

# An introduction to programming for the web.

Part Two: Server side programming and databases

# What we're going to be doing

1. Intro
2. Setting up a backend
3. Serving templates
4. Making AJAX requests
5. Using a database



# Web Requests

# Recap - What is a web request?

When you open a webpage in your browser, it is fetching that content from a server somewhere via a protocol called HTTP.

The browser sends a request to the server

- The request has a URL, e.g. “http://www.someapp.com/login”

- And some headers

And the server replies with a response

- The response has also got headers

- And a body - usually the contents of a file.

- One of the headers: “Content-Type” tells the browser what to do with the contents of the body

# Recap - What program is running on the server?

On our client we're using a web browser. But the program on the server could be anything, so long as it replies using HTTP. (You *could* use almost any programming language)

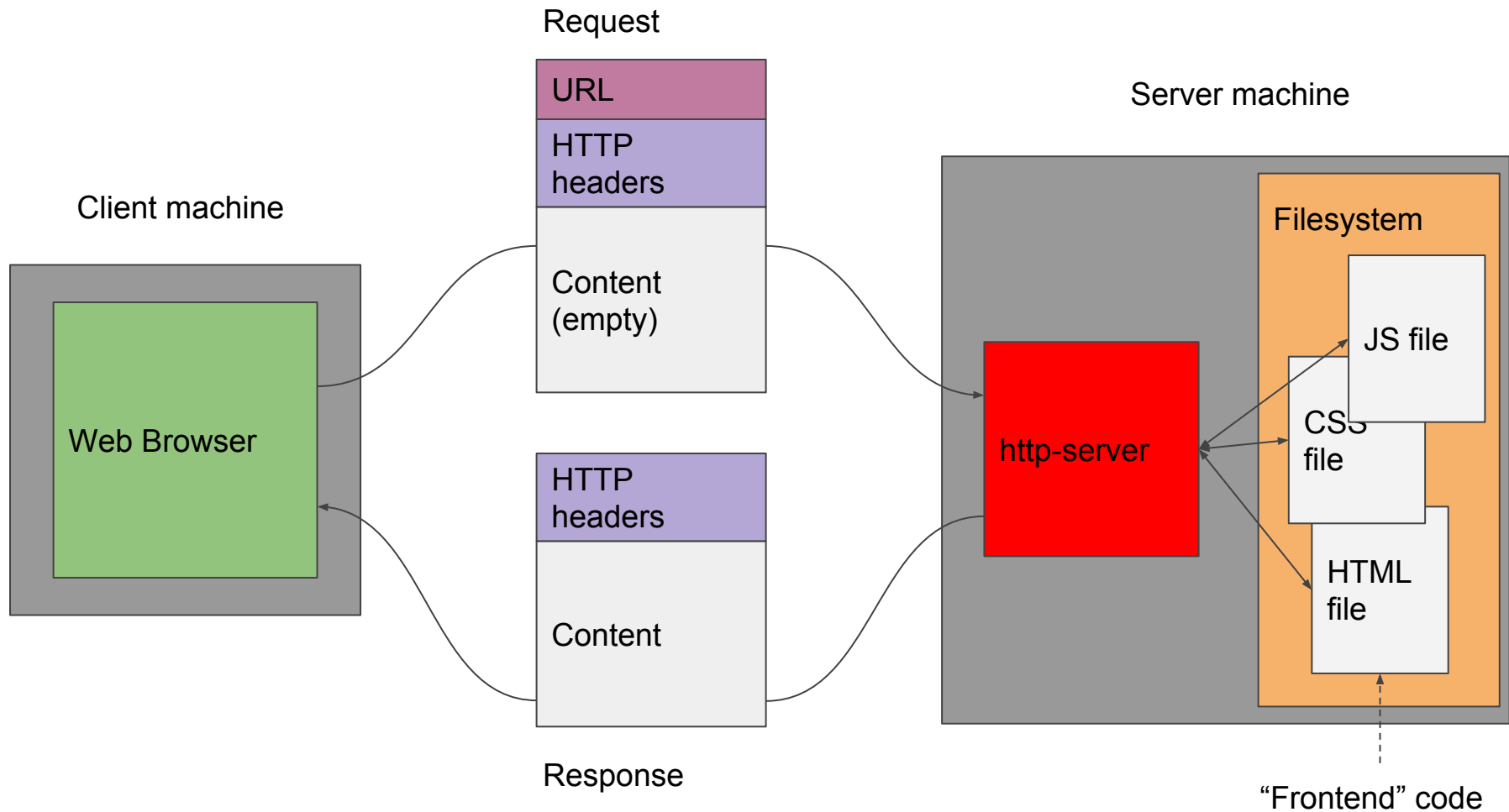
In our case, we want a typical static file webserver. (our http-server module)

