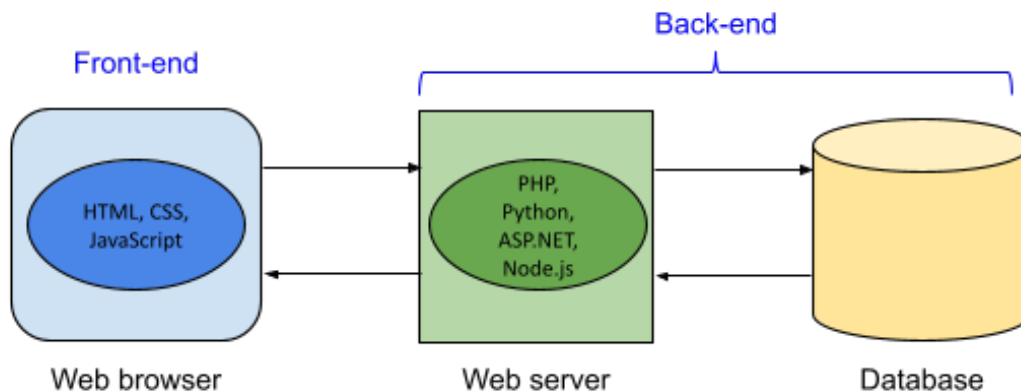


11.1 Full-stack development (Node)

Overview of front-end and back-end development

Most websites and web applications require the development of client-side technologies that interact with server-side technologies. **Client-side** (or **front-end**) refers to those technologies that run in the web browser like HTML, CSS, and JavaScript. **Server-side** (or **back-end**) refers to those technologies that run on the web server like PHP, Python, Node.js, etc. and databases. Ex: Amazon uses server-side technologies to store information on millions of products and a client-side search interface that interacts with the web server so customers can find and purchase products.

Figure 11.1.1: Front-end and back-end technologies.



A **front-end developer** is a developer that is proficient in client-side technologies. A **back-end developer** is a developer that is proficient in server-side technologies. Many developers strive to be proficient in both front-end and back-end technologies and how the two sides work together. A **full-stack developer** is a developer who has expertise in all aspects of a website or web application's development, including client technologies, server technologies, data modeling, and user interfaces. The "stack" in "full-stack" refers to the various layers that compose websites and web applications. Technology stacks have increased in complexity over the years, so even "full-stack" developers typically specialize in a few areas of the technology stack.



If unable to drag and drop, refresh the page.

User interface (UI)

Testing framework

Business logic

Server and hosting environment

Application Programming Interface (API)

Data modeling

	Issues regarding network throughput, cloud storage, virtualization, hardware constraints, multithreading, and data redundancy.
	Representing, storing, and retrieving application data in relational and non-relational databases.
	Programming logic on the front or back-end that determines how data can be created, displayed, stored, and changed.
	Programmable actions that may be performed on the underlying data. Often used by the front-end to interact with the back-end.
	Visual part of the application that users interact with.
	Automated tests that verify the web application components are working properly, independently and together.

[Reset](#)

Web hosting

When creating a web application, developers must decide where the application and application data are going to be hosted. Large companies like Google, Amazon, and Facebook have the resources to host their web applications on their own servers. Smaller companies and individuals often outsource their server hosting to web hosting companies. A **web hosting company** is a company that hosts others' websites on the company's servers, usually for a fee. Factors to consider when choosing a web hosting company include:

- **Reliability:** Many web hosting companies guarantee a certain level of uptime, and the level can be increased by paying more. Some companies backup data daily, and others provide little to no backups.
- **Flexibility:** Websites that become popular may need to quickly scale-up to handle more users. Web hosting companies may offer a virtual private server that can quickly be duplicated on other servers to meet high demand. A **virtual private server (VPS)** is an autonomous server that is hosted on a physical server with other virtual servers. Amazon Web Services (AWS) allows organizations to host virtual servers in the Amazon cloud that can quickly scale-up hosted websites when necessary.
- **Security:** Hackers may attempt to access a website's data or upload malware to a hosted website that attacks the website, other hosted websites, or the website's users. **Malware** is malicious software designed to cripple a computer system or perform unwanted actions. Some hosting companies offer extra security measures like encrypting web traffic or providing dedicated servers in heavily-guarded data centers.
- **Price:** Some web hosting companies offer limited services for free and subsidize lost income with advertising. Prices go up depending on reliability, services provided, security, amount of traffic, etc. The most expensive plans usually involve dedicated hosting where the customer is given full control over the web server.

The choice of platform dictates many of the web application's implementation decisions, since certain server-side technologies are only offered on certain platforms. Most web hosting companies provide a Linux or Windows server to host the website. Linux servers typically use open-source software: Apache web server with support for PHP, Python, Ruby, or Perl, and the MySQL database system. Windows servers generally run Microsoft's IIS web server, which supports ASP.NET and the SQL Server database system. Linux servers usually cost less than Windows servers because of the use of open-source software.

**PARTICIPATION
ACTIVITY**

11.1.2: Web hosting.



- 1) A small company may host the company's own website on the company's own web server.
☐ True
☐ False
- 2) A web hosting company that hosts websites for free is likely to provide services like backup, unlimited email addresses, and 24-hour customer support.
☐ True
☐ False
- 3) A VPS generally runs slower than a dedicated host.
☐ True
☐ False
- 4) Web hosting companies provide various levels of security.
☐ True
☐ False
- 5) Web hosting companies generally charge more for hosting on Linux servers than for hosting on Windows servers.
☐ True
☐ False

**Server-side programming**

Web developers have a wide range of options when choosing a server-side programming platform or language. When choosing a server-side programming platform, developers must consider:

- **Server platform:** Some web servers support certain languages and not others. Ex: IIS supports ASP.NET, and Apache supports PHP.
- **Tool support:** Some tools are ideal for working with certain programming languages. Ex: PhpStorm is ideal for PHP development, and Visual Studio is ideal for ASP.NET.
- **Developer experience:** JavaScript developers may choose Node.js instead of learning a new language like C#. Developers who are new to web development might already know Java or Python and prefer those languages.
- **Library support:** Some languages may have pre-built libraries that support some web applications better than others.

PARTICIPATION
ACTIVITY

11.1.3: Server-side programming platforms and languages.



If unable to drag and drop, refresh the page.

- ASP.NET
- Java
- Node.js
- Python
- Ruby on Rails
- PHP

	Scripting language created in 1994 by Rasmus Lerdorf. Currently the most popular server-side language in use.
	Collection of web development technologies first released in 2002 by Microsoft that uses the C# or VB.NET programming languages.
	General-purpose scripting language created by Guido van Rossum in the 1990s that uses frameworks like Django, web2py, and Flask to create web applications.

Web application framework written in Ruby and created by David Heinemeier Hansson in 2004.

Used to create applets on the front-end and servlets, JavaServer Pages, and web APIs on the back-end.

Runtime environment that uses modules written in JavaScript. Originally created in 2009 by Ryan Dahl.

