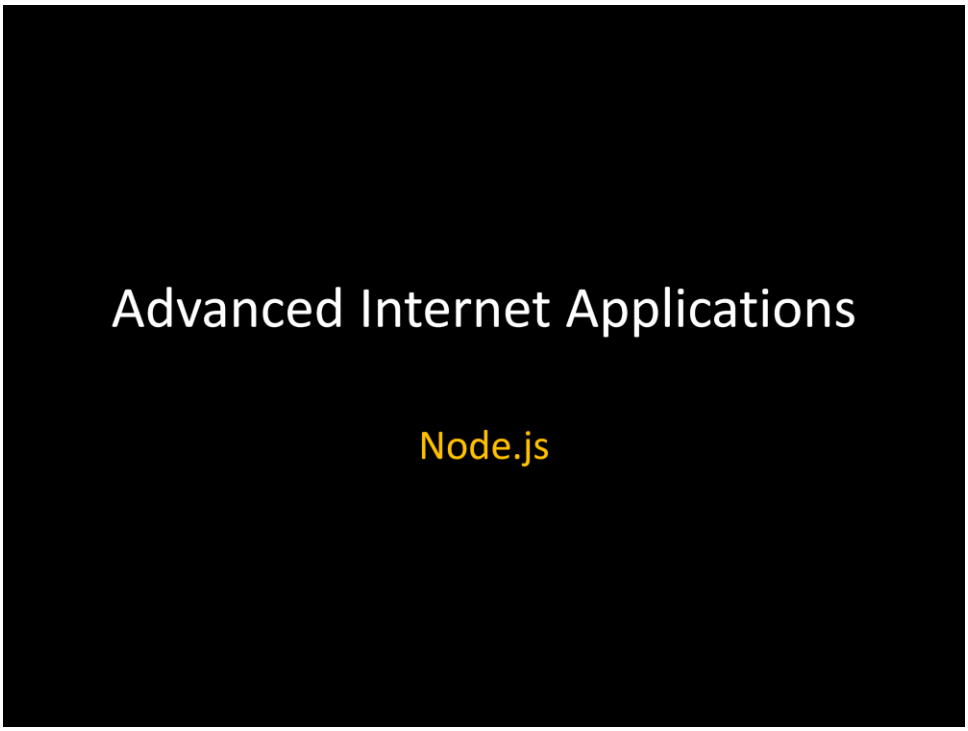
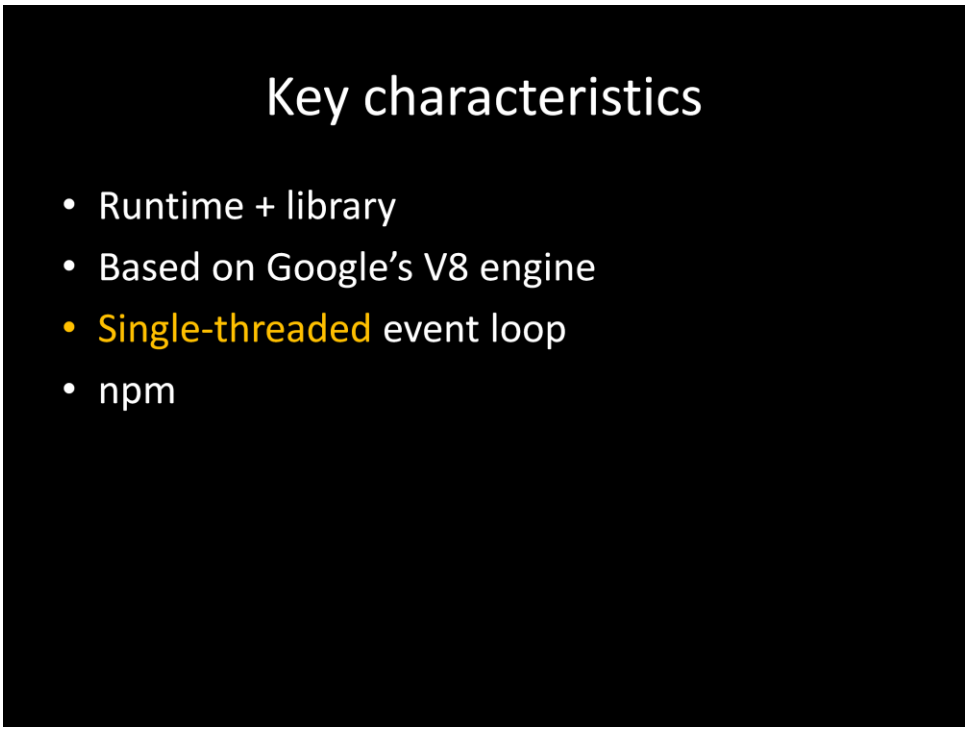