We want it to map URLs to files. E.g.

localhost:8080	}	>> C:\Users\JohnDoe\Documents\App\index.html
localhost:8080/index.html		

localhost:8080/an-image.png	>> C:\Users\JohnDoe\Documents\App\an-image.png
-----------------------------	--

(If you write your own program to run on the server, it's called a "backend". Many languages have libraries for serving web content, such as: Express for NodeJS, Flask and Django for Python, ASP.NET for C#, Rails and Sinatra for Ruby)



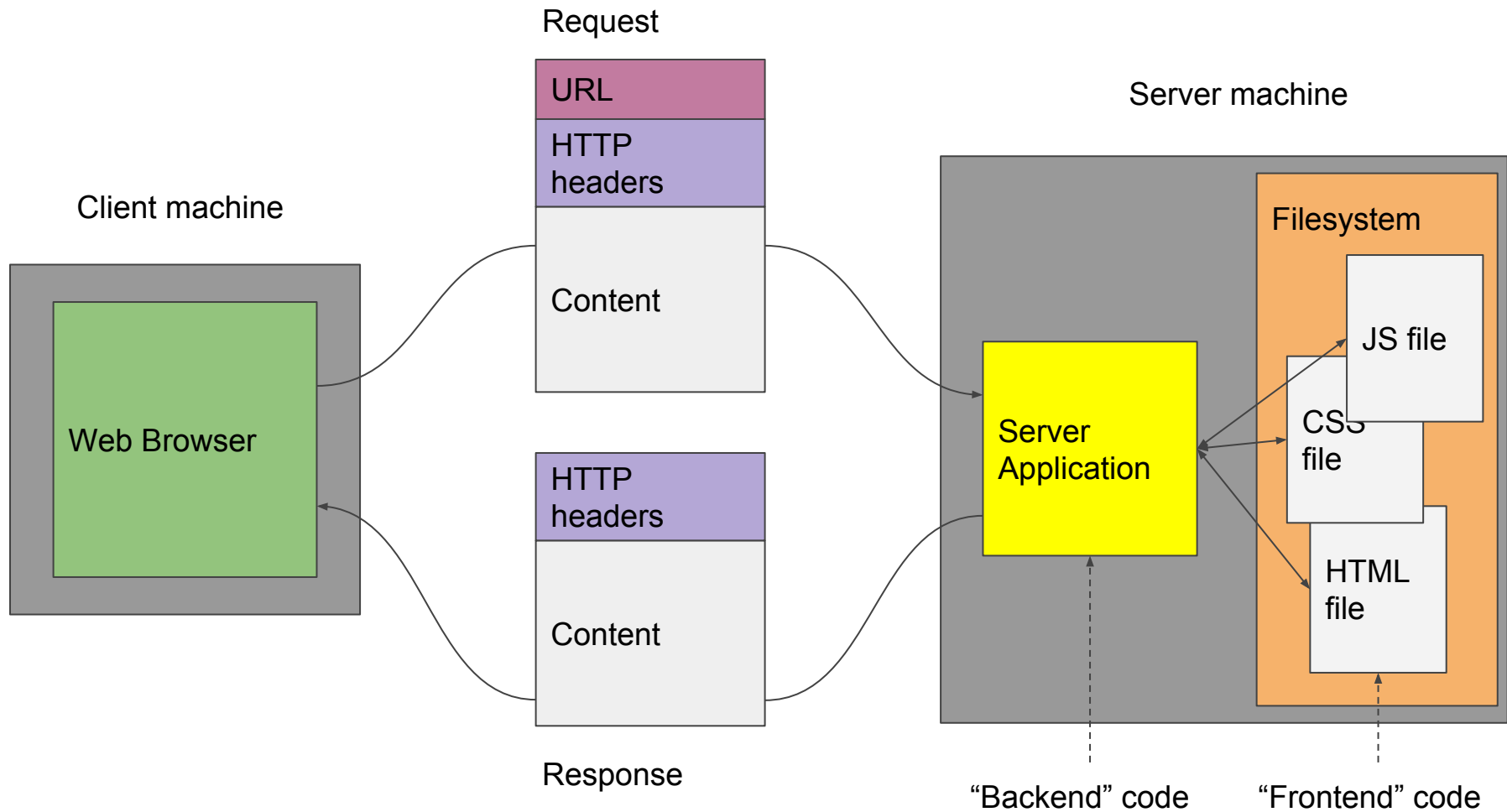
# Where does the backend application go?