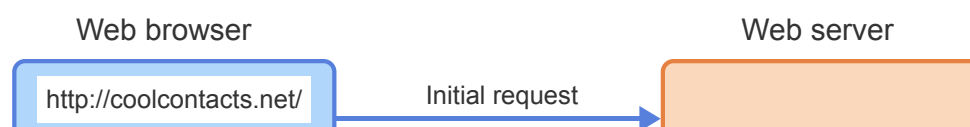
Reset

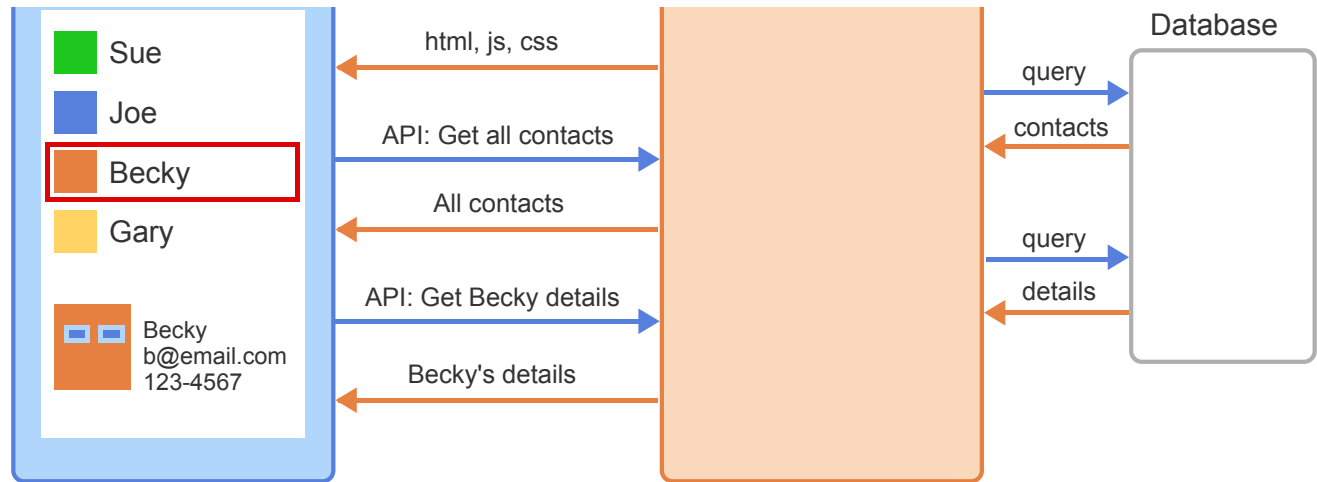
Developers have traditionally used server-side technologies to generate dynamic webpages. A **dynamic webpage** is a webpage that is generated on the web server when requested, typically personalized to the user who requested the page. With advances in web browsers, developers have begun creating static webpages that are dynamically altered by JavaScript. In this new paradigm, server-side technologies are used primarily to respond to Ajax requests and send data to the front-end for rendering.

Single Page Applications are an example of modern web development. A **Single Page Application (SPA)** is a web application that provides a similar user experience as a desktop application, all in a single webpage. Ex: Gmail, Google Docs, and Google Calendar are all SPAs. An SPA initially loads all of the application's resources so subsequent user interaction results in loading small pieces of content dynamically. Much of an SPA's programming logic is written in JavaScript, which loads data via Ajax calls to a web API. A **web API** is a collection of functions that are invoked using HTTP. Ex: An HTTP GET request to the URL <https://linkedin.com/api/contacts> may retrieve a list of all contacts from the web server.

PARTICIPATION
ACTIVITY

11.1.4: Cool Contacts SPA.





Animation content:

A web browser called `http://coolcontacts.net/`, a web server, and a database are shown. The web browser sends an initial request to the web server for the CoolContacts web app. The web server sends back all resources needed for the app in multiple request-response messages in HTML, JavaScript, and CSS. The web browser uses web API to request all contacts from the web server. The web server sends a query for the contacts to the database, which sends back all contacts to the web server, who passes it to the web app and the contacts are displayed on the web browser. When the user selects a contact, the web browser uses web API to request details for the selected contact from the web server. The web server sends a query for the details to the database, which gets sent back to the web server, who passes it to the web app and the contact details are displayed on the web browser.

Animation captions:

1. Initial request for CoolContacts web app sent to the web server.
2. All resources needed for app are downloaded in multiple request-response messages.
3. JavaScript uses web API to request all contacts.
4. All contacts are retrieved from the database and sent back to the web app for displaying.
5. User selects a contact from the web app.
6. JavaScript uses web API to request details for selected contact.
7. Web server requests Becky's details from the database and sends the details back to the web app for displaying.

**PARTICIPATION
ACTIVITY**

11.1.5: Server-side programming platforms.

- 1) A dynamic webpage might look different for two different users who are accessing the same page.
- ☐ True
- ☐ False
- 2) The business logic of an SPA should generally be encoded in the front-end.
- ☐ True
- ☐ False
- 3) SPAs generally result in less data being sent over the network than web applications developed with dynamically generated webpages.
- ☐ True
- ☐ False
- 4) Developers use ASP.NET, Java, PHP, Python, Node.js, and Ruby on Rails to create web APIs.
- ☐ True
- ☐ False

Databases

Websites and web applications normally store and retrieve information from a database and have traditionally used relational databases. A **relational database** stores data in relations (usually called tables). The **Structured Query Language (SQL)** is a language for creating, editing, selecting, and deleting data in a relational database. Popular relational database management systems (RDBMS) include MySQL, PostgreSQL, Oracle, and SQL Server.

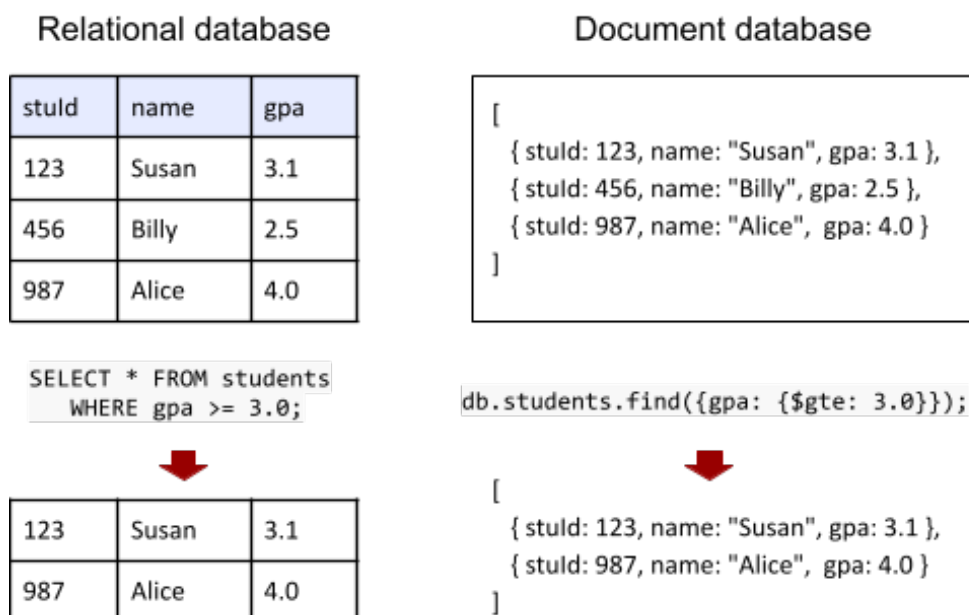
Non-relational databases, sometimes called **non-SQL** or **NoSQL** databases, have become increasingly popular over the last few years. Non-relational databases use different methods to

store and retrieve data using a variety of data access languages. Non-relational databases come in several flavors:

- Document database: For storing documents in JSON format with many levels of nesting. Ex: MongoDB.
- Key-value database: For storing values that are associated with unique keys. Ex: Redis.
- Object database: For storing objects created in object-oriented programming languages. Ex: Caché.
- Column database: For storing and processing large amounts of data using pointers that link to columns distributed over a cluster. Ex: HBase.
- Graph database: For storing graph structures with nodes and edges. Ex: Neo4j.

The figure below illustrates how information about students might be stored in a relational database with a table versus a document database using JSON-like documents. The "SELECT" statement is an SQL statement used to extract students with a 3.0 GPA or above from the table. The "db.students.find" statement is a MongoDB function used to extract the same information from the document database.