Welcome to another meeting on Advanced Internet Applications! Today, you'll finally get to know a server-side technology (if you haven't noticed, so far we've been only dealing with front-end). We'll start with Node and the main reason for that is that it is naturally low-level, i.e., you can work very closely to the HTTP protocol and I think it is appropriate for an advanced course. Also, in Node you write in JavaScript which is a language you should be pretty familiar with by now. Finally, it's extremely popular right now, so I think it'll equip you well for your future. Let's dive in!

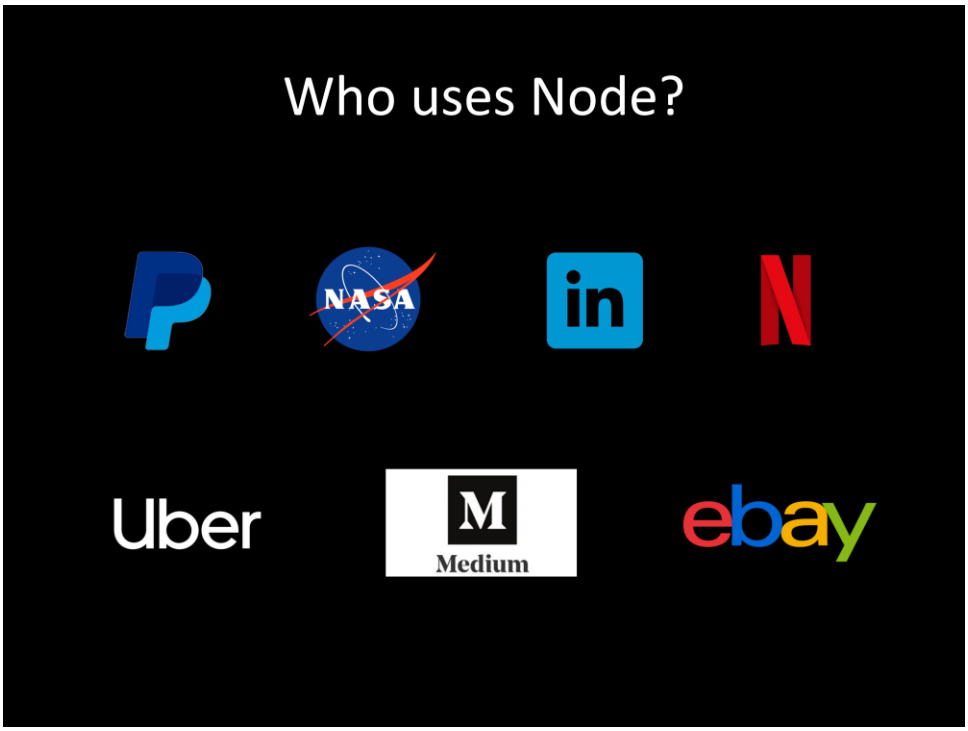


I know you've already had a lecture on Node.js, so I'll try to keep this short.

Just a quick reminder.

The event loop is single-threaded, so you shouldn't block it or else the server will immediately choke and stop responding. In Node, you do any heavier lifting in terms of processing asynchronously and rely on callbacks (promises will come in handy). Just remember – the fact that the event loop is single threaded doesn't mean that everything else is executed in a single thread! When you read something from a database (and do it in a non-blocking way) it will be executed in a separate thread.

So, you can view the event loop as a dispatcher of stuff to do.



O yeah – it is popular!

Now, let’s go and write our first app.

That’s right! Who said you have to write a server in Node. Here’s a simple app for you, which… well… I hope you can figure out what it’s doing…

Hello, World!

```
const http = require('http');

const server = http.createServer((request, response) => {
  response.setHeader('content-type', 'text/plain');
  response.end('Hello, World!');
});

server.listen(3000, () => {
  console.log('Server listening at http://localhost:3000/');
});
```

… however, Node is most widely known for its server capabilities so we'll focus on this part (+ this will be more appropriate for Advanced Internet Applications course…).

To write a simple server you'll need the http module (modules are packages in Node). The silde shows a „Hello, World!" application a bit more suited to the subject of this course.

Congratulations!
You now know how to create a simple server and write some response for the incoming requests.