Replaces the 'http-server' module

The HTTP protocol is exactly the same, so the web browser won't know anything has changed

But, now our server can reply with *anything*

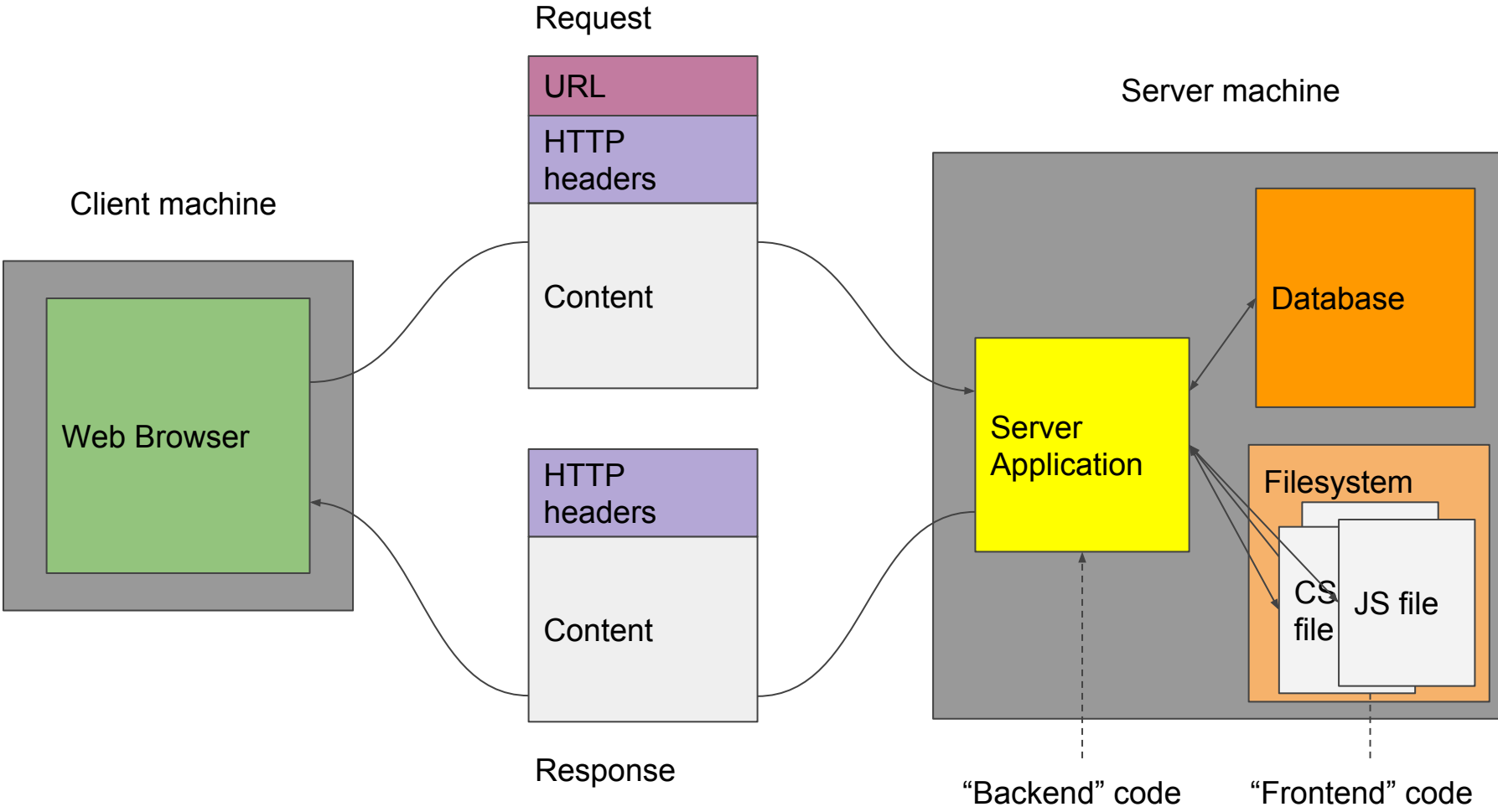






# What is the role of the backend?

- Listening for HTTP requests and replying with HTTP responses
- Routing request URLs to the right content:
  - Static (from a file) content (HTML, CSS, JS, images, data, PDFs, ...)
  - Dynamic Content (Dynamically generated HTML/data, or anything from above)
- Persisting application/website state. E.g. login/logout, Guestbook entries
  - In memory
  - In a file
  - Using another program: A database
- Dynamically generating content based on parameters, such as the state.