Figure 11.1.2: Relational database vs. document database for student data.



1) Relational databases will likely not be used for many web applications in the future.

- ☐ True
☐ False

2) A relational database can be used to store documents, objects, graphs, and key-value pairs.

- ☐ True
☐ False

3) Column databases are generally faster than relational databases for accessing vast amounts of data.

- ☐ True
☐ False

4) Both relational and non-relational databases have been implemented with open source software.

- ☐ True
☐ False

Client-side technologies

The user interface (UI) governs the interaction between users and web applications. Developers use HTML, CSS, and JavaScript to create the UI. Various tools exist to aid UI development:

- An **HTML preprocessor** is a program that converts a markup language into HTML. The markup languages supported by HTML preprocessors are generally easier to use and read than HTML. Ex: Haml, Markdown, Slim, Pug.
- A **CSS preprocessor** is a program that converts a CSS-like language into CSS. CSS-like languages simplify the development of CSS stylesheets used in large projects. Ex: Sass, Less, Stylus.
- A **UI library** is a library that creates UI widgets like sliders, dialog boxes, and drop-downs or

simplify DOM manipulation. Ex: jQuery UI, Bootstrap, YUI, Ext JS. Libraries like React and Vue.js support more extensive UI management.

- A **CSS front-end framework** is a framework that uses CSS or CSS pre-processors to aid in developing responsive websites that work well on every screen size. Ex: Bootstrap, YAML 4, Skeleton, Foundation.

Most modern web applications use an extensive amount of JavaScript, so developers use various tools to aid in JavaScript development:

- A **compile-to-JavaScript language** is a programming language that is compiled into JavaScript. Compile-to-JavaScript languages provide benefits lacking in JavaScript like type safety, simplified class creation, and module creation. Ex: TypeScript, CoffeeScript, and Haxe.
- A **JavaScript framework** is a JavaScript environment that dictates the organization of the application's JavaScript to simplify many programming tasks. JavaScript frameworks often dictate how UI widgets receive data or send data to the web server. Ex: AngularJS, Backbone, Ember.

Figure 11.1.3: Example use of HTML and CSS preprocessors and compile-to-JavaScript.

Haml	Resulting HTML
<pre>nav ul li a href='/home' Home li a href='/about' About li a href='/sales' Sales</pre>	<pre><nav> Home About Sales </nav></pre>

Less	Resulting CSS
<pre>@nice-green: #097911; @light-green: @nice-green + #111; header { color: @light-green; .logo { width: 250px; } }</pre>	<pre>header { color: #1a8a22; } header .logo { width: 250px; }</pre>

CoffeeScript	Resulting JavaScript
<pre>math = root: Math.sqrt square: (x) -> x * x</pre>	<pre>math = { root: Math.sqrt, square: function(x) { return x * x; } };</pre>



- 1) Haml code can be rendered directly in a web browser.
☐ True
☐ False
- 2) CSS preprocessors allow developers to write much less code compared to writing straight CSS.
☐ True
☐ False
- 3) UI libraries always use JavaScript to govern the behavior of the UI widgets.
☐ True
☐ False
- 4) CSS front-end frameworks are required to build responsive websites that work well on mobile devices.
☐ True
☐ False
- 5) TypeScript code is executed by the web browser.
☐ True
☐ False
- 6) JavaScript frameworks often simplify the use of web APIs in the browser.
☐ True
☐ False



Testing

Developers must test the full technology stack used by web applications. A variety of testing frameworks exist to automate the testing of web applications. Ex: Selenium is used to automate a

test user's interaction with a web application and verify that the UI behaves correctly.

PARTICIPATION
ACTIVITY

11.1.8: Testing a web application.

If unable to drag and drop, refresh the page.

- Usability testing
- Compatibility testing
- Security testing
- Interface testing
- Performance testing
- Functionality testing

	Verifying that each individual application function is working as expected.
	Testing the interaction between the front-end and back-end and the interactions between the server-side programs and the database.
	Testing the user's ability to properly use the web application for specific purposes.
	Testing the web application's ability to work on various browsers, operating systems, and platforms.
	Verifying the web server is able to respond reasonably under various load conditions.
	Ensuring the integrity and privacy of the user's data and interactions with the web application.

Reset

Exploring further:

- [Ranking of database systems](#)
- [6 Current Options for CSS Preprocessors](#)
- [Best languages that compile to JavaScript](#)
- [Top JavaScript Frameworks, Libraries and Tools and When to Use Them](#)
- [Summary of web application testing methodologies and tools](#)

11.2 Getting started with Node.js

Introduction to Node.js

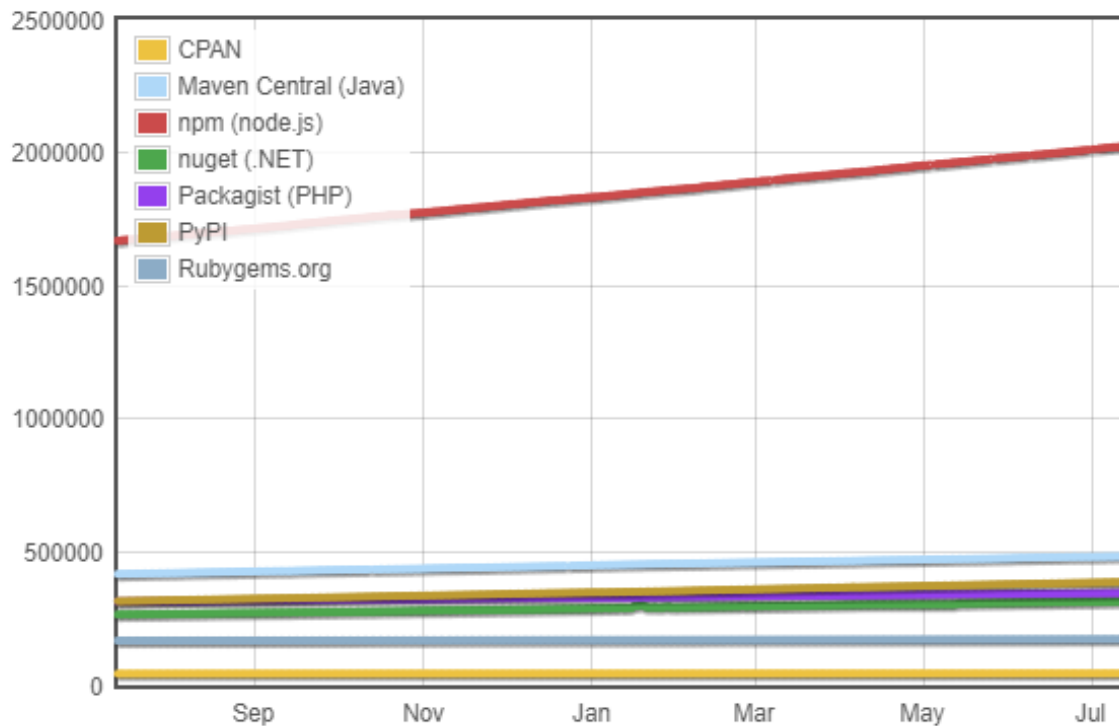
Node.js is a JavaScript runtime environment that is primarily used to run server-side web applications. Node.js has many benefits:

- The event-driven, non-blocking I/O architecture of Node.js allows Node.js to handle high loads.
- Node.js allows developers to write JavaScript on the server and client, simplifying some development tasks.
- Node.js provides a simple mechanism to create and distribute modules. A Node.js **module** is a JavaScript file that provides some useful functionality.
- Node.js works seamlessly with MongoDB, a document database that stores JSON and uses JavaScript as a query language. Web development is greatly simplified when JSON is used between the client and server, and between the server and database.

