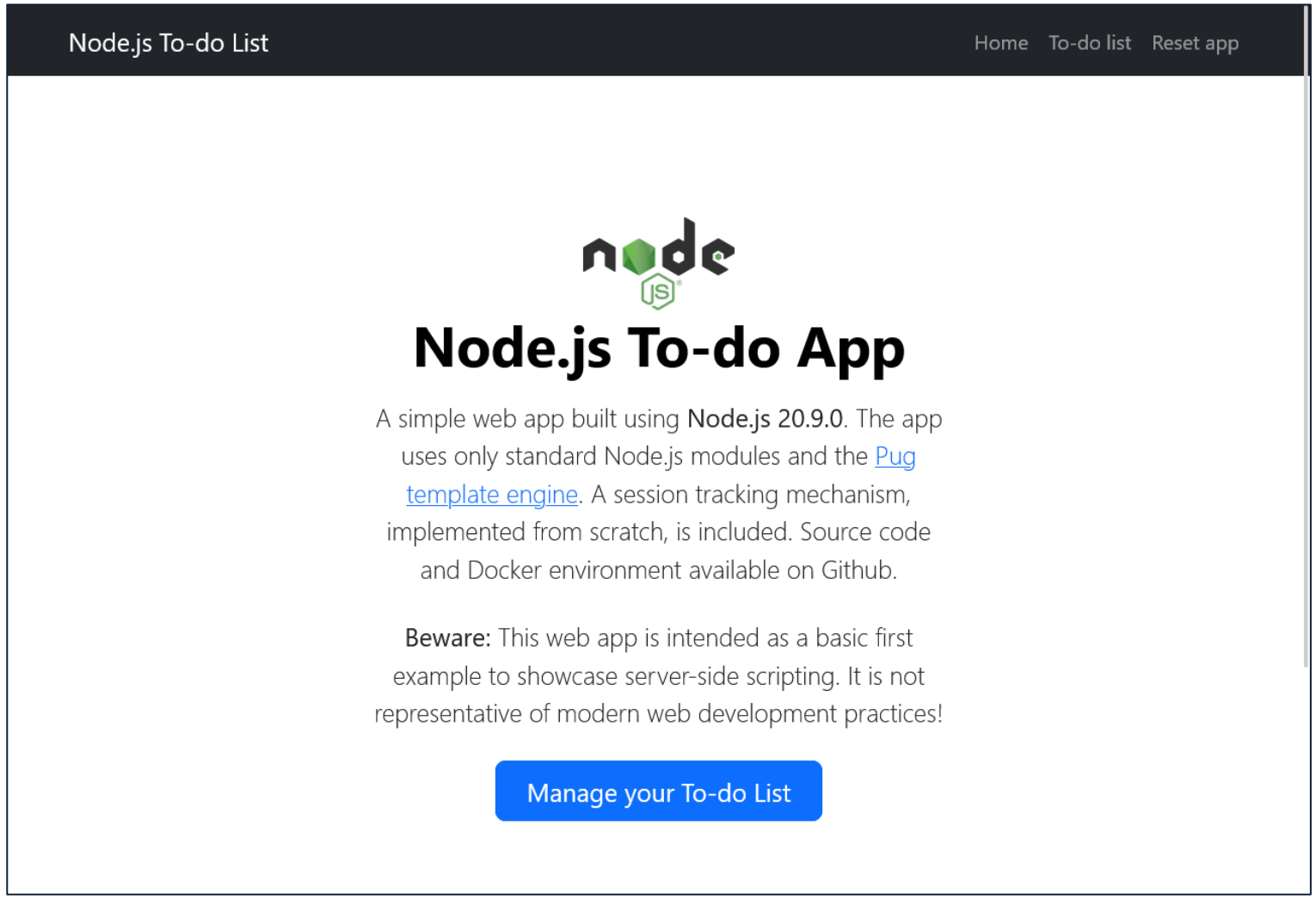
THE TO-DO LIST WEB APP





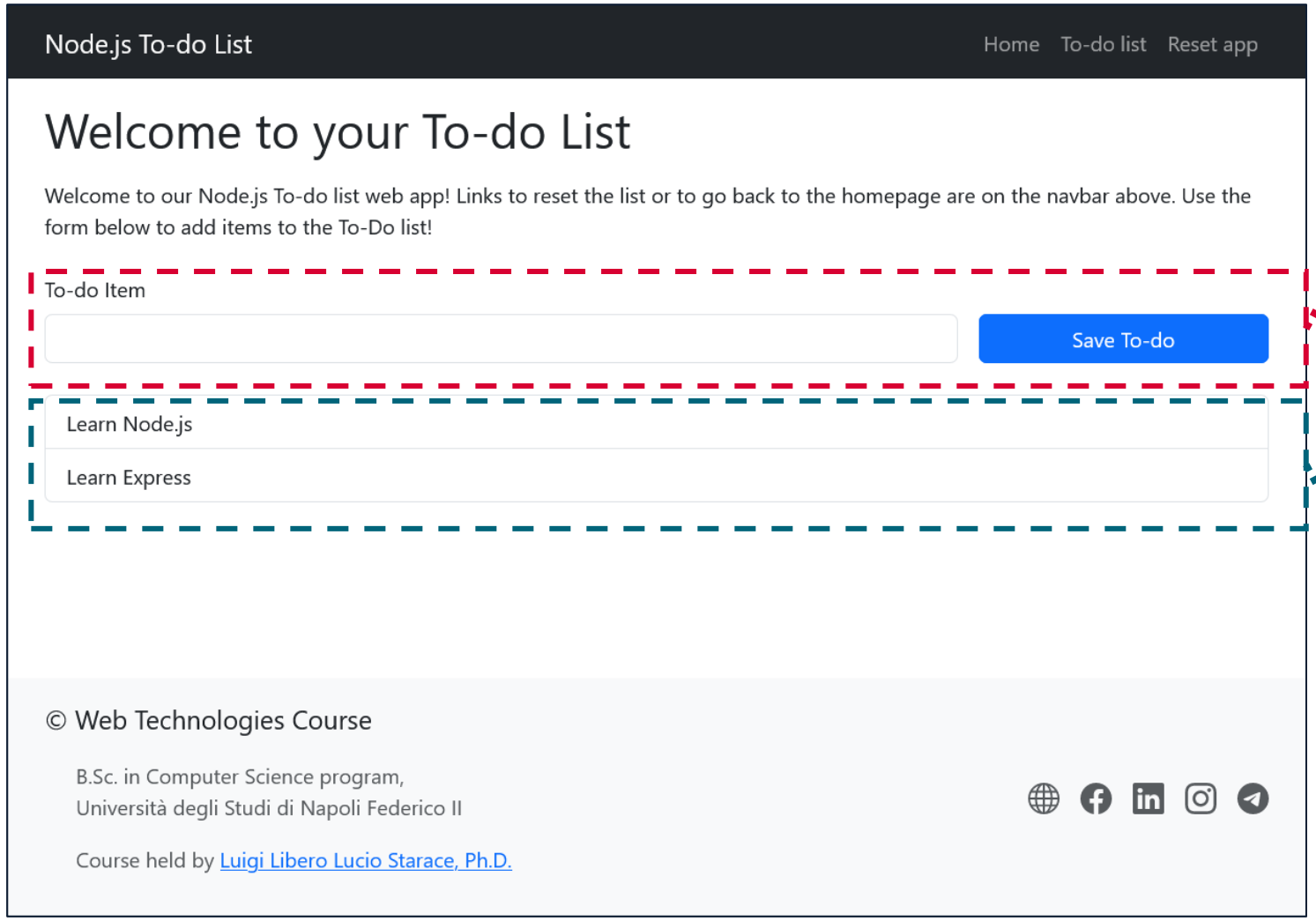


TO–DO LIST APP: HOMEPAGE



- Landing page
- Includes links to the list page and the reset app page.