# Choice of Backend Language

# Not PHP

It is very easy to start writing PHP code.

Instead of writing a server application, you add PHP syntax to your HTML files. (and change them to <filename>.php)

You use a server application such as Apache to evaluate these templates before it sends them back to the client - just like it would with a regular HTML file.

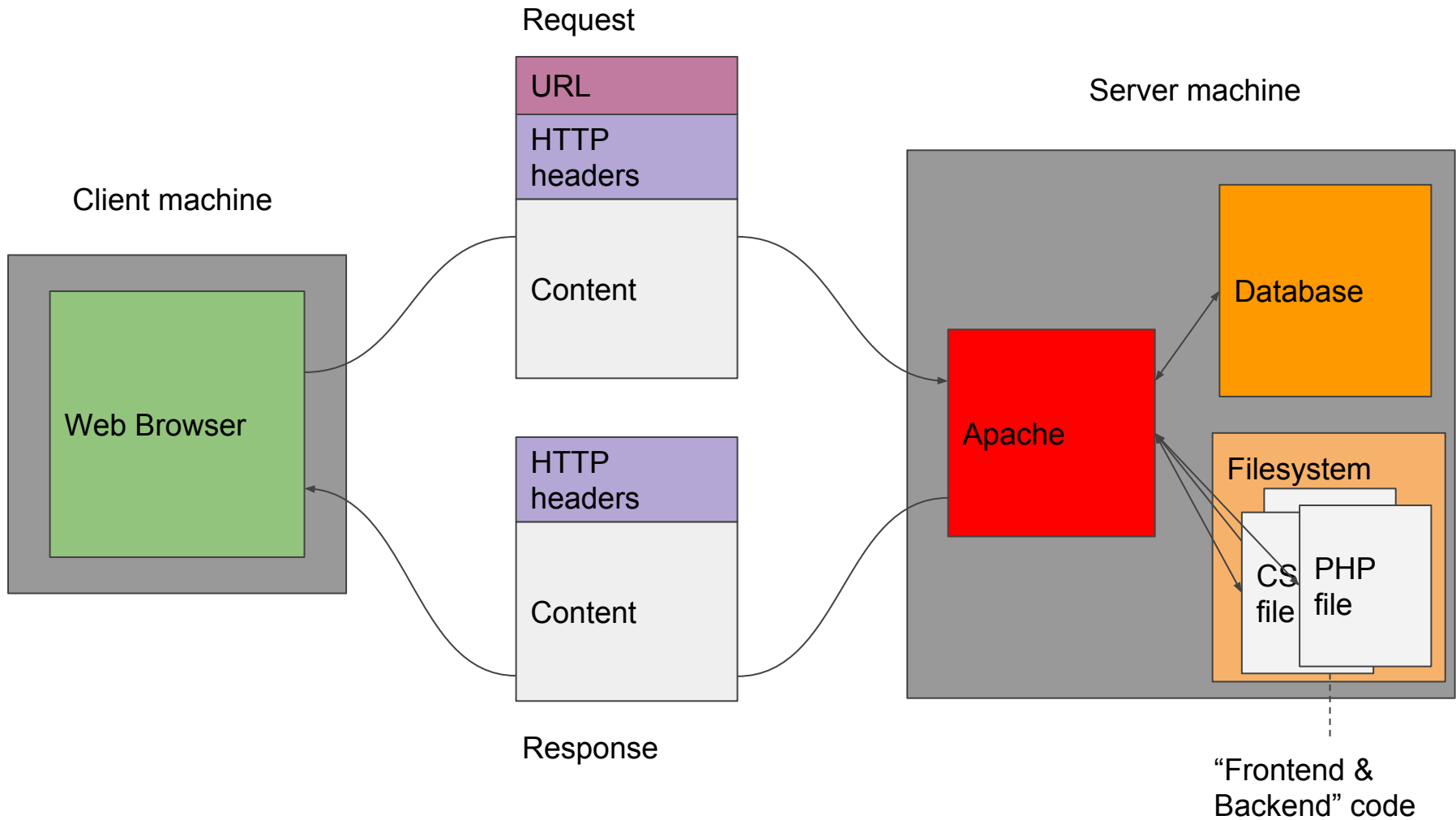
This is *okay* but it's not designed for large complicated server applications. (Although many people still decide to use it for them)

index.php or index.html

```
<html>
<body>
<form action="welcome_get.php" method="get">
Name: <input type="text" name="name"><br>
E-mail: <input type="text" name="email"><br>
<input type="submit">
</form>
</body>
</html>
```

welcome\_get.php

```
<html>
<body>
Welcome <?php echo $_GET["name"]; ?><br>
Your email is: <?php echo $_GET["email"]; ?>
</body>
</html>
```