Companies using Node.js include Netflix, Walmart, Ebay, and LinkedIn. Adoption by these companies helped validate the Node.js approach and spur development of more Node.js modules.

Figure 11.2.1: Number of publicly accessible modules for Node.js (npm) in 2022 far exceeds other languages.

Module Counts



Source: [ModuleCounts.com](https://modulecounts.com)

PARTICIPATION ACTIVITY

11.2.1: Introduction to Node.js.

- 1) Node.js programs are written in JavaScript.
☐ True
☐ False
- 2) Node.js programs run in the web browser.
☐ True
☐ False

3) Node.js has over one million modules.

- ☐ True
- ☐ False



Installing and running

Developers may install Node.js using installers from the [Node.js website](#) for Windows, macOS, and other operating systems.

After installing Node.js, a developer can start the Node.js interactive shell and execute JavaScript statements. The figure below shows a command line prompt from which the user started the Node.js interactive shell by entering "node". The ".exit" command exits the interactive shell.

Figure 11.2.2: Node.js interactive shell.

```
$ node
Welcome to Node.js
> console.log("Hello,
Node.js!")
Hello, Node.js!
undefined
> x = 2
2
> .exit
```

A developer may write a Node.js program in a text editor and execute the program using the "node myprogram.js" command.

Figure 11.2.3: Simple Node.js program.

```
// hello.js
for (let i = 0; i < 5; i++) {
  console.log("Hello,
Node.js!");
}
```

```
$ node hello.js
Hello, Node.js!
Hello, Node.js!
Hello, Node.js!
Hello, Node.js!
Hello, Node.js!
```

Online services

Online IDEs like [Replit](#) and [StackBlitz](#) allow developers to run Node.js programs in the cloud instead of installing Node.js on the developer's machine.

PARTICIPATION ACTIVITY

11.2.2: Running Node.js.

- 1) The Node.js command-line program only runs on Windows.

☐ True

☐ False
- 2) The command `node test.js` starts the Node.js interactive shell.

☐ True

☐ False

3) A Node.js program can display output to the console using the `console.log()` method call.

- ☐ True
- ☐ False

Creating a simple web server

The **http module** allows a Node.js application to create a simple web server. The http module's **createServer()** method creates a web server that can receive HTTP requests and send HTTP responses. The **listen()** method starts the server listening for HTTP requests on a particular port. The server continues to run until the developer enters Ctrl+C to kill the Node.js application.

The program below shows the http module being imported with `require()`. The **require()** function imports a module for use in a Node.js program.

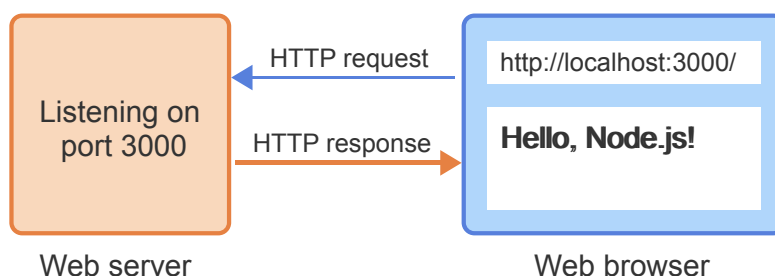
PARTICIPATION ACTIVITY

11.2.3: A simple Node.js web server.

\$ node server.js

```
// server.js
const http = require("http");

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("<h1>Hello, Node.js!</h1>");
  response.end();
}).listen(3000);
```



Animation content:

A block of code is shown:

```
// server.js
const http = require("http");
http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("<h1>Hello, Node.js!</h1>");
  response.end();
}).listen(3000);
```

The following command line prompt is used to run server.js:

```
$ node server.js
```

A web server listening on port 3000 is shown along with a web browser with the URL `http://localhost:3000/`. The web browser sends an HTTP request to the web server, which sends back an HTTP response that displays "Hello, Node.js!" on the web browser.

Animation captions:

1. server.js is executed from the command line.
2. The `require("http")` command imports the "http" module.
3. `createServer()` creates a web server object that calls the provided callback function when an HTTP request is received.
4. `listen()` starts the web server listening on port 3000 for HTTP requests.
5. The user enters a URL to access the web server running on the same machine on port 3000.
6. The HTTP request is routed to the web server, causing the request callback function to execute.
7. `writeHead()` creates an HTTP response with a 200 status code and text/html content type.
8. `write()` sends the HTML to the HTTP response object.
9. `end()` sends the HTTP response to the web browser, which renders the HTML.



1) What method causes the web server to begin listening for HTTP requests?

- ☐ require()
- ☐ createServer()
- ☐ listen()

2) What URL accesses a web server running locally and listening on port 8080?

- ☐ http://localhost/
- ☐ http://localhost:8080/
- ☐ http://8080/

3) What method sets the status code for the HTTP response?

- ☐ response.writeHead()
- ☐ response.write()
- ☐ response.end()

4) What keyboard command kills the web server program?

- ☐ Ctrl+C
- ☐ Ctrl+Z
- ☐ Ctrl+X

Projects and npm

Node.js programs are typically organized into projects. A **Node.js project** is a collection of JavaScript files, packages, configuration files, and other miscellaneous files that are stored in a directory.

Figure 11.2.4: Example Node.js project with a single JavaScript file.

```
myproject
└──
    server.js
```

A **package** is a directory containing one or more modules and a `package.json` file. A **package.json** file contains JSON that lists the package's name, version, license, dependencies, and other package metadata.

The **Node Package Manager (npm)** is the package manager for Node.js that allows developers to download, install, and update packaged modules. npm is installed with Node.js and is executed from the command line.

Figure 11.2.5: Display npm's version.

```
$ npm -
v
8.7.0
```

npm can install packages in one of two modes:

- Local mode: Packages are installed in a `node_modules` directory in the parent working directory. Ex: `npm install mypackage`
- Global mode: Packages are installed in a `{prefix}/node_modules` directory, where `{prefix}` is a location set in npm's configuration. The `--global` flag (or `-g`) directs npm to install in global mode. Ex: `npm install mypackage --global`

Local mode is ideal for installing project dependencies. A **dependency** is a package that a Node.js project must be able to access to run. Global mode is typically for installing command-line tools.

Figure 11.2.6: Get npm's prefix directory where global packages are installed.

```
$ npm config get prefix
/usr/local
```

The prefix directory will be different for Windows users.

The figure below shows a developer installing the nodemon package globally. **Nodemon** is a utility that saves developers time by restarting a Node.js application whenever the files in a project are modified.

Figure 11.2.7: Installing and running nodemon.

```
$ npm install nodemon --global
...
+ nodemon@1.19.1
added 147 packages from 90 contributors in
41.265s

$ nodemon myproject/server.js
[nodemon] 1.9.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node myproject\server.js`
```

Underscore is a library of helpful functions that extends some built-in JavaScript objects. The figure below shows a developer changing to the **myproject** directory that stores a Node.js project, installing underscore as a local package, and producing a list of the project's local packages. The underscore module is stored in **myproject/node_modules/underscore**.

Figure 11.2.8: Installing "underscore" as a local package.

```
$ cd myproject
$ npm install underscore
...
+ underscore@1.9.1
added 1 package from 1 contributor and audited 1 package in
1.983s

$ npm list
/myproject
└─ underscore@1.9.1
```

A module is imported and assigned to a variable with `require()`. *Good practice is to assign imported modules to variables that are named similar to the module name. Ex: Variable `http` for the "http" module.* However, the underscore module is usually assigned to the variable `_`, as shown in the figure below.