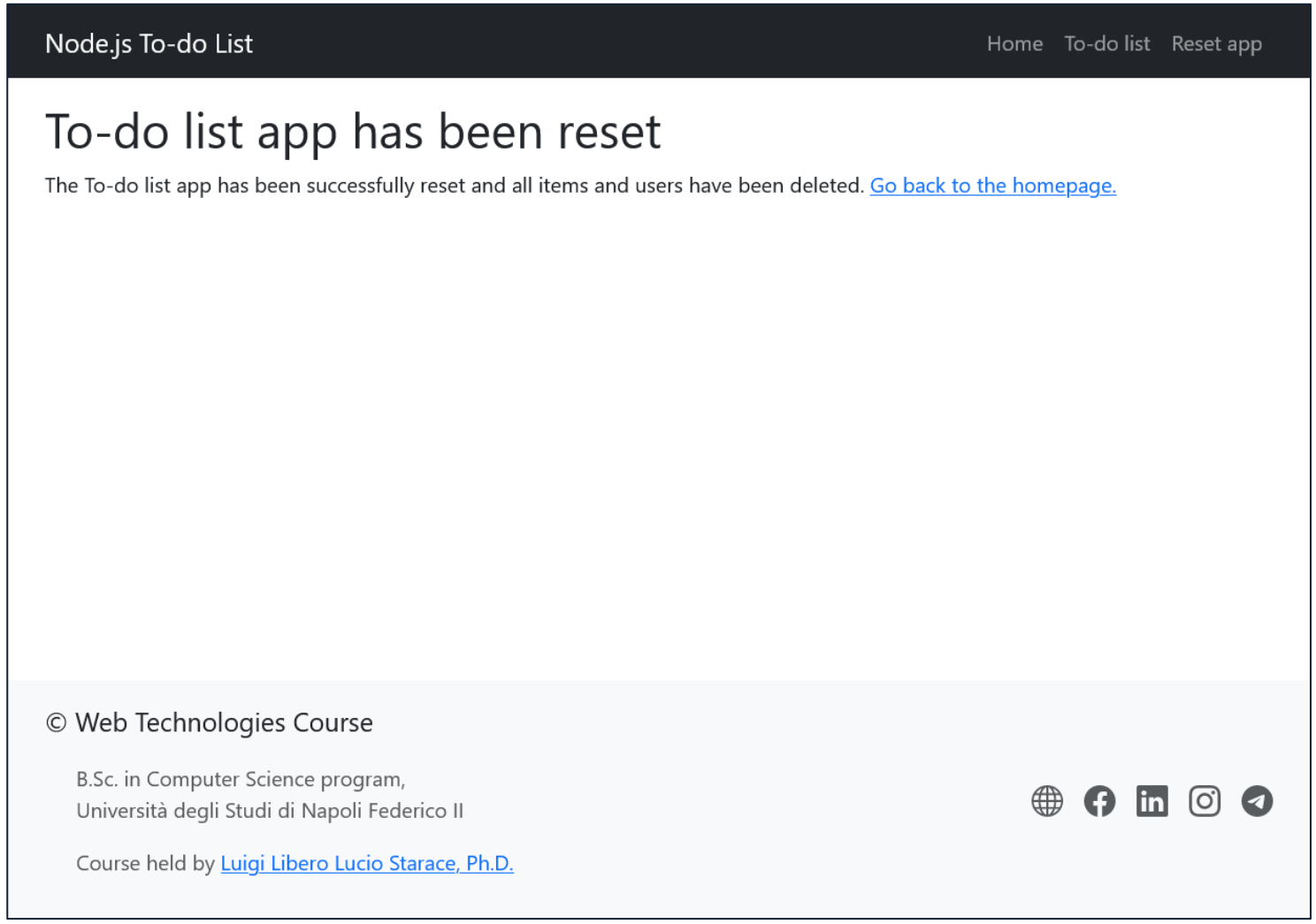
TO–DO LIST APP: THE LIST



Form for saving new To-do items. Submits using POST to the same url as the page.

List showing the saved To-do list items

TO–DO LIST APP: RESET PAGE



- Resets the app (i.e., deletes all saved To-do items)

IMPLEMENTING A TO–DO LIST IN NODE.JS

- We start by creating an http server

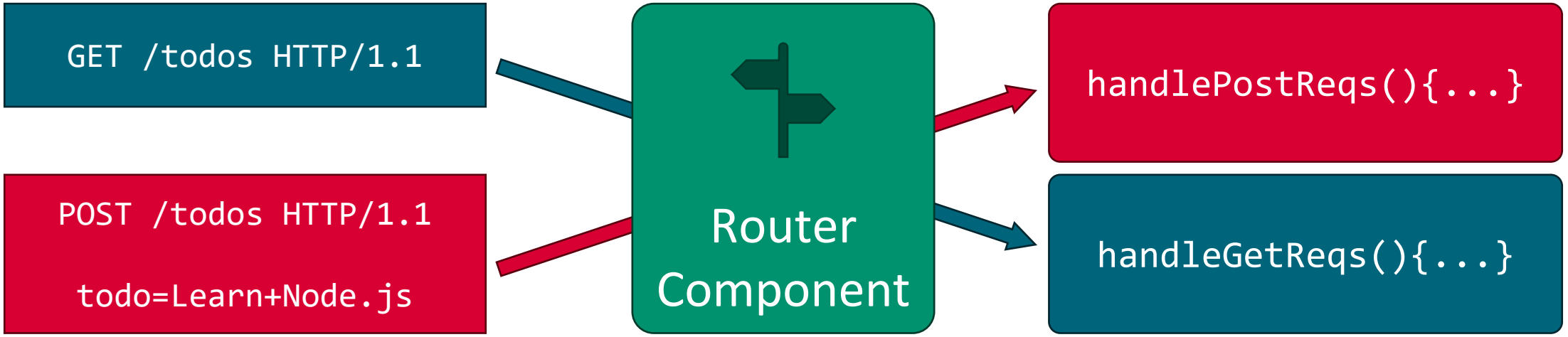
```
import http from 'http';

const PORT = 3000;

let server = http.createServer();
server.listen(PORT);

server.on('request', handleRequest);

function handleRequest(request, response){
  /* handle request */
}
```

WEB APPS WITH BARE NODE.JS

- Node.js allows us to easily and efficiently implement an http server leveraging built-in modules and its single threaded event loop
- Node.js also does some pre-processing on requests (i.e.: parses the headers) and provides an abstraction for requests and responses.
- Aside from that, we're on our own when it comes to implement the To-do list app
- The first three issues we need to tackle are
  - **Routing**
  - **Templating**
  - **Parsing request bodies**

ROUTING

- Our web application will be handling a many different types of HTTP requests (e.g.: show homepage, show list, reset app, …)
- A first core problem to address is **routing**
- **Given a request, what code should I execute to serve that request?**



ROUTING — PATHS

- We need to handle requests to different paths
- An homepage, a page where To-do items are listed, a reset page, …
- Requests to each path is possibly handled differently

```
function handleRequest(request, response){
  switch(request.url){
    case "/":
      renderHomepage(request, response);
      break;
    case "/reset":
      handleResetRequest(request, response);
      break;
    case "/todo":
      handleTodoListRequest(request, response);
      break;
    /* and so on... */
    default:
      handleError(request, response, 404);
  }
}
```

ROUTING — HANDLING HTTP METHODS

Requests to a certain path may also be handled differently depending on the used HTTP method