# What then?

You will find that almost *any* language you have heard of has had a webserver or webserver library written in it. Just try googling “<language> webserver”!

However, for various reasons, the most commonly used ones are:

- Python
  - NodeJs
  - C#
  - Java
  - Ruby
- 

# Setting up

# Hello World

Create a new folder.

Inside that folder create 2 files:

- index.js
- package.json

In a terminal window in this folder  
run:

`node index`

Inside your index.js file:

```
console.log('Hello, World!');
```

Inside your package.json file:

```
{  
  "name": "web-workshop-part2",  
  "version": "1.0.0",  
  "main": "index.js",  
}
```



```
npm install --save express
```

# Express

```
var express = require('express');  
var app = express();  
  
app.get('/', function (req, res) {  
  res.send('Hello, World!');  
});  
  
app.listen(3000, function () {  
  console.log('Example app listening on port 3000!');  
});
```




# Counter

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

var counter = 0;

app.get('/', function (req, res) {
  counter = counter + 1;
  res.send('The counter is at ' + counter);
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```



# Routing

...

```
app.get('/', function (req, res) {  
  counter = counter + 1;  
  res.send('The counter is at ' + counter);  
});
```

```
app.get('/is-even', function (req, res) {  
  if(counter % 2 === 0) {  
    res.send('Yes');  
  } else {  
    res.send('No');  
  }  
});
```

...



# Static Files

Make a directory called 'public', put an html file in there called index.html

...

```
var counter = 0;
```

```
app.use(express.static('public'));
```

```
app.get('/', function (req, res) {  
  counter = counter + 1;  
  res.send('The counter is at ' + counter);  
});
```

...



```
npm install --save mustache-express
```

# Templating

```
var express = require('express');
var mustacheExpress = require('mustache-express');
var app = express();

// Configure out view engine
app.engine('html', mustacheExpress());
app.set('view engine', 'mustache');
app.set('views', __dirname + '/views');

app.get('/', function (req, res) {
  counter = counter + 1;
  res.render('index.html', { counter: counter });
});

...
```



# Templating

```
var express = require('express');
var mustacheExpress =
  require('mustache-express');
var app = express();

// Configure out view engine
app.engine('html', mustacheExpress());
app.set('view engine', 'mustache');
app.set('views', __dirname + '/views');

app.get('/', function (req, res) {
  counter = counter + 1;
  res.render('index.html', {counter...
});

...
```

Create a new folder called 'views' and create an index.html there:

```
<html>
  <head>
    <title>Guestbook App</title>
  </head>
  <body>
    Hello World.
    The Counter is {{ counter }}
  </body>
</html>
```





# Using templating for our app

# What are the advantages of using templating?

- When we refresh the page, the list is not cleared
- Now the HTML that describes how the comment should look is stored next to the HTML it belongs with
- There is no HTML embedded within strings in our JavaScript code



# What are the disadvantages of using templating

- When we submit the form, the page refreshes
- The page does not update if another visitor of the page writes a comment
- Code in the 'views' folder is both tied to the backend and the frontend





# AJAX

# AJAX

## Asynchronous JavaScript and XML

You can send an HTTP  
request from within  
JavaScript, without  
reloading the page!

The code on the right is an  
example. We're going to  
work it into our app.

```
function doAsyncRequest() {  
    var request;  
  
    if (window.XMLHttpRequest) {  
        // code for IE7+, Firefox, Chrome, Opera, Safari  
        request=new XMLHttpRequest();  
    }  
    else {  
        // code for IE6, IE5  
        request=new ActiveXObject("Microsoft.XMLHTTP");  
    }  
    // Assign new anonymous function to request.onreadystatechange  
    request.onreadystatechange=function () {  
        if (request.readyState == 4 && request.status == 200){  
  
            // Now we can do something with request.responseText:  
            document.getElementById("myDiv")  
                .innerHTML=request.responseText;  
  
        }  
    }  
  
    request.open("POST", "/", true);  
    request.setRequestHeader("Content-type", "application/x-www-form-urlencoded");  
    request.send("name=Steven&comment=SomethingProfound");  
}
```

# How does AJAX improve our app?

Now we can use JavaScript to send requests to the server.