Handling requests

```
const http = require('http');
const url_parser = require('url');

const server = http.createServer((request, response) => {
  const { url, method, headers } = request;

  const parsedUrl = url_parser.parse(url, true);

  response.setHeader('content-type', 'text/plain');

  response.write(`Your browser: ${headers['user-agent']}\n`);
  response.write(`HTTP method:  ${method}\n`);
  response.write(`id: ${parsedUrl.query.id}`);
  response.end();
});

server.listen(3000, () => {
  console.log('Server listening at http://localhost:3000/');
});
```

This example shows a bit more complex request handling. As you can see, request allows you to access things like the request url, the headers sent with the request and the HTTP method used. We're also using the url module, which allows us to conveniently access all parts of the url, like the protocol, host, pathname, and query. In this example, we are expecting an „id" query parameter in the request url.

Handling requests

```
const server = http.createServer((request, response) => {
  const body = [];

  request.on('error', (error) => {
    console.error(error.stack);
  });

  request.on('data', (chunk) => {
    body.push(chunk);
  });

  request.on('end', () => {
    const fullBody = Buffer.concat(body).toString();

    response.setHeader('content-type', 'text/plain');
    response.end(`This was posted: ${fullBody}`);
  });

});
```

Here, you can see how to process larger requests. Since, as we've already mentioned, the event loop is single threaded, if the data sent with the request is particularly large, we don't want to choke our server on it, so it may be a good idea to spread this data into chunks and read them only once they are available. This way, the event loop stays nice and responsive while we handle this large request.

Routing

```
const server = http.createServer((request, response) => {
  const path = url_parser.parse(request.url).pathname;

  if (path === '/') {
    response.writeHead(200, { 'Content-Type': 'text/plain' });
    response.write('Hello world');
  }
  else if (path === '/about') {
    response.writeHead(200, { 'Content-Type': 'text/plain' });
    response.write('If you change nothing, nothing will change.');
  }
  else {
    response.writeHead(404, { 'Content-Type': 'text/plain' });
    response.write('Error page');
  }

  response.end();
});
```

Often in web apps you'll want some sort of routing, i.e., a mechanism which will direct the processing of the requests to various functions depending on the path in the request. In this slide you can see how you could achieve this effect by hand. As you see – it's nothing fancy. Just a bunch of if statements depending on the path… In a larger project we would naturally tidy this example up and divide it into multiple functions and probably even files, but the mechanism would remain the same.

As you can see, all of this is prettly low level stuff (at least by today's standards). If you want routing, you have to create one on your own. You want to simply serve files from the server? Well – you have to write that on your own as well! And that's great!

Howerver, it's not that often that you actually need to have a totally customized control over every aspect of your server and some of this stuff could be solved externally.

Enter express…

Express

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

app.all('/', (request, response) => {
  response.send('Hello, World!')
});

app.get('/book/:id', (request, response) => {
  response.send(`Sending book with id ${request.params.id}`)
});

app.post('/book', (request, response) => {
  response.send('New book created.')
});

app.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

Express is an awesome, lightweight module which takes care of most of the mundane stuff you usually have in a server anyway. In the example above you can see that you can conveniently write separate responses for different request methods and for different paths or even templates (notice the :id parameter in the get request and how simply you can access it). Sending a standard response is easier as well!

As you can see, this mechanism is much cleaner out of the box compared to the one we created on the previous slide.

However, we can still clean it up even further…

Express – routing

```
const express = require('express');
const router = express.Router();

router.all('/', (request, response) => {
  response.send('Hello, World!')
});

router.get('/book/:id', (request, response) => {
  response.send(`Sending book with id ${request.params.id}`)
});

router.post('/book', (request, response) => {
  response.send('New book created.')
});

module.exports = router;
```

In order to clean the routing of your app it is a good idea to create separate routing files, just like the one in this example.

Notice the additional step of creating a router in the second line and later exporting it out.

After you're done, all you have to do is…

Express – routing

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

const routing = require('./routing');

app.use('/', routing);

app.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

…import the created routing and attach it to your app.

That’s it!

And look how clean it looks right now – mmmmm:-)

Express makes other things easier too.

Express – static files

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

const routing = require('./routing');

app.use('/', routing);

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

app.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

Serving static files is a breeze! Just add this line and you’re good to go. Now all the files in the public folder will be sent upon request.

