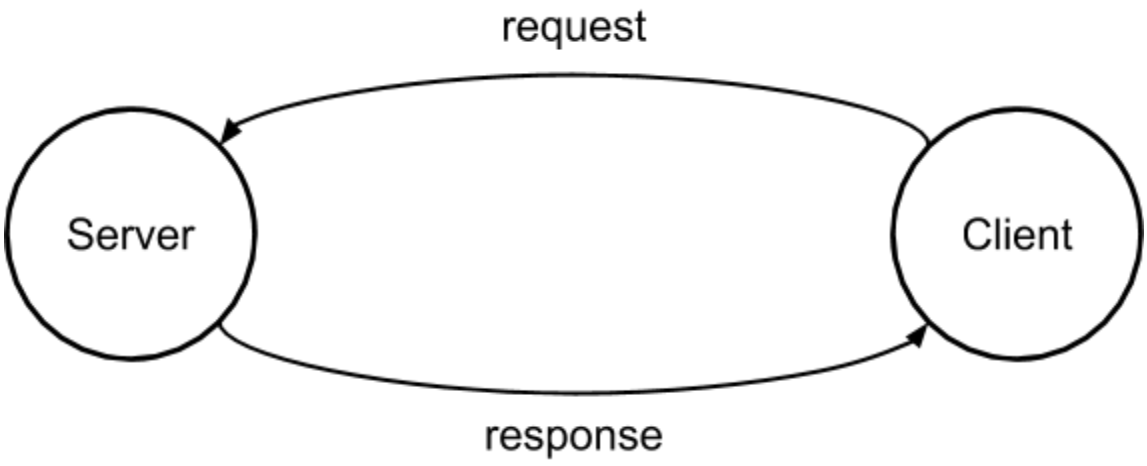
Serving with Express

- ***Express*** is a widely-used framework for creating web applications and APIs with Node.js.
    - Express is so widely-used that it's become the de facto standard server framework for Node.

- To start to understand how Express works, remember that HTTP (the protocol through which web communication occurs) is a request-response protocol:



    - Specifically, under the HTTP protocol, communication occurs in transactions, where the client (e.g. a user's browser) makes a request to a server for some resource (e.g. a specific webpage), and, to satisfy that request, the server sends back to the client a single response.

- We use Express to build HTTP server processes that respond to client requests.

- Specifically, Express allows us to specify a sequence of functions to be called in response to every single HTTP request that the server receives.

- These functions are known as ***middleware***. A middleware function is specifically a function that is passed three arguments:
    - An object representing a single HTTP request (`req`).
    - An object representing the corresponding HTTP response (`res`).

    - A reference to the next middleware function in the in the sequence of functions specified by the server programmer (`next`).

- With access to these three things, a single middleware function can do the following:
    - Execute any code it needs to in order to help process the response.
        - E.g. fetch data from a database, perform a computation based on the request object, log information about the request to the console, etc.
    - Update/modify the request object and/or the response object.
    - Terminate the request-response cycle by sending the response back to the client.
    - Call the next middleware function.

- Through the sequence of middleware functions we specify in our Express-based server, we will enable it to process and respond to every HTTP request the server receives.

- We'll look at a few different ways to incorporate middleware into an Express app in order to provide some basic HTTP server functionality. First, however, let's look at how we simply get an Express server up and running.

Running an Express server

- To be able to use Express, we first need to install it using npm. Note that express is a runtime dependency, so we save it there in our `package.json`:

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

- Once we have Express installed, we can start writing our server. The first thing we'll do is require the Express module and create a new Express application object, which will serve as the representation of our Express server:

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

- Once we have our application object `app`, we can start the server listening for requests. To do this, we specify two things, one of which is optional:
    - The number of a TCP port on which to listen for requests.

    - An optional callback function to be called when the server is successfully started.

- Here's what it might look like to start our server listening:

```
app.listen(3000, function () {
     console.log("== Server is listening on port 3000");
});
```

- Of course, at this point, we haven't told our server what to do when it actually gets a request, so it won't do anything right now. It won't even send a response back for any requests it gets. Let's look at how to add functionality to the server to make it respond to requests. We'll do this by adding middleware functions.

Routing middleware

- ***Routing*** is the process of determining how to respond to a specific HTTP request. In Express, we can specify middleware functions to help our server perform routing.

- Before we look at how to write routing middleware, let's look again at what an HTTP request looks like.

- The specific part of the HTTP request we're interested in when we're talking about routing is the request line, which specifies two things:
    - An HTTP method, e.g. GET, POST, PUT, DELETE, etc.
    - A URI specifying the resource being requested.