Figure 11.2.9: Using the underscore package to get random dice rolls.

```
const http = require("http");
const _ = require("underscore");

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("<!DOCTYPE html>\n<html>\n");
  response.write("<title>Dice Roll</title>\n");
  response.write("<body>\n");

  for (let i = 0; i < 5; i++) {
    // Use underscore to get a random number between 1 and 6
    let randNum = _.random(1, 6);

    response.write("<p>" + randNum + "</p>\n");
  }
  response.write("</body>\n</html>");
  response.end();
}).listen(3000);
```

```
<!DOCTYPE html>
<html>
<title>Dice Roll</title>
<body>
<p>3</p>
<p>5</p>
<p>5</p>
<p>6</p>
<p>1</p>
</body>
</html>
```

Table 11.2.1: Summary of npm commands.

Command	Description	Example
<code>config</code>	Manage npm configuration files	<code>npm config list</code> <code>npm config get prefix</code>
<code>install</code>	Install package locally or globally (-g)	<code>npm install nodemon -g</code>
<code>list</code>	List all installed local or global (-g) packages	<code>npm list</code>
<code>update</code>	Update a local or global (-g) package	<code>npm update lodash</code>
<code>uninstall</code>	Uninstall a local or global (-g) package	<code>npm uninstall lodash</code>

**PARTICIPATION
ACTIVITY**

11.2.5: Using npm.

1) Where does the npm command below install the grunt package?

```
$ npm install grunt -g
```

- ☐ The project's `node_modules` directory
- ☐ `{prefix}/node_modules` directory
- ☐ `{prefix}` directory

- 2) Which command displays all the installed global npm packages?
- ☐ npm install --global
 - ☐ npm list
 - ☐ npm list --global
- 3) Which command updates the local mkdirp package?
- ☐ npm install mkdirp
 - ☐ npm update mkdirp
 - ☐ npm update mkdirp -g
- 4) Which command uninstalls the local mkdirp package?
- ☐ npm update mkdirp
 - ☐ npm uninstall mkdirp -g
 - ☐ npm uninstall mkdirp

The package.json and package-lock.json files

Node.js projects use package.json to list information about the project, including the project's name, version, license, and package dependencies. Developers can manually create package.json or use the **npm init** command, which prompts the user to enter various fields and generates package.json automatically.

Figure 11.2.10: Example package.json file.

```
{
  "name": "my-web-server",
  "version": "1.0.0",
  "description": "A simple web server",
  "main": "server.js",
  "dependencies": {
    "underscore": "^1.9.1"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "",
  "license": "ISC"
}
```

When a project's package.json file is present, all the project's dependencies can be installed with a single command: `npm install`.

A **package-lock.json** file is created or modified when project dependencies are added or removed. The file ensures that the same dependency versions are always used when the project is installed on different machines. Ex: A project's package.json file may indicate a dependency version ^1.9.1. The caret character (^) means that npm should install the highest version of the library that exists, as long as the major version number, the number following the caret, is the same. So if version 1.9.2 is available, 1.9.2 is installed. But version 2.0.0 is not installed since ^1 requires the major version number to be 1. If the package-lock.json indicates that version 1.9.1 should be used, npm will install version 1.9.1 instead of any newer versions.

Semantic versioning

npm uses semantic versioning to ensure the correct package version is installed.

Semantic versioning is a popular software versioning scheme that uses a sequence of three digits: *major.minor.patch*. Ex: 1.2.3.

- *The major number indicates a major version of the package, which adds new functionality and possibly alters how previous functions now work.*
- *The minor number indicates a minor change to the package, which usually entails bug fixes and minor changes to how the package's functions work.*
- *The patch number indicates a bug fix to a minor version.*

Figure 11.2.11: Files composing Node.js project.

```
myproject
├── node_modules
│   └── underscore
├── package.json
├── package-
lock.json
└── server.js
```

PARTICIPATION ACTIVITY

11.2.6: Node.js project's package.json file.

- 1) A package.json file may list the project developers, homepage, and bugs.
☐ True
☐ False
- 2) Packages installed from npm occasionally have package.json files.
☐ True
☐ False

3) The "scripts" value in the example package.json above allows the web server to be started with the command:



```
npm start
```

- ☐ True
- ☐ False

4) The following command installs the mkdirp module and adds mkdirp to the "dependencies" block of package.json:



```
npm install mkdirp
```

- ☐ True
- ☐ False

5) A project's package.json and package-lock.json files may list different dependency versions.



- ☐ True
- ☐ False

Exploring further:

- [Node.js website](#)
- [npm documentation](#)
- [package.json documentation](#)
- [package-lock.json documentation](#)
- [Understanding module.exports and exports in Node.js](#)

11.3 Express

Express server

Express is a popular web application framework for Node.js because Express allows developers to create web servers with less code. Express is installed with npm: `npm install express`.

PARTICIPATION ACTIVITY

11.3.1: Simple Express web server.

```
myproject
├── node_modules
│   ├── express
│   └── etc...
├── public
│   └── hello.html
├── package.json
└── server.js
```

hello.html

```
<!DOCTYPE html>
<html>
  <title>Hello Express</title>
  <body>
    <h1>Hello, Express!</h1>
  </body>
</html>
```

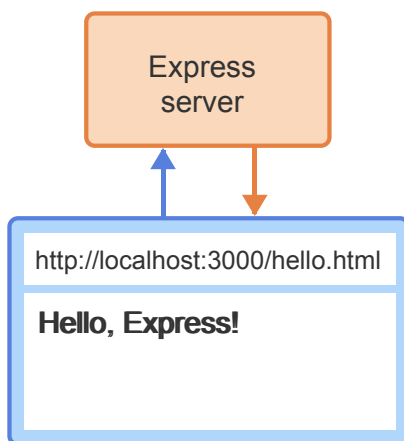
server.js

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

// Serve static files from the public dir
app.use(express.static("public"));

// Start the web server
app.listen(3000, function() {
  console.log("Listening on port 3000...");
});
```

```
$ node server.js
Listening on port 3000...
```



Animation content:

The project's directory is shown. In the myproject directory is: package.json file and server.js file,

the directory `node_modules`, containing the express package, etc, and the public directory, containing `hello.html`. `hello.html` contains the following code:

```
<!DOCTYPE html>
<html>
  <title>Hello Express</title>
  <body>
    <h1>Hello, Express!</h1>
  </body>
</html>
```

`server.js` contains:

```
const express = require("express");
const app = express();
// Serve static files from the public dir
app.use(express.static("public"));
// Start the web server
app.listen(3000, function() {
  console.log("Listening on port 3000...");
});
```

To execute `server.js`, `node server.js` is typed at the command line. In `server.js` the `express` module is imported, the path with the static files is applied to the `express` application object, and the web server starts listening on port 3000. A user enters the URL `http://localhost:3000/hello.html` in the browser. The browser sends a request for `hello.html` and displays `hello.html`.

Animation captions:

1. After installing, the `express` package is located in the project's `node_modules` directory.
2. The public directory is for storing static files (HTML, images, etc.) that can be served by the Express server.
3. `server.js` contains the Node.js application code and is executed from the command line.
4. The `require()` function imports the `express` module, and `express()` creates the `express` application object.
5. `express.static()` names the path that contains static files to be served by the server. `app.use()` applies the path to the `express` object.
6. `app.listen()` starts the web server listening for HTTP requests on port 3000 and outputs to the console after the web server starts.
7. When the browser requests `hello.html` from the Express server, the server responds with `hello.html`.

**PARTICIPATION
ACTIVITY**

11.3.2: Express server.

Refer to the animation above.

- 1) The **express ()** function starts the Express server.

☐ True

☐ False
- 2) To serve **images/dog.jpg** to the browser, the **images** directory must reside in the ____ directory.

☐ public

☐ node_modules
- 3) If the browser requested **http://localhost:3000/bye.html** in the animation above, the Express server would return a 404 status code.

☐ True

☐ False

Routes

Express uses routing to handle browser requests. An Express **route** is a specific URL path and an HTTP request method to which a callback function is assigned. A route's callback function has **request** and **response** parameters, which represent the HTTP request and response.

A route is defined with the structure: **app.method(path, callback)**

- **app** - Express instance
- **method** - HTTP request method (get, post, etc.)
- **path** - URL path
- **callback** - Callback function executed when the route matches

The figure below shows an example of a GET route and a POST route with different paths. The route callback functions use **res.send()** to send an HTTP response containing HTML.

Figure 11.3.1: Example routes.