- **GET** requests to **/todos** get the list of to-do items
- **POST** requests to **/todos** try to save a new item

```
function handleTodoListRequest(request, response, context={}){
  switch(request.method){
    case "GET":
      handleTodoListRequestGet(request, response, context); break;
    case "POST":
      handleTodoListRequestPost(request, response, context); break;
    default:
      handleError(request, response, 405, "Unsupported method");
  }
```

SERVING STATIC FILES

- If our HTML output includes static files (e.g.: images, css or js files), we need to configure the Router component to serve these files

```
switch(request.url){
  /* ... */
  case "/css/bootstrap.css":
      fs.readFile('./static/css/bootstrap.css', function(err, data){
        response.writeHead(200, {'Content-Type': 'text/css'});
        response.end(data, 'utf-8');
      }); break;
  case "/img/node.png":
    fs.readFile('./static/img/node.png', function(err, data){
      response.writeHead(200, {'Content-Type': 'image/png'});
      response.end(data, "binary");
    }); break;
}
```

HANDLING ERRORS

- When no one of the known routes applies, we can return a 404

```
switch(request.url){
  /* other routes */
  default:
      handleError(request, response, 404, "Web page not found");
}
```

```
function handleError(request, response, statusCode, message){
  let renderedContent = pug.renderFile("./templates/errorPage.pug",
    {"code": statusCode, "description": message});
  response.writeHead(statusCode, {"Content-Type": "text/html"});
  response.end(renderedContent);
}
```

ROUTING: THERE'S MORE TO THAT

Routing seems quite simple, but be aware that it can grow more complex as apps grow (we'll see soon enough!):

- Some routes might be accessible only to some users (e.g.: admins)
- Some routes may depend on patterns or parameters in the request path (e.g.: `GET /users/42/friends`, where 42 could be any user id)
- There might be the need to run two or more pieces of code to handle a request (e.g.: one function that just logs data, and a different function that prepares the response)

TEMPLATE ENGINES

PRODUCING HTML OUTPUTS

- At some point, our web app will need to produce some HTML code to send back in the response body, so browsers can render it.
- In our first example, we wrote the HTML manually

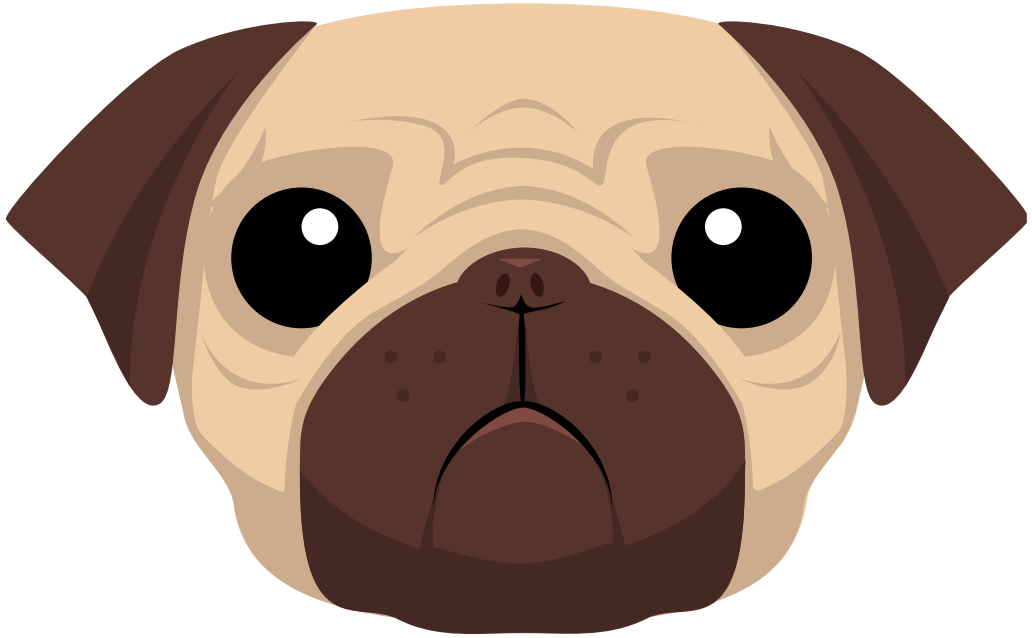
```
let server = http.createServer(function(request, response){
  let course = "Web Technologies";
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(`<!DOCTYPE html>
  <html><body>
    <h1>Hello ${course}</h1>
    <p>Current date is ${new Date().toDateString()}</p>
  </body></html>`);
  response.end();
}).listen(PORT);
```

TEMPLATE ENGINES

- Some parts of the HTML we produce may be repeated in multiple pages (e.g.: navbar, footer, etc.)
- As pages grow more and more complex, building a response by manipulating and concatenating strings becomes rapidly unfeasible.
- **Template engines** are great to address this issue!