- As we saw earlier in the course, the URI specifying the requested resource usually takes the form of an absolute path, which is determined from the URL that initiated the request.

- For example, for the domain flickr.com, users might visit URLs like this:

  https://www.flickr.com/photos/50643829@N02/5616147572/

- When a flickr.com user visits the URL above, the HTTP request to the flickr.com servers would specify the following path as the requested resource:

  ```
  /photos/50643829@N02/5616147572/
  ```

- In addition to that path, the HTTP request would also specify an HTTP method. When you visit a web page in a browser, the HTTP corresponding request will specify the GET method. Thus, if you visited the flickr.com URL above, the request line in the HTTP request would look something like this:

  ```
  GET /photos/50643829@N02/5616147572/
  ```

- This is the information a server uses in the routing process. Specifically, the flickr.com server in the example above has to determine what resource lives at the location `/photos/50643829@N02/5616147572/` and how to respond to a GET request for that resource.
  - In this case the resource is a specific photo, and the flickr.com server responds to the GET request for that photo by returning HTML content for a web page in which the photo is displayed.

- When we write routing middleware in Express, we need to do something similar. Specifically, when an HTTP request comes into our server, we need to look at the HTTP method and the path of the requested resource and respond with the appropriate content.

- Fortunately, Express makes it pretty easy to do this. Specifically, the Express application object `app` has a number of methods we can use to specify a middleware function to handle requests for a specific path and a specific HTTP method.

- For example, if we wanted to specify a middleware function to handle HTTP GET requests on the root path `/`, we could do that using `app.get()`:

  ```
  app.get('/', function (req, res, next) {...});
  ```

- Or, if we wanted to respond to HTTP POST requests on the path `/photos`, we could do that using `app.post()`:

  ```
  app.post('/photos', function (req, res, next) {...});
  ```

- There are many other methods on the application object we can use to respond to HTTP requests of different types, including `app.put()`, `app.delete()`, etc. All of these methods are named after the corresponding HTTP method, and they take two arguments:
    - The path of a resource for which we want to handle requests.
    - A middleware function that will be called whenever a request is received with the specified HTTP method for specified resource.

- HTTP method/path/middleware combinations assigned this way are often referred to as ***routes***.

- Let's see how to actually implement a route. To start out, let's write a route for GET requests for the root path `(/)` on our server. This is the path that is often thought of as a web application's "home page" because it is the one requested when a user doesn't specify a path (e.g. visiting http://flickr.com in their browser).

- To keep things simple, let's just respond to GET requests for the root path by simply sending back the string "Hello world!" using the `send()` method of the request object:

```
app.get('/', function (req, res, next) {
     res.send("Hello world!");
});
```

- With that route in place, we can now run our server and visit its root path in our browser (e.g. using the URL http://localhost:3000/), and we'll see the text "Hello world!" displayed.

- Importantly, note that, while we have access to the next middleware function `next` in this route, we do not call it here. This is because `res.send()` ends the request-response cycle by sending the response off back to the client. Thus, no more middleware functions can be called after this one. Indeed, `res.send()` may only be called once, even within a single route.

- If we wanted to do something more interesting, we could send back some HTML content, and we could explicitly set the status code of the response using `res.status()`:

```
app.get('/', function (req, res) {
     var content = "<html>";
     content += "<body>";
     content += "<h1>Welcome!</h1>";
     content += "<p>We hope you like our page.</p>";
     content += "</body>";
     content += "</html>";

     res.status(200).send(content);
});
```

    - Note that if the argument to `res.send()` is a string, as it is here, the `Content-Type` header of the response is automatically set to `text/html`. You can use `res.type()` to set a different type if needed.

- We could write many more routes this way, and in doing so, we would make our server respond to requests for various different resources, as specified by the paths for which we write routes.
    - Of course, we can do more complicated things in a route than to send back a constant HTML string. We'll see how to do these things in due time!

Routes matching multiple paths

- Note that the middleware functions of routes specified like we did above will only be called if the specified HTTP method and path exactly match those specified in the incoming HTTP request. If our server gets a request for a method/path combination that's not specified in our routes, no routing middleware function will be called, and thus no response will be sent for that request.

- This seems to place a heavy burden on us as server developers. How can we possibly write an individual route to handle each of the many pages a large website might contain? Moreover, what can we do to handle erroneous URLs entered by the user? We can't possibly anticipate all of the typos the internet users of the world could make.

