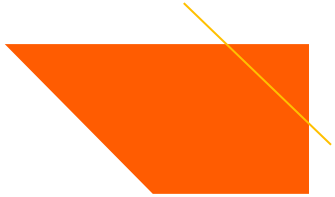
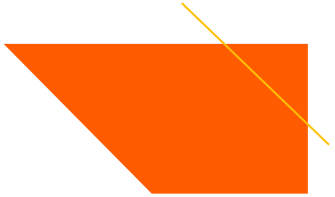
Web Application Development

COSI 152A

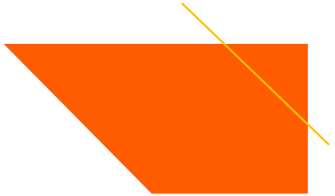


Serving External Files



Serving External Files

- Say goodbye to plain-text responses as you learn how to
  - serve HTML files
  - serve assets such as client-side JavaScript, CSS, and images

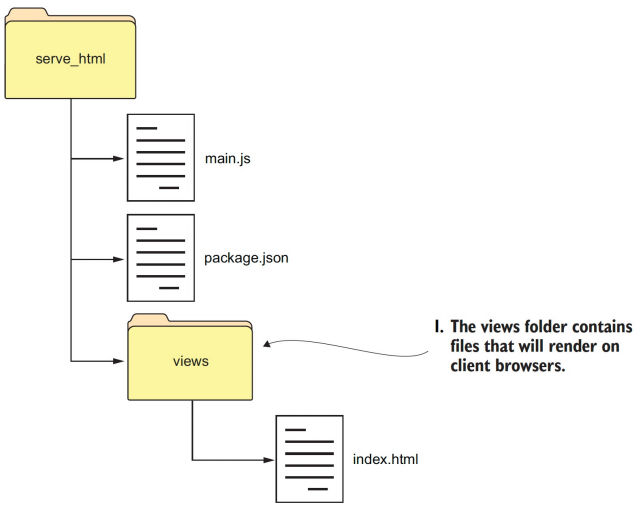


Why you need HTML files?

- Building static sites using HTML snippets can get cumbersome and clutter your main.js file.
  - Instead, build an HTML file that you'll use in future responses.
  - This file lives within the same project directory as your server.

Project Structure

- A typical project file structure
  - all content you want to show the user goes in the views folder
  - all the code determining which content you show goes in the main.js file.



Application structure with views

Build the HTML file

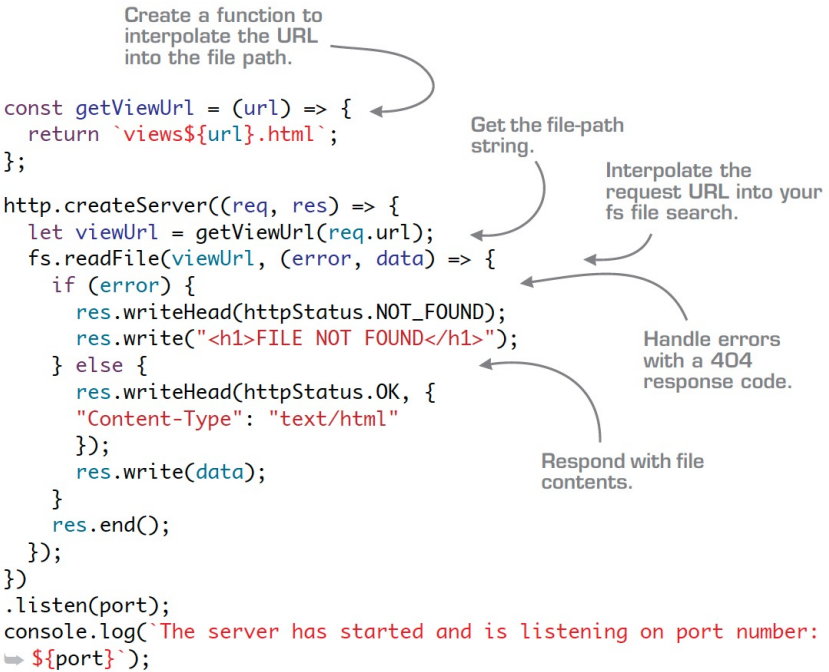
- Add the HTML boilerplate code to index.html
- The client can see this page rendered in a browser only if
  - The server is able to interact with the filesystem
  - The server can access and read the index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Home Page</title>
  </head>
  <body>
    <h1>Welcome!</h1>
  </body>
</html>
```

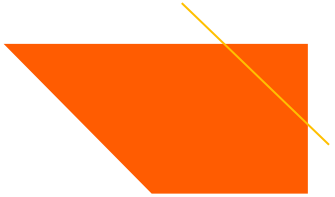
Add a basic HTML structure to your views.

Serving Static Files with the fs Module

- fs module interacts with the filesystem on behalf of your application.
  - Use fs module and routing to dynamically read and serve files via main.js