TEMPLATING WITH PUG (FORMERLY JADE)

- Pug is a high-performance template engine implemented in JavaScript (ports available in many other languages)
- Whitespace sensitive syntax (Python-like) for writing HTML templates
- Pug templates can be easily «**compiled**» to HTML code



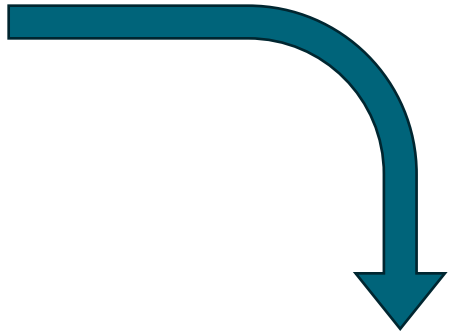
INSTALLING AND USING PUG

- Installing Pug is as simple as running `npm install pug`
- The simplest way to use Pug is as follows:

```
import pug from "pug"

//basic_example.pug is the file with the template to render
let html = pug.renderFile("./templates/basic_example.pug");

console.log(html); //html contains the generated HTML code
```

A FIRST PUG TEMPLATE

```
doctype html
html(lang="en")
  head
    title Hello Pug!
  body
    h1(class="foo") First Pug template
    p This is our first Pug template!
```



```
<!DOCTYPE html>
<html lang="en">
<head><title>Hello Pug!</title></head>
<body>
  <h1 class="foo">First Pug template</h1>
  <p>This is our first Pug template!</p>
</body>
</html>
```

TEMPLATING WITH PUG: INCLUDES

- Pug's much more than a different way to write HTML…
- For a starter, a template can **include** other templates
- This is a big help for template re-use!

```
//- index.pug
doctype html
html
  include partials/head.pug
  body
    h1 Hello Pug!
    include partials/footer.pug
```

```
//- partials/head.pug
head
  title Hello Pug
  script(src='/js/script.js')
  link(rel='stylesheet' href='/style.css')
```

```
//- partials/footer.pug
footer#footer
  p Copyright (c) Web Tech
```

TEMPLATING WITH PUG: INCLUDES

```
//- index.pug
doctype html
html
  include partials/head.pug
  body
    h1 Hello Pug!
    include partials/footer.pug
```

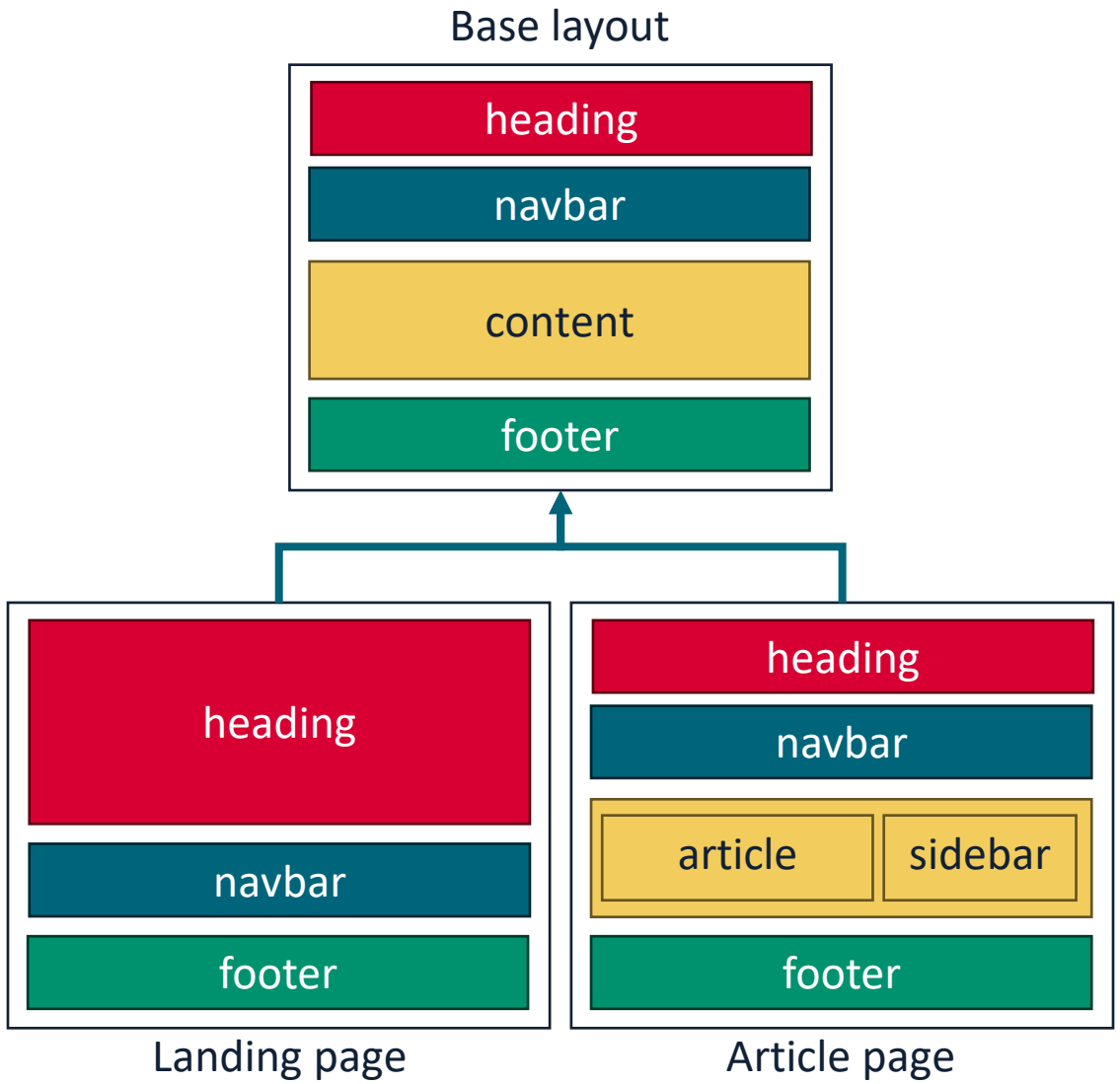
```
//- partials/head.pug
head
  title Hello Pug
  script(src='/js/script.js')
  link(rel='stylesheet' href='/style.css')
```

```
//- partials/footer.pug
footer#footer
  p Copyright (c) Web Tech
```