```
// Called for GET request to
http://localhost:3000/hello
app.get("/hello", function(req, res) {
  res.send("<h1>Hello, Express!</h1>");
});

// Called for POST request to
http://localhost:3000/goodbye
app.post("/goodbye", function(req, res) {
  res.send("<h1>Goodbye, Express!</h1>");
});
```

**PARTICIPATION
ACTIVITY**

11.3.3: Express routes.

- 1) Referring to the figure above, which route matches a POST request to "/hello"?
 - ☐ "/hello"
 - ☐ "/goodbye"
 - ☐ Neither route
- 2) Assuming an Express app running on port 3000, what URL matches the route below?

```
app.get("/droids",
function(req, res) {
  res.send("These aren't the
droids you're looking for.");
});
```

- ☐ http://localhost/droids
- ☐ http://localhost:3000/droidssss
- ☐ http://localhost:3000/Droids

3) What route matches a POST request to the following URL?

`http://localhost:3000/student`

- ☐ `app.get("/student", callback);`
- ☐ `app.post("/student", callback);`
- ☐ `app.post("http://localhost:3000/student", callback);`

Middleware

A **middleware function** (or just **middleware**) is a function that examines or modifies the `request` and/or `response` objects. A middleware function has three parameters:

- `req` - HTTP request object
- `res` - HTTP response object
- `next` - Callback to the middleware function

The Express method **`use()`** enables a middleware function to execute.

PARTICIPATION ACTIVITY

11.3.4: Middleware function that logs requests.

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

const logRequest = function(req, res, next) {
  console.log(`Request: ${req.method} for ${req.path}`);
  next();
};

app.use(logRequest);

app.get("/hello", function(req, res) {
  res.send("<h1>Hello, Express!</h1>");
});

app.listen(3000, function() {
  console.log("Listening on port 3000...");
});
```

`http://localhost:3000/hello`

Hello, Express!

```
$ node server.js
Listening on port 3000
Request: GET for /hel
```

Animation content:

A code block reads:

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

const logRequest = function(req, res, next) {
  console.log(`Request: ${req.method} for ${req.path}`);
  next();
};

app.use(logRequest);

app.get("/hello", function(req, res) {
  res.send("<h1>Hello, Express!</h1>");
});

app.listen(3000, function() {
  console.log("Listening on port 3000...");
});
```

The console is shown:

```
$ node server.js
Listening on port 3000...
Request: GET for /hello
```

The node server.js command starts the server. A browser shows the URL `http://localhost:3000/hello`, which logs the request on the console and displays 'Hello, Express!' in the browser.

Animation captions:

1. The middleware function `logRequest()` logs the request method and path to the console.
2. `logRequest()` is enabled by calling `app.use()`.
3. After starting the server, the browser's request to `/hello` causes `logRequest()` to be called.
4. The request method `GET` and path `/hello` are logged in the server's console.
5. `logRequest()` calls `next()` to allow other middleware and the `/hello` route to execute.
6. After `logRequest()` finishes, the `/hello` route responds with a "Hello, Express!" message, which is displayed in the browser.

**PARTICIPATION
ACTIVITY**

11.3.5: Middleware.



Refer to the animation above.

1) What does a GET request to the path `"/"` log to the console?



- ☐ Request: GET for `/`
- ☐ Request: GET for `/hello`
- ☐ Nothing is logged

2) If the `app.use()` call is removed from the server, what does a GET request to the path `"/hello"` log to the console?



- ☐ Request: GET for `/`
- ☐ Request: GET for `/hello`
- ☐ Nothing is logged

3) If the `next()` call is moved from `logRequest()`, what does a GET request to the path `"/hello"` log to the console?



- ☐ Request: GET for `/`
- ☐ Request: GET for `/hello`
- ☐ Nothing is logged

4) If the `app.use()` call is moved *after* the call to `app.get()`, what does a GET request to the path `"/hello"` log to the console?



- ☐ Request: GET for `/`
- ☐ Request: GET for `/hello`
- ☐ Nothing is logged

Third-party middleware

Third parties have created many useful middleware functions that can be installed with npm. Ex: `npm install morgan` installs middleware that logs information about HTTP requests.

Figure 11.3.2: Using morgan middleware.

```
const express =  
require("express");  
const morgan = require("morgan");  
  
const app = express();  
  
// Show HTTP requests in the  
console  
app.use(morgan("dev"));  
  
app.listen(3000);
```

Example console logging of requests:

```
GET /hello.html 200 8.645 ms - 132  
POST /hello 200 4.280 ms - 25  
GET /blah 404 1.297 ms - -
```

Table 11.3.1: Popular third-party middleware for Express.

Middleware	Description
morgan	Logs HTTP request information
cookie-parser	Parses the cookie header in an HTTP request
errorhandler	Aids debugging during development
csurf	Protects against cross-site request forgery (CSRF)
compression	Compresses response bodies
express-session	Manages session data on the server

©zyBooks 04/15/24 16:48 2071381
Marco Aguilar
CIS192_193_Spring_2024

**PARTICIPATION
ACTIVITY**

11.3.6: Popular middleware.

Refer to the table above.

- 1) What middleware indicates that an HTTP request returned a 404 status code?

☐ cookie-parser

☐ morgan

☐ csurf
- 2) What middleware could protect an Express server from a malicious CSRF attack?

☐ csurf

☐ compression

☐ express-session

3) What middleware should a developer use to track the user's temporary session data?

- ☐ cookie-parser
- ☐ compression
- ☐ express-session

Express application generator.

*A developer can use the application generator tool **express-generator** to automatically create an Express application skeleton. The tool saves developers time from writing application code that is common to many Express applications. Projects produced by **express-generator** use the Pug template engine to create views or webpages that interact with the MongoDB database.*

Exploring further:

- [Express API reference](#)
- Middleware: [morgan](#), [cookie-parser](#), [errorhandler](#), [csurf](#), [compression](#), [express-session](#)
- [Express application generator](#)

11.4 Express request data

Query string parameters

Express automatically parses query string parameters (the values appearing after the "?" in a URL) and stores the parameters' names and values in the **req.query** object.

PARTICIPATION ACTIVITY

11.4.1: Extracting query string values.

http://localhost:3000/hello?name=Bob&age=21

Hello, Bob!

You are 21 years old.