```
const getViewUrl = (url) => {
  return `views${url}.html`;
};

http.createServer((req, res) => {
  let viewUrl = getViewUrl(req.url);
  fs.readFile(viewUrl, (error, data) => {
    if (error) {
      res.writeHead(httpStatus.NOT_FOUND);
      res.write("<h1>FILE NOT FOUND</h1>");
    } else {
      res.writeHead(httpStatus.OK, {
      "Content-Type": "text/html"
      });
      res.write(data);
    }
    res.end();
  });
})
.listen(port);
console.log(`The server has started and is listening on port number:
➥ ${port}`);
```

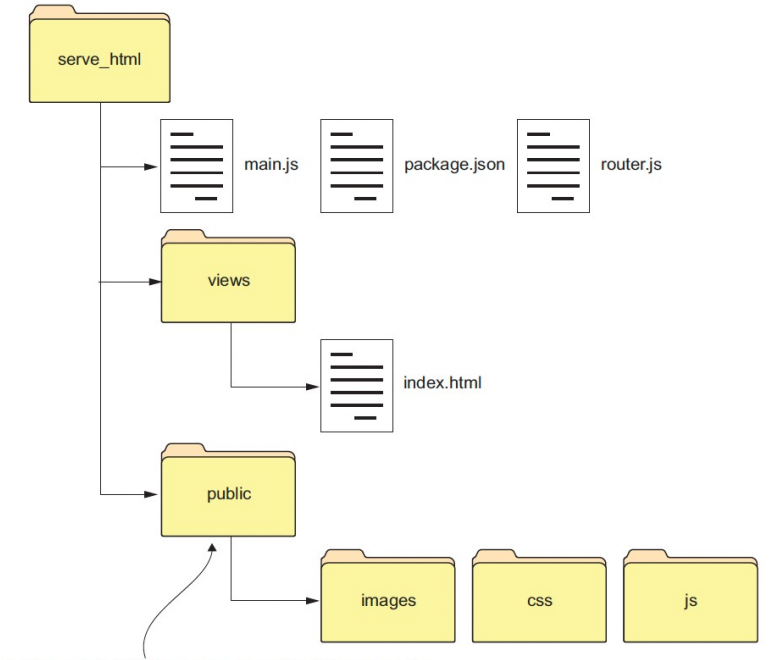


Serving Assets

- Assets are the images, stylesheets, and JavaScript that work alongside your views on the client side.
- Like your HTML files, these file types, such as .jpg and .css, need their own routes to be served by your application.

Serving Assets

- Arrange your assets in a way tat they’re easier to separate and serve:



1. The public folder can be organized to separate your most common assets served to the client.

Serving Assets

- Refine your routes to better match your goal.

```
const sendErrorResponse = res => {
  res.writeHead(httpStatus.NOT_FOUND, {
    "Content-Type": "text/html"
  });
  res.write("<h1>File Not Found!</h1>");
  res.end();
};

http
  .createServer((req, res) => {
    let url = req.url;
    if (url.indexOf(".html") !== -1) {
      res.writeHead(httpStatus.OK, {
        "Content-Type": "text/html"
      });
      customReadFile(`./views${url}`, res);
    } else if (url.indexOf(".js") !== -1) {
      res.writeHead(httpStatus.OK, {
        "Content-Type": "text/javascript"
      });
      customReadFile(`./public/js${url}`, res);
    } else if (url.indexOf(".css") !== -1) {
      res.writeHead(httpStatus.OK, {
        "Content-Type": "text/css"
      });
      customReadFile(`./public/css${url}`, res);
    } else if (url.indexOf(".png") !== -1) {
      res.writeHead(httpStatus.OK, {
        "Content-Type": "image/png"
      });
      customReadFile(`./public/images${url}`, res);
    } else {
      sendErrorResponse(res);
    }
  })
  .listen(3000);