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello Pug</title>
  <script src="/js/script.js">
  </script>
  <link rel="stylesheet"
        href="/style.css"/>
</head>
<body>
  <h1>Hello Pug!</h1>
  <footer id="footer">
    <p>Copyright (c) Web Tech</p>
  </footer>
</body>
</html>
```

TEMPLATING WITH PUG: INHERITANCE

- When designing web applications, it is common to have a common layout shared by all web pages
- Each web page can eventually override some parts of the common template
- In Pug, such scenarios are supported with **template inheritance**



TEMPLATING WITH PUG: INHERITANCE

- Pug supports template inheritance via the `block` and `extends` keywords
- A block is a «block» of Pug template that child templates may replace
- Blocks may also contain default content, if appropriate

```
//- base-layout.pug
doctype html
html
  head
    title Pug Inheritance
  body
    block heading
      h1 Pug Ineritance
    block content
    block footer
      footer This is the default footer.
```

- The example on the right defines three blocks: `heading`, `content`, and `footer`. `heading` and `footer` contain default content.

TEMPLATING WITH PUG: INHERITANCE

- A Pug template can extend other templates using the `extends` keyword
- Templates that extend other templates may override some of the templates defined the parent template
- In the example, the content and the footer blocks are **overridden**
- The heading block is not overridden, and will display the default value.

```
extends ./base-layout.pug

block content
  p The actual content of the homepage.

block footer
  footer This is a specialized footer.
```

TEMPLATING WITH PUG: INHERITANCE

```
//- base-layout.pug
doctype html
html
  head
    title Pug Inheritance
  body
    block heading
      h1 Pug Ineritance
    block content
    block footer
      footer This is the default footer.
```

```
extends ./base-layout.pug

block content
  p The actual content of the homepage.

block footer
  footer This is a specialized footer.
```

```
<!DOCTYPE html>
<html>
<head>
  <title>Pug Inheritance </title>
</head>
<body>
  <h1>Pug Ineritance </h1>
  <p>
    The actual content of the
    homepage.
  </p>
  <footer>
    This is a specialized
    footer.
</footer>
</body>
</html>
```

TEMPLATING WITH PUG: LOCALS

- Pug templates can also render content based on a provided data object (a.k.a. «**locals**»), passed to the **render()** or **renderFile()** functions

```
import pug from "pug";

let html = pug.renderFile("./locals-example.pug", {
  "product": {
    "name": "Samsung S24",
    "description": "Smartphone"
  }
});

console.log(html);
```

```
- let footer = "a locally defined variable"

h1 #{product.name.toUpperCase()}
p Description: #{product.description}
footer #{footer}
```

TEMPLATING WITH PUG: LOCALS

- The code contained within #{ and } is evaluated, escaped, and buffered in the output

```
import pug from "pug";

let html = pug.renderFile("./locals.pug", {
  "product": {
    "name": "Samsung S24",
    "description": "Smartphone"
  }
});

console.log(html);
```

```
//- locals.pug
- let footer = "a nice string"

h1 #{product.name.toUpperCase()}
p Description: #{product.description}
footer #{footer}
```

```
<h1>SAMSUNG S24</h1>
<p>Description: Smartphone</p>
<footer>a nice string</footer>
```

TEMPLATING WITH PUG: CONDITIONALS

- Often, templates are rendered differently depending on certain conditions
- To this end, Pug supports a familiar `if`/`else` construct

```
let user = {"isAdmin": true, "name": "Brendan"};
let html = pug.renderFile("./conditionals.pug", {"user": user});
```

```
if !user
  a(href="/login") Please, authenticate yourself.
else if user.isAdmin
  p At your command, #{user.name}
else
  p Welcome back, #{user.name}
```

```
<p>At your command, Brendan</p>
```

TEMPLATING WITH PUG: ITERATIONS

- A common task when working with templates is **iterating** over sequences of data

```
let items = [
  {"name": "Margherita", "price": 5.50}, {"name": "Marinara", "price": 5.00},
  {"name": "Capricciosa", "price": 6.50}
]
let html = pug.renderFile("./iteration.pug", {"items": items});
```

```
ul
  each item in items
    li #{item.name} - € #{item.price.toFixed(2)}
  else
    li No pizza is available at the moment!
```