```
app.get("/hello", function(req, res) {  
  const html =  
    `<h1>Hello, ${req.query.name}!</h1>  
    <p>You are ${req.query.age} years old.</p>`;  
  res.send(html);  
});
```

Animation content:

A web browser is shown with the URL `http://localhost:3000/hello?name=Bob&age=21` and the Node.js code block:

```
app.get("/hello", function(req, res) {  
  const html =  
    `<h1>Hello, ${req.query.name}!</h1>  
    <p>You are ${req.query.age} years old.</p>`;  
  res.send(html);  
});
```

The browser requests the `/hello` route with `name = Bob` and `age = 21` in the query string. The route's callback function, `function(req, res)`, replaces `req.query.name` with `Bob` in the `h1` tag and replaces `req.query.age` with `21` in the `p` tag. `res.send(html)` sends back a response with the name and age, which then displays in the browser.

Animation captions:

1. The browser requests the `/hello` route with data in the query string.
2. The route's callback function extracts name and age from the `req.query` object.
3. `res.send()` sends back a response with the name and age.

PARTICIPATION ACTIVITY

11.4.2: Query string parameters.

Refer to the animation above.

- 1) If name is not found in the query string, the webpage says "Hello, undefined!"
☐ True
☐ False
- 2) If the query string contains "name=Doctor+Who", the webpage says "Hello, Doctor+Who!"
☐ True
☐ False
- 3) If the query string contains "city=Dallas", the city is available in the route callback function as `res.query.city`.
☐ True
☐ False

Posting form data

When a form posts data to an Express route, the form's data is URL-encoded and sent in the body of the HTTP request. The **`express.urlencoded()`** method is middleware that parses URL-encoded data in a request body and adds the parsed values to the `req.body` object.

PARTICIPATION ACTIVITY

11.4.3: Extracting posted form data.

public/hello.html

```
<form method="post" action="/hello">
  <p>
    <label>Name? <input type="text" name="name"></label>
  </p>
  <p>
    <label>Age? <input type="number" name="age"></label>
  </p>
  <input type="submit" value="Submit">
</form>
```

http://localhost:3000/hello

Hello, Tamika!

You are 24 years old.

server.js

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

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

app.use(express.urlencoded({extended: false}));

app.post("/hello", function(req, res) {
  const html = `

# Hello, ${req.body.name}!</h1> <p>You are ${req.body.age} years old.</p>`; res.send(html); }); app.listen(3000);


```

```
POST /hello HTTP/1.1
Content-Type: application/x-www-form-urlencoded
name=Tamika&age=24
```

Parsed URL-encoded data

req.body.name = Tamika

req.body.age = 24

Animation content:

Two blocks of code are shown. public/hello.html:

```
<form method="post" action="/hello">
  <p>
    <label>Name? <input type="text" name="name"></label>
  </p>
  <p>
    <label>Age? <input type="number" name="age"></label>
  </p>
  <input type="submit" value="Submit">
</form>
```

And server.js:

```
const express = require("express");
const app = express();
app.use(express.static("public"));
app.use(express.urlencoded({extended: false}));
app.post("/hello", function(req, res) {
  const html = `

# 


```

A web browser with the URL <http://localhost:3000/hello.html> is shown. On the screen a name and age field is displayed. The user types Tamika for name and 24 for age. Pressing submit sends the following request to the Express server:

POST /hello HTTP/1.1

Content-Type: application/x-www-form-urlencoded

name=Tamika&age=24

Next the Parsed URL-encoded data is shown as the following:

req.body.name = Tamika

req.body.age = 24

The Express server sends the following to the web browser screen:

Hello, Tamika!

You are 24 years old.

Animation captions:

1. Express is configured to serve files in the "public" directory.
2. `express.urlencoded()` returns middleware that only parses HTTP request bodies when Content-Type: application/x-www-form-urlencoded is present in the header.
3. The web browser requests `hello.html` from the Express server and renders the webpage.
4. The user enters her name and age and presses Submit. A POST request with the user's name and age is sent to the Express server.
5. The middleware decodes posted data from the HTTP request body and attaches the data to `req.body`.
6. The request is posted to the `/hello` route callback function, which returns a response saying hello to the user.

PARTICIPATION ACTIVITY

11.4.4: Posting form data.

Refer to the animation above.

1) When the form is submitted to the Express server, what is the Content-Type?

- ☐ text/html
- ☐ application/x-www-form-urlencoded
- ☐ application/json

- 2) If the user clicks Submit without entering a name, what does the resulting webpage display?
- ☐ Hello, !
 - ☐ Hello, Bob!
 - ☐ Hello, Express!
- 3) Is the `express.static()` method middleware?
- ☐ Yes
 - ☐ No

Route parameters

Express applications often use routes with data in the URL path to generate dynamic webpages. A **route parameter** is a string near or at the end of the URL path that specifies a data value. A route parameter's name is defined in the route path with a colon (:) before the parameter name. Ex: In the route path `"/users/:username"`, `"username"` is a route parameter's name. Route parameters are attached to the **`req.params`** object.

PARTICIPATION ACTIVITY

11.4.5: Extracting route parameters.

http://localhost:3000/users/jblack

Profile for jblack

```
app.get("/users/:username", function(req, res) {  
  const username = req.params.username;  
  res.send("<h1>Profile for " + username + "</h1>");  
});
```

Animation content:

A web browser is shown with the URL `http://localhost:3000/user/jblack` and the code block:

```
app.get("/users/:username", function(req, res) {  
  const username = req.params.username;  
  res.send("<h1>Profile for " + username + "</h1>");  
});
```

The browser requests `/user/jblack`, which matches the `/users/:username` route format. The routes callback function extracts `jblack` from `req.params.username` to store in `username`. `res.send()` sends back a response to the web browser and displays "Profile for jblack" on the screen.

Animation captions:

1. The browser requests `/user/jblack`, which matches the `/users` route with the `:username` parameter name.
2. The route's callback function extracts the username "jblack" from the `req.params` object.
3. `res.send()` sends back a response with the username.

PARTICIPATION ACTIVITY

11.4.6: Route parameters.

Refer to the animation above.

1) If the URL path is `/users/abc`, the webpage says "Profile for abc"

- ☐ True
☐ False

2) If the URL path is `/users/`, the webpage says "Profile for "

- ☐ True
☐ False

- 3) If the URL path is `/city/Denver`, the city is available in the route callback function below as `req.params.city`.



```
app.get("/city/:cityName",  
function(req, res) {  
    ...  
});
```

- ☐ True
- ☐ False

Route parameter middleware

The Express object's ***param()*** method defines parameter middleware that executes before a route's callback function. The middleware function has a fourth parameter that contains the value assigned to the route parameter.

The figure below shows the username parameter being examined in the parameter middleware. If the username is "jblack", the user's name is "Jack Black". Otherwise, the user's name is unknown. The **req** object is modified to contain a **user** object with properties **name** and **username**. A real-world application would replace the if-else statement with a database query that looks up names and usernames from a database.

Figure 11.4.1: Route parameter middleware example.

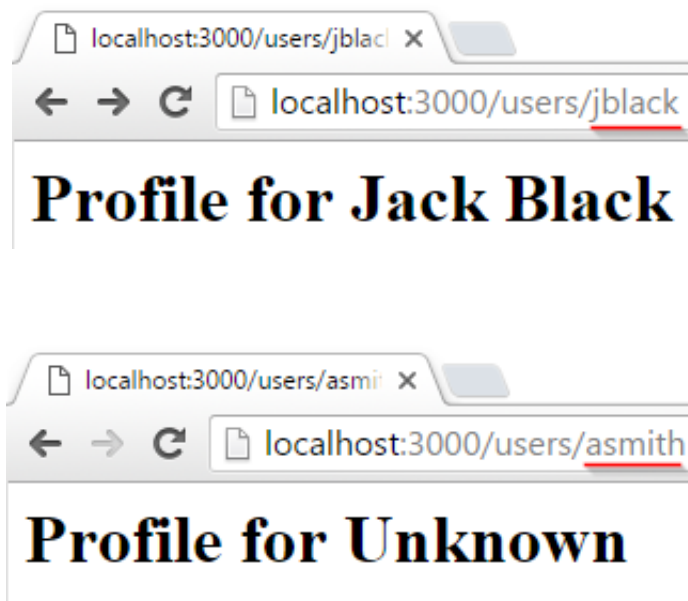
```
// Parameter middleware executes before the route
app.param("username", function(req, res, next, username) {

  // Check if username is recognized
  if (username === "jblack") {
    req.user = { name: "Jack Black", username: username };
  }
  else {
    req.user = { name: "Unknown", username: username };
  }

  // Continue processing the request
  next();
});

app.get("/users/:username", function(req, res) {

  // req.user contains the user info set in the parameter
  // middleware
  res.send("<h1>Profile for " + req.user.name + "</h1>");
});
```

**PARTICIPATION
ACTIVITY**

11.4.7: Route parameter middleware.