console.log(`The server is listening on port number: ${port}`);

const customReadFile = (file_path, res) => {
  if (fs.existsSync(file_path)) {
    fs.readFile(file_path, (error, data) => {
      if (error) {
        console.log(error);
        sendErrorResponse(res);
        return;
      }
      res.write(data);
      res.end();
    });
  } else {
    sendErrorResponse(res);
  }
};
```

Create an error-handling function.

Store the request's URL in a variable url.

Check the URL to see whether it contains a file extension.

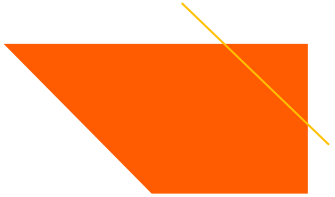
Call readFile to read file contents.

Customize the response's content type.

Look for a file by the name requested.

Check whether the file exists.

- Now your application can properly handle requests for files that don't exist.



Moving the Routes to Another File

- The goal is to make it easier to manage and edit your routes.
  - Any change might affect other routes if all the routes are in an if-else block
    - Adding a new route or removing an existing route
  - It is easier to separate routes based on the HTTP method if list of routes grow
    - /contact path can respond to POST and GET requests relatively.

- As the main.js file grows, your ability to filter through all the code you've written gets more complicated.
- You can easily find yourself with hundreds of lines of code representing routes alone!

Moving the Routes to Another File

- To alleviate this problem, move your routes into a new file called router.js.
- Also restructure the way you store and handle your routes.

```
const httpStatus = require("http-status-codes"),
  htmlContentType = {
    "Content-Type": "text/html"
  },
  routes = {
    "GET": {
      "/info": (req, res) => {
        res.writeHead(httpStatus.OK, {


          "Content-Type": "text/plain"
        })
        res.end("Welcome to the Info Page!")
      }
    },
    'POST': {}
  };

exports.handle = (req, res) => {
  try {
```

Define a routes object to store routes mapped to POST and GET requests.

Create a function called handle to process route callback functions.

```
    if (routes[req.method][req.url]) {
      routes[req.method][req.url](req, res);
    } else {
      res.writeHead(httpStatus.NOT_FOUND, htmlContentType);
      res.end("<h1>No such file exists</h1>");
    }
  } catch (ex) {
    console.log("error: " + ex);
  }
};

exports.get = (url, action) => {
  routes["GET"][url] = action;
};

exports.post = (url, action) => {
  routes["POST"][url] = action;
};
```

Build get and post functions to register routes from main.js.

Moving the Routes to Another File

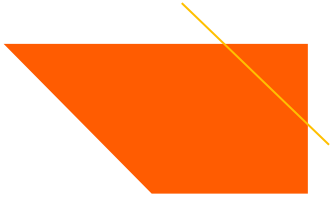
- Import router.js into main.js.

```
const port = 3000,
  http = require("http"),
  httpStatusCodes = require("http-status-codes"),
  router = require("./router"),
  fs = require("fs"),
  plainTextContentType = {
    "Content-Type": "text/plain"
  },
  htmlContentType = {
    "Content-Type": "text/html"
  },
  customReadFile = (file, res) => {
    fs.readFile(`./${file}`, (errors, data) => {
      if (errors) {
        console.log("Error reading the file...");
      }
      res.end(data);
    });
  };

router.get("/", (req, res) => {
  res.writeHead(httpStatusCodes.OK, plainTextContentType);
  res.end("INDEX");
});

router.get("/index.html", (req, res) => {
  res.writeHead(httpStatusCodes.OK, htmlContentType);
  customReadFile("views/index.html", res);
});

router.post("/", (req, res) => {
  res.writeHead(httpStatusCodes.OK, plainTextContentType);
  res.end("POSTED");
});

http.createServer(router.handle).listen(3000);
console.log(`The server is listening on port number:
➥ ${port}`);
```

Create a custom readFile function to reduce code repetition.

Register routes with get and post.

Handle all requests through router.js.



Thank You!