```
<ul>
  <li>Margherita - € 5.50</li>
  <li>Marinara - € 5.00</li>
  <li>Capricciosa - € 6.50</li>
</ul>
```

OTHER TEMPLATE ENGINES

Pug is just one of the many template engines available. Other well-known template engines include:

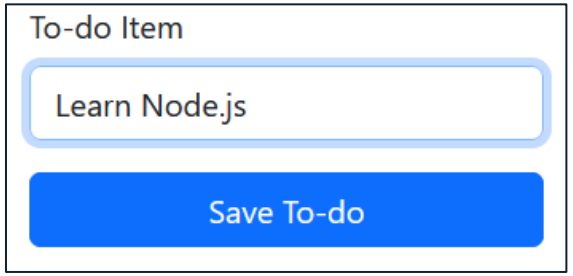
- EJS (formerly Jake): Syntax somewhat similar to JSP/PHP
- SquirellyJS
- Nunjucks
- LiquidJS
- Eta
- Haml (Ruby)

PARSING REQUEST BODIES

PARSING REQUEST BODIES

- Users fill the form on the To-do list page to save new To-do items
- Form is submitted using the **POST** method
- We need to parse the body, to get the param names and their value
- Not as easy as it may seem!

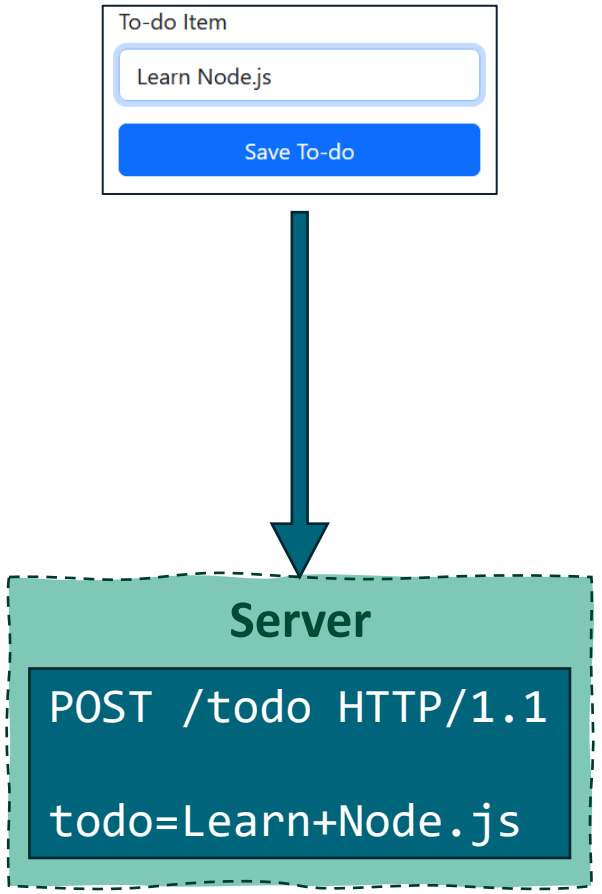
```
form(action="/todo" method="POST")
  label(for="todo") To-do Item
  input(type="text" id="todo" name="todo" required)
  button(type="submit") Save To-do
```




```
POST /todo HTTP/1.1

todo=Learn+Node.js
```

READING THE REQUEST BODY

- The request body might not be available yet when processing the request. Maybe the user has a bad connection, or is uploading some large files, and the body will become available at a later time
- The request object (http.IncomingMessage) extends stream.Readable. This means it generates a **`data`** event when a new chunk of data becomes available in the request body, and an **`end`** event when there is no more data to be consumed from the stream.
- To read the request body, we need to operate in an **asynchronous** way, leveraging these two events



READING THE REQUEST BODY

- Some work is needed to read the body