- Luckily, we don't have to do these things, because Express provides us with a few ways to write a single route that handles requests for many different paths.

- The first approach we'll look at to handling multiple paths with a single route in Express is the use of paths containing ***string patterns***, which allow us to match multiple request paths that all take a certain specified form.
    - In addition to string patterns, Express also allows the use of regular expressions in a route's path specification. Though they are extremely useful, a discussion of regular expressions is far beyond the scope of this course. If you're interested to learn more about how regular expressions can be used in JavaScript, check out the MDN page on regular expressions.

- The most familiar string pattern is the asterisk (`*`), which matches any number of any character.
    - For example the string pattern `c*t` matches the following strings:
        - cat
        - cot
        - cut
        - c123t
        - cANY_NUMBER_OF_ANY_CHARACTERt
        - etc.

- Specifying a route path that contains a string pattern allows the route to handle requests for many different paths. For example, a common practice when is to include a catch-all route by specifying the path '`*`' and using this route to respond with a 404 (not found) status and some content (even an HTML page) that lets the client know the resource they requested doesn't exist, e.g.:

```
app.get('*', function (req, res) {
     res.status(404);
     res.send("The page you requested doesn't exist");
});
```

    - Importantly, a catch-all route like this must be specified *after* any other routes we have specified for the same HTTP method. This is because Express tries to match requests against routes in the order in which the routes were specified. A route with the path '`*`' will match any request, so any other routes specified below that one would never be matched by Express.

- Another way a catch-all path like this can be used is to deny all requests with a certain HTTP method. For example, we could deny all POST requests like this:

```
app.post('*', function (req, res) {
     res.status(405);  // 405 status means "not allowed".
     res.send("No POSTs allowed!");
});
```

Parameterized routes

- Another way we can use a single route to respond to multiple request paths is to include parameters in our route path. ***Route parameters*** allow us to use a single route to match multiple request paths that follow the same pattern, and they allow us to capture values at a specified position in the request path.

- To see how route parameters can be useful, let's go back to the example we looked at above, where a user made a request for the following path to a flickr.com server:

```
/photos/50643829@N02/5616147572/
```

- In this case, the requested path corresponds to a particular photo (`5616147572`) by a particular user (`50643829@N02`), and the flickr.com servers respond to the request by sending back HTML content for a web page in which the photo is displayed.

- This path follows a pattern: any user ID and any photo ID can be plugged into the appropriate locations in the path, and the flickr.com servers will respond with an HTML page displaying *that particular photo* by *that particular user*. For example, all of the below paths are valid ones that follow the same pattern:

```
/photos/pustovit/15867520885/
/photos/parismadrid/3977203168/
/photos/cubagallery/6330704947/
/photos/egorfdrv/5675069067/
/photos/maiumaiu/33531427203/
...
```

- There are literally billions more valid paths that follow the same pattern, and for all of them, the server responds with an HTML page displaying the specified photo.

- Of course, we could probably write a string pattern (or more likely a regular expression) matching paths that follow this pattern. However, one of the key elements of patterns of this type is the fact that the particular user ID and photo ID that comprise the specified path are important: the flickr.com servers need to know them in order to grab the correct photo to send back the correct HTML content.

- For situations like this, Express allows us to create ***parameterized routes***, which are routes whose paths are written in such a way as to allow us to identify and capture values at specific locations in a path that follows a pattern, like above.

- To create a parameterized route, we can include route parameters in our paths by including a colon-prefixed identifier (e.g. `:anIdentifier`) at each location in the path at which we want to capture a value.

- For example, if we wanted to write a route that accepted HTTP GET requests on paths matching the `/photos` pattern above, we could do that like this:

```
app.get('/photos/:userId/:photoId', function (req, res) {
      ...
});
```

- When a client makes a request for a specific path that follows this pattern, we can access the particular values the client specified for the user ID and the photo ID via the `req.params` object. Specifically `req.params` will be an object that contains one entry for each colon-prefixed identifier specified in the route path, where the entry's key is the identifier itself, and the entry's value is the particular value specified at the corresponding location in the requested path.