- We can use AJAX to post a new entry without reloading the page
- We could set the JavaScript to check every so often for new entries.

What should we do when we discover new entries?

- Use JavaScript to tell the page to reload?
- Fetch the new entries in the background and update the page without reloading?



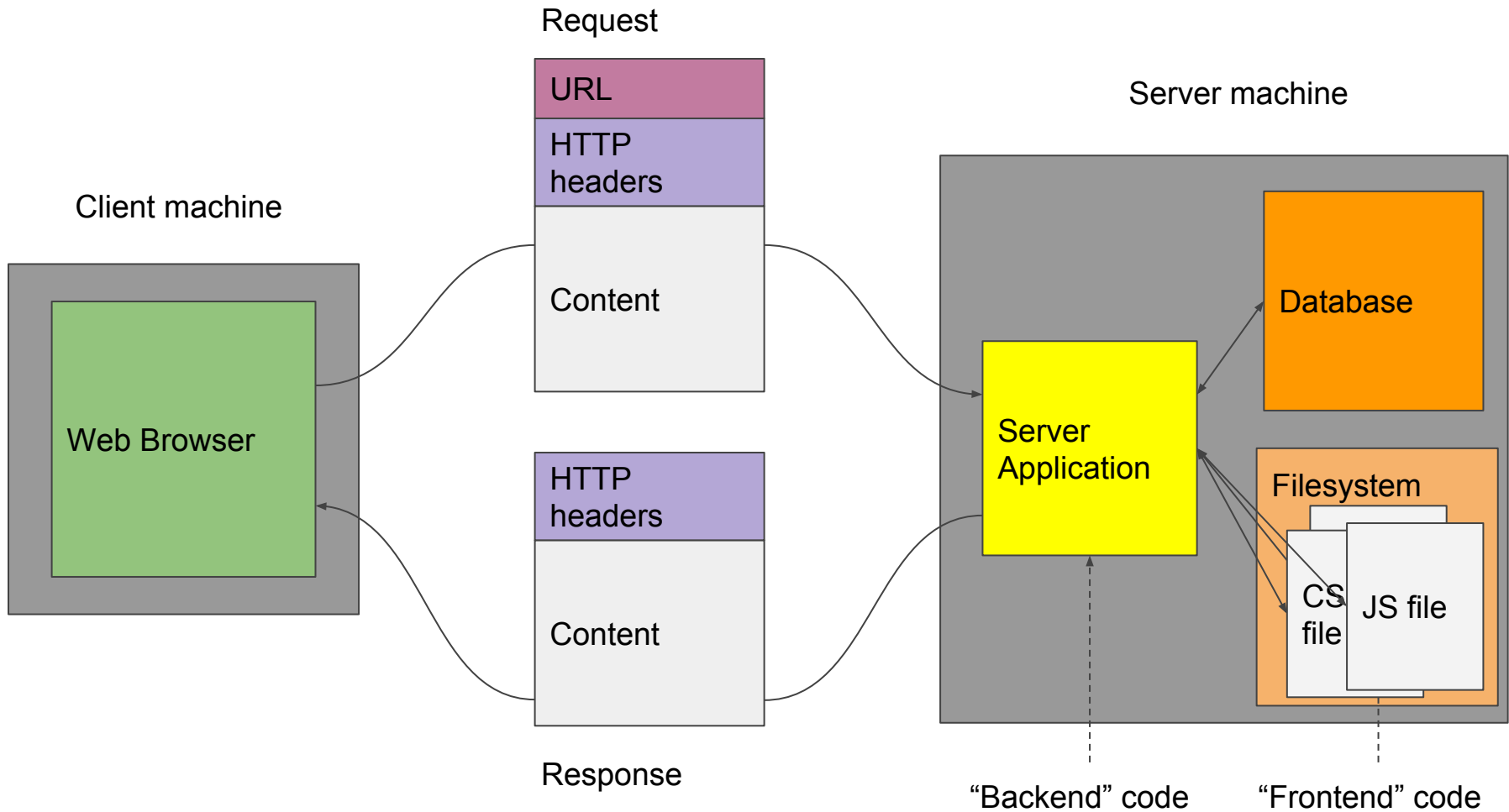
# What about not using templates at all?