```
function parseRequestBody(request){
  return new Promise((resolve, reject) => {
    let body = [];
    request
      .on('data', chunk => { body.push(chunk); })
      .on('end', () => {
        body = Buffer.concat(body).toString();
        // at this point, `body` has the entire request body stored as a string
        let data = parseRequestBodyString(body);
        resolve(data);
      });
  });
}
```

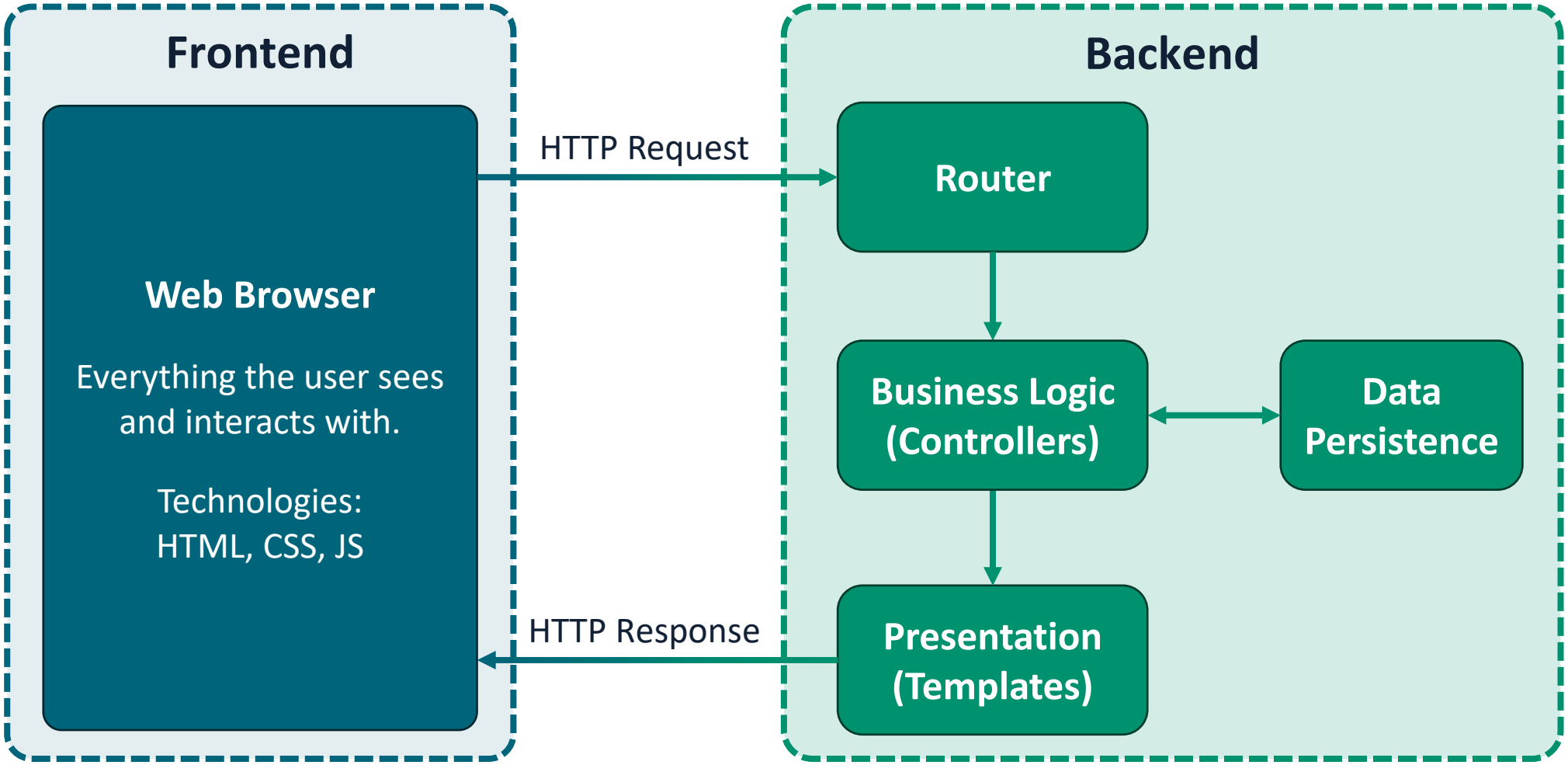
PARSING THE BODY

- Once we read the body as a string, it is just a matter of splitting it to get parameter names and values
- Recall that the body is a string of the form **p1=v1&p2=v2&...**

```
function parseRequestBodyString(body) {
    let data = {};
    let slices = body.split("&");
    for (let slice of slices) {
      let paramName = slice.split("=")[0];
      paramName = decodeURIComponent(paramName.replace(/\+/g, " "));
      let paramValue = slice.split("=")[1];
      paramValue = decodeURIComponent(paramValue.replace(/\+/g, " "));
      data[paramName] = paramValue;
    }
    return data;
  }
```

Recall that the body is URL encoded!
E.g.: «Prove that P=NP» is encoded as «Prove+that+P%3DNP»!

ANATOMY OF A TYPICAL WEB APPLICATION



LET'S LOOK AT THE CODE

- Live demo time!
- We will take a look at the entire to-do list web app
- Source Code is available in the Course Materials on Teams
  - You should check the code out, and try to run (and debug) the web app

MINI–ASSIGNMENT

- Download and run the To-do List Web App we discussed in this lecture
- Feel free to try and add some new feature to our Node.js To-do app
  - For example, you may implement the possibility of deleting To-do items
  - Think about it and try to come up with a solution, using the tools and approaches we've seen so far!
  - **Some hints**:
    - a possible way to do this is to add a new path in our app (e.g.: `/delete`)
    - you may pass an `id` of the to-do item to delete as a query parameter (e.g.: `/delete?id=X`)
    - you can specify the delete url for each to-do item when displaying the list in /todo, so that when a user clicks on a given to-do item, that item gets deleted. Alternatively, you may add a dedicated delete link for each to-do item!

REFERENCES

- **Web Templating**
  Wiki page from the EduTech wiki hosted at the University of Geneva, CH
  Available at https://edutechwiki.unige.ch/en/Web_templating and archived here.
- **Single page apps in depth**
  By Mikito Takada
  Available at http://singlepageappbook.com/ and on GitHub
  Relevant parts: Templating: from data to HTML (direct link here)
- **Node.js v.20.X (LTS) documentation**
  Available at https://nodejs.org/docs/latest-v20.x/api/index.html
  ℹ️ In case you want to learn more about the built-in Node.js methods we used in this lecture.
- **A beginner's Guide to Pug**
  By James Hibbard
  Available at https://www.sitepoint.com/a-beginners-guide-to-pug/ and archived here.