- For example, if we had a route like the one just above, and a client made an HTTP GET request for the path `/photos/cubagallery/6330704947/`, the `req.params` object would look like this inside our route's middleware function:

```
{ userId: "cubagallery", photoId: "6330704947" }
```

- Then, inside the route's middleware function, we could use those values to fetch the specified photo and include it in an HTML page that we send back to the client. For now, we can simply tell the client what photo they asked for:

```
app.get('/photos/:userId/:photoId', function (req, res) {
     var userId = req.params.userId;
     var photoId = req.params.photoId;
     var content = "<html>";
     content += "<body>";
     content += "<h1>Welcome!</h1>";
     content +=
          "<p>You asked for this photo:" + photoId +
"</p>";
     content += "<p>By this user:" + userId + "</p>";
     content += "</body>";
     content += "</html>";

     res.status(200).send(content);
});
```

- Once we start to use databases and templates to serve dynamic content, we'll see how these parameterized routes can be even more useful.

Non-routing middleware

- Often, we'll want our server to do something for each request that's not directly related to routing for that request. For example, we might simply want to log information about the request, which we can later aggregate with other request information to compute statistics about how people are using our application.

- To accomplish tasks like this, Express allows us to specify middleware functions that are not tied to a specific HTTP method or request path. We do this by calling the `use()` method on the application object:

```
app.use(function (req, res, next) {...});
```

- Middleware functions registered this way (along with routing middleware functions) will be called for every request in the order in which they were

registered.

- For example, if we wanted to accomplish our logging task, we could register a middleware function to do that:

```
app.use(function (req, res, next) {
     console.log("== Request made");
     console.log("  - Method:", req.method);
     console.log("  - URL:", req.url);
     next();
});
```

- Importantly, note that if a middleware function registered this way does not call `res.end()`, it *must*, call the next middleware function in the sequence using `next()`, as is done in the logging example above.
    - If a middleware function calls neither `next()` nor `res.end()`, the server process will hang when it reaches that middleware function. No response will be sent, and no further middleware functions will be called.

- Note that middleware functions do not need to be anonymous, nor do they need to be be defined in the same file as the Express application. For example, we could define our logging function in an external file named `logger.js`:

```
module.exports = function (req, res, next) {
     console.log("== Request made");
     console.log("  - Method:", req.method);
     console.log("  - URL:", req.url);
     next();
};
```

- And then we could import that function in our server script and register it with Express:

```
var logger = require('./logger');
...
app.use(logger);
```

- There are also many third-party modules available on NPM that already implement useful middleware functions, and these can also be used in the same

way as our logger just above.

- Note that middleware functions can also modify either the request or response object. For example, if we wanted to use a middleware function to put a timestamp on each request to be used in later middleware functions, we could:

```
app.use(function (req, res, next) {
     var now = new Date();
     req.timeStamp = now.toString();
     next();
});

app.get('/', function (req, res) {
     var content = "<html>";
     content += "<body>";
     content += "<h1>Welcome!</h1>";
     content += "<p>We hope you like our page.</p>";
     content +=
          "<p>You visited at " + req.timeStamp + ".</p>";
     content += "</body>";
     content += "</html>";

     res.status(200).send(content);
});
```

Serving static files

- There is no restriction preventing middleware functions registered using `app.use()` from performing routing functionality. Indeed, within such a middleware function, we could check the method and path of the request, process it accordingly, and send back a response using `res.send()` (indeed, we can even specify a specific path for which to call any middleware function registered using `app.use()`). It is simply often easier to use methods like `app.get()`, `app.post()`, etc. to perform routing.

- One situation in which it is common to use `app.use()` to register routing-related functionality is when we want to serve static files living on the server's filesystem (e.g. HTML files, CSS files, image files, etc.).

- To accomplish this, the Express object `express` provides a built-in middleware function called `static()` that we can use. Specifically, if all of our static files are stored in a directory `public/` that's in the same directory as our server script (or any other directory), we can tell the Express app to serve them like so:

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

- By registering this middleware function, if a request comes in whose path matches the name of one of the files in `public/`, the server will respond with the contents of that file.

- For example, say our `public/` directory contains the following files:
  - `index.html`
  - `style.css`
  - `index.js`
  - `image.jpg`

- Then, requests for any of the following paths will be handled by sending back the contents of the corresponding file:
  - `/index.html` (or just `/`)
  - `/style.css`
  - `/index.js`
  - `/image.jpg`

- If we wanted these files to be available at a different request path, we could specify a route path in the call to `app.use()`, and that path will serve as a prefix to the filenames. For example, say we made this call to `app.use()` instead of the one above:

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

- Then the files in `public/` would be served at the following request paths:
  - `/static/index.html` (or just `/static/`)
  - `/static/style.css`
  - `/static/index.js`
  - `/static/image.jpg`

- The `static()` built-in middleware is useful, since many applications, even dynamic ones, have a significant amount of static content to serve.