If we use AJAX calls to fetch new entries, why not use it to fetch all the entries?

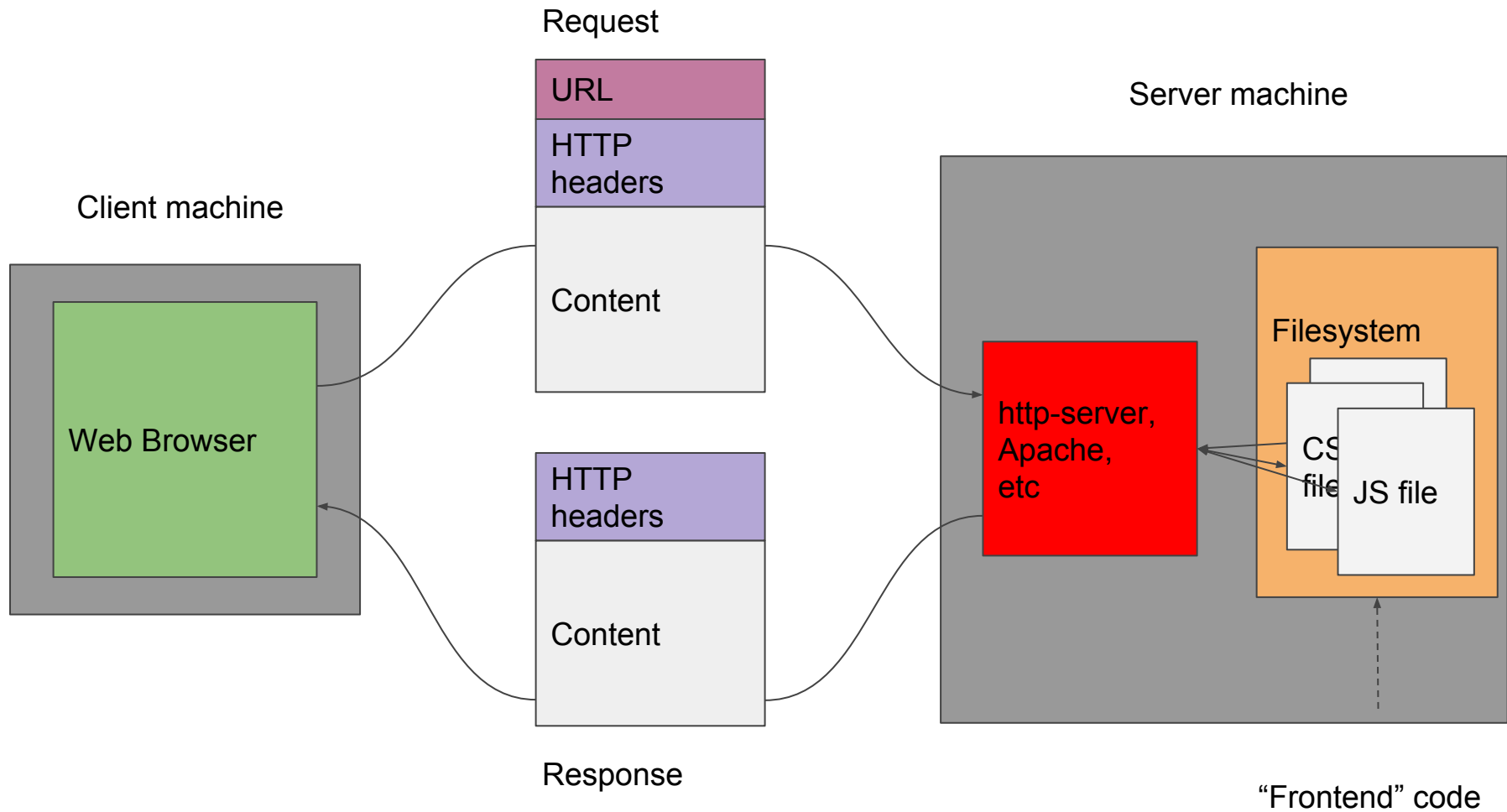
We could have a clear separation of frontend and backend code:

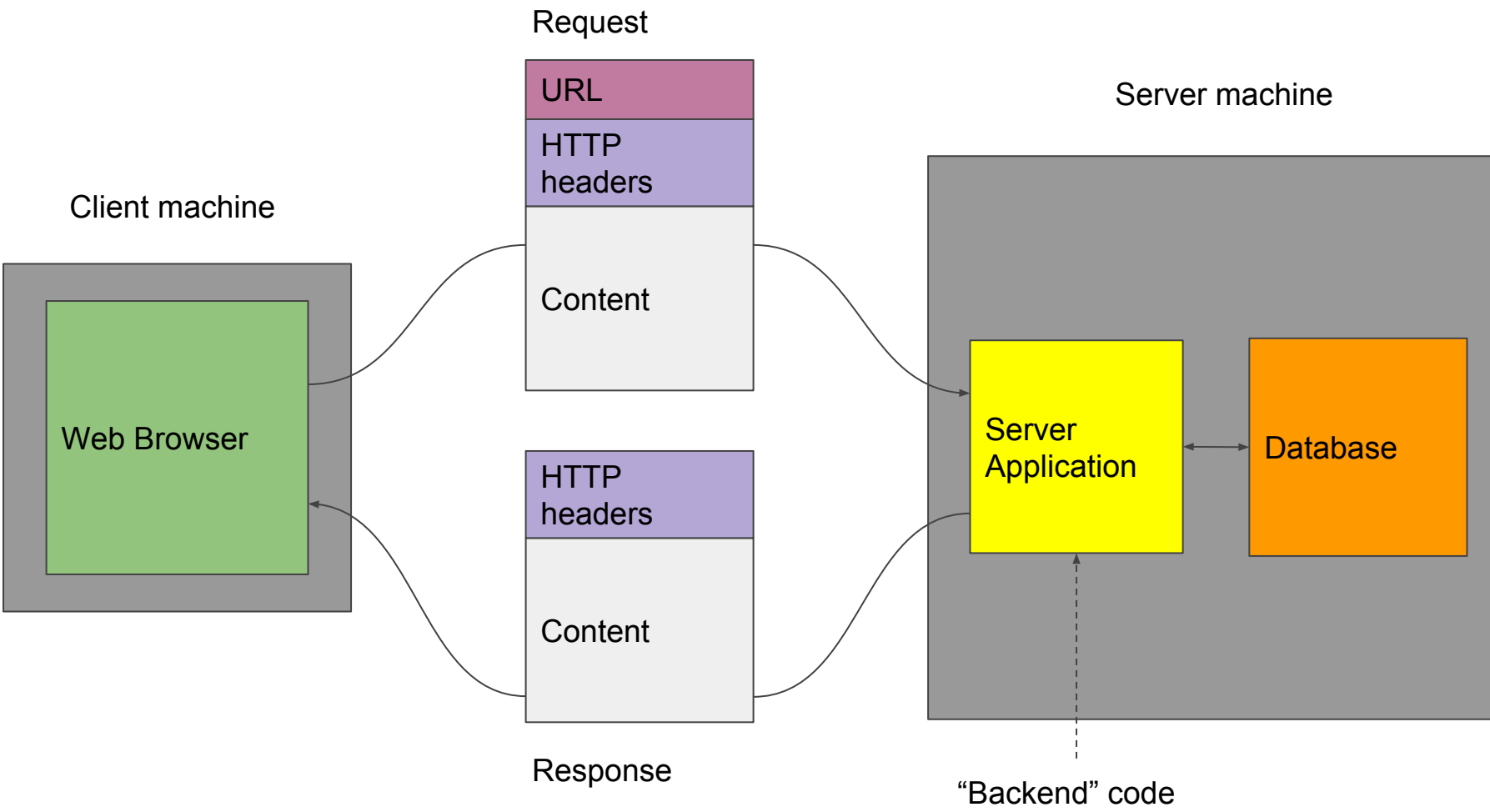
- Our index.html would become a static file again, and be put in the 'public' folder
- The backend routes would only send back JSON data, never HTML, i.e. the backend would not know which page needed what data, or how it was being displayed











# A backend that just deals with data

The backend only responds with JSON.

So only AJAX/background calls ever interact with it.

Users will never see the routes, but developers will.

So the routes should make sense to someone building frontend AJAX code who wants to send requests to the backend.

Or they could be building *any* kind of client app

There's already a convention, it is called REST  
(Representational state transfer)

# Solving the problem of embedded HTML

With a separated frontend and backend, we would still get the problem of having to use JavaScript to append new HTML elements to the document.

The typical way web developers solve this is by using a library that makes this code nicer. There are **tons** of libraries to choose from, and they often solve the problem in very different ways.



# Databases