Express – middleware

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

const routing = require('./routing');

app.use((request, response, next) => {
  console.log(`${request.method} ${request.url}: ${new Date()}`);
  next();
});

app.use('/', routing);

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

app.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

When writing a server you'll often want some sort of pre- and post- request filters which should execute something before or after each request. Express has that covered for you in a form of middleware, which is… well… exactly that – a way of injecting some actions before or after the actual request processing. A good example which should probably appear on any web server is logging. Here's a very simple example that should get you started with this.

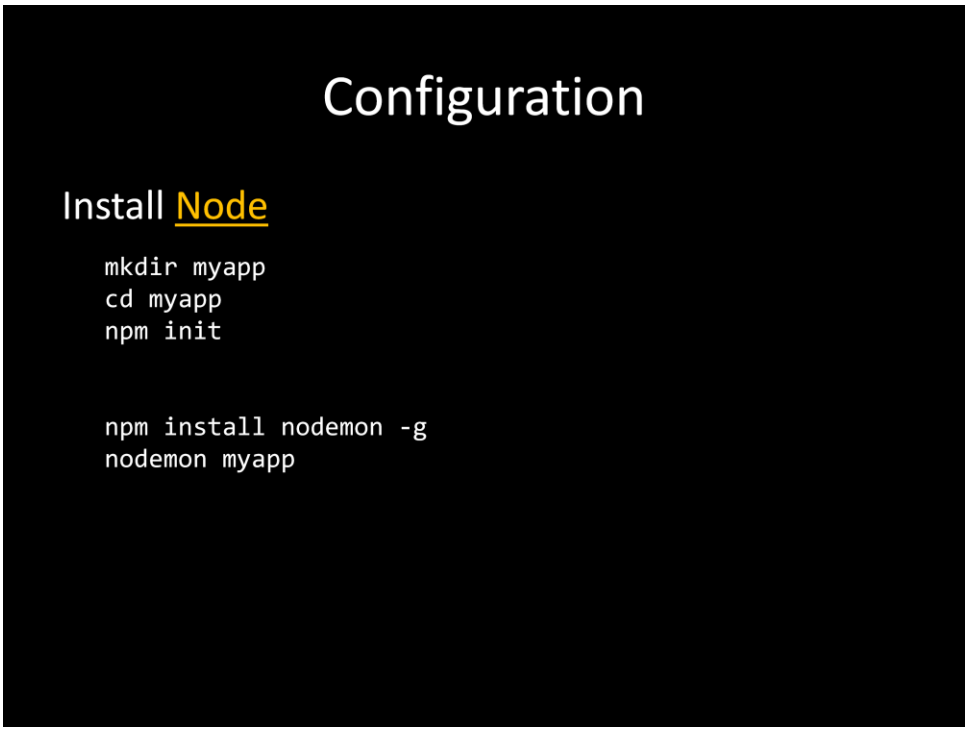
Express – templating

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

app.set('view engine', 'ejs');

app.get('/book', (request, response) => {
  response.render('books/index', { 'name': request.query.name });
});

app.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

views/books/index.ejs

```
...<body>
    <h1>Welcome to my online store</h1>
    <form action="/book" method="get">
        <input type="text" name="name">
        <input type="submit" value="Read">
    </form>
    <% if(name) { %>
        <div>You're reading <%=name%></div>
    <%}%>
</body></html>
```

As you have hopefully seen so far, Node makes it very easy to write back-end for you web applications. You can than proceed with using it as a REST API and build a React front-end app on top of it. However, more ofte than not we will just want to make a regular web application, not a single-page one. In such cases, we'll just need an easy way to dynamically generate html documents or – even more conveniently – dynamically fill pre-defined html templates with data. This too you can do by hand, but it is far easier to use one of many templating engines (https://colorlib.com/wp/top-templating-engines-for-javascript/). The one I find the easiest to use is EJS (Embedded JavaScript) and you can find it in the example on this slide. As you can see, it works by allowing you to embed JavaScript into your html template – just like the name suggests:-)



In order to start writing apps in Node you obviously need to install it first, but afterwarts, creating a new project is very easy. Just follow these three commands, replace „myapp" with whatever you want your app to be called, and that's it!

I also recommend using nodemon which makes running and updating the app much easier.

The –g switch installs nodemon globally, so you won't have to install it separately in every project.

nodemon myapp starts the app and listens to any changes you make to it. Once any file is changed, the browser will immediately refresh, so you've got that covered for you.

And that's it!

You've made it to the top of Node-mountain.

15 slides up, 15 slides down.

Now there's nothing left to do but roll out the red carpet for you my friend!