Refer to the code segment below.


```
app.param("zip", function(req, res, next, zip) {
  if (zip === "80015") {
    req.forecast = {high: 95, low: 72};
  }
  next();
});

app.get("/weather/:zip", function(req, res) {
  if (req.forecast) {
    res.json(req.forecast);
  }
  else {
    res.status(404).json({ error: "Unknown ZIP code" });
  }
});
```

- 1) What response is returned for the following URL?



`http://localhost:3000/weather/80015`

- ☐ A webpage that displays the high and low values
- ☐ { "high": 95, "low": 72 }
- ☐ { "error": "Unknown ZIP code" }

- 2) What response is returned for the following URL?



`http://localhost:3000/weather/12345`

- ☐ A webpage that displays the high and low values
- ☐ { "high": 95, "low": 72 }
- ☐ { "error": "Unknown ZIP code" }

3) Suppose the `next()` function call in the parameter middleware is removed. What response is returned for the following URL?

`http://localhost:3000/weather/80015`

- ☐ The server returns a 404 status.
- ☐ The server returns the JSON: `{ "high": 95, "low": 72 }`
- ☐ The server does not return a response.

4) The `res.json()` method encodes a JavaScript object as JSON. What Content-Type is sent in the HTTP response when using `res.json()`?

- ☐ text/plain
- ☐ text/javascript
- ☐ application/json

Exploring further:

- [Express API reference](#)

11.5 Pug

Template engines

A **template engine** creates dynamic webpages by replacing variables in a template file with specific values, creating HTML that is sent to the web browser. Template engines usually support programming features like conditional statements, loops, and functions. Several template engines work with Express: Pug, Mustache, EJS, and others. **Rendering** is the process of combining data with a template. The rendered webpages serve as the web application's views. A **view** is the visual presentation of the application's data.

To render template files, the following Express application properties must be set with the `set()` method:

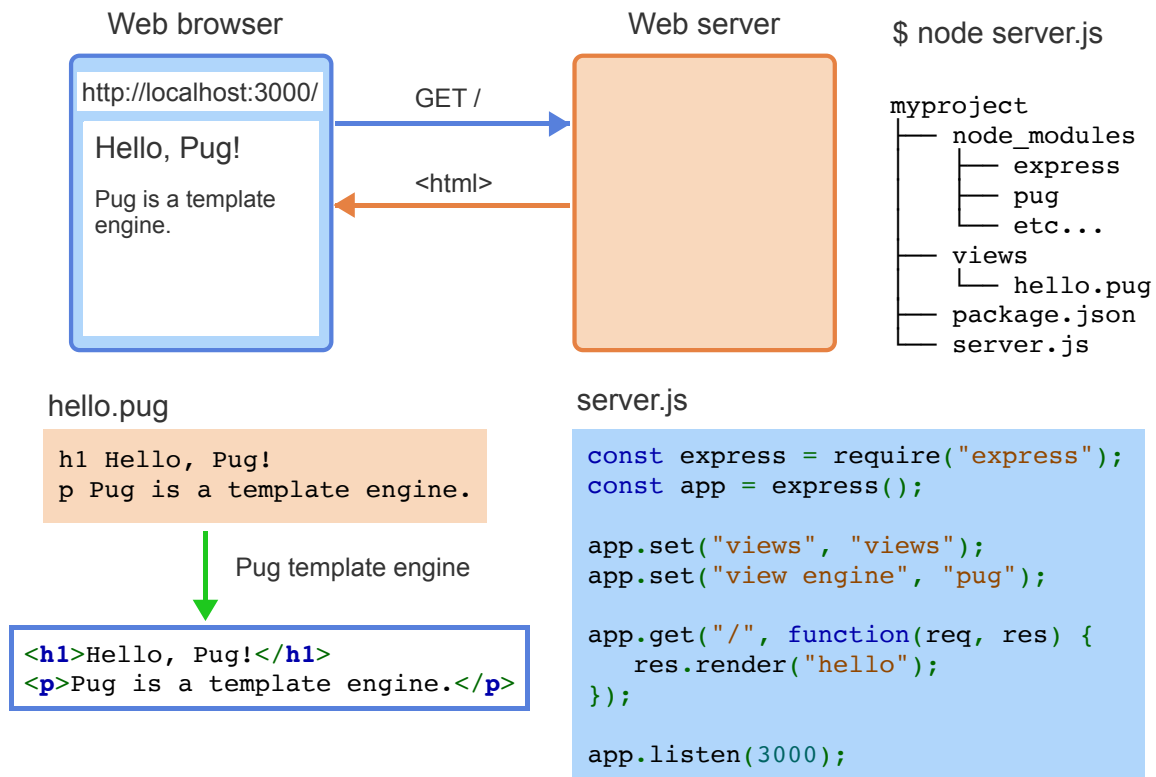
- **views** - The directory where the template files are located. Ex:
`app.set("views", "views-dir");`
- **view engine** - The template engine to use. Ex: `app.set("view engine", "pug");`

Pug (previously Jade) is a popular template engine for Express. Pug is installed using:
`npm install pug`.

Pug files use the `.pug` file extension. The Express response method **`render()`** uses the Pug template engine to render a `.pug` file and return the rendered HTML.

PARTICIPATION ACTIVITY

11.5.1: Rendering Pug on the web server.



Animation content:

A directory called myproject has 2 subdirectories. The first is node_modules and contains express, pug, and etc. The second is views and contains hello.pug. myproject also contains package.json and server.js. server.js contains the following code:

```
const express = require("express");
const app = express();
app.set("views", "views");
app.set("view engine", "pug");
app.get("/", function(req, res) {
  res.render("hello");
});
app.listen(3000);
```

Hello.pug contains the following code:

h1 Hello, Pug!

p Pug is a template engine.

The Pug code renders the following HTML:

<h1>Hello, Pug!</h1>

<p>Pug is a template engine.</p>

The console uses the node server.js command to start the server. A web browser is shown with the URL http://localhost:3000/. The web browser sends a GET request to the web server with GET \. The rendered HTML is sent back to the web browser from the web server, to display the following:

Hello, Pug!

Pug is a template engine.

Animation captions:

1. Start the Node.js app, import and instantiate Express.
2. Set Express's "views" property to the location of Pug files: "views" directory.
3. Set Express's "view engine" property to the "pug" template engine.
4. The path "/" will use res.render() to render the "hello" view.
5. Start the web server listening on port 3000.
6. User enters URL to access the server running on the same machine on port 3000.
7. The "/" route matches, so the "hello" view (hello.pug) is rendered by the Pug template engine.
8. Rendered HTML is sent to the web browser and displayed.

PARTICIPATION
ACTIVITY

11.5.2: Template engines.

- 1) A template engine runs on the web server.
☐ True
☐ False
- 2) The "pug" module must be explicitly imported for Express to use Pug.
☐ True
☐ False
- 3) The JavaScript code below causes the Pug template engine to render the file goodbye.html into HTML.

```
res.render("goodbye");
```

- ☐
- True
-
- ☐
- False

Pug syntax

The Pug template engine interprets text at the beginning of a line as an HTML tag by default. Unlike HTML, Pug uses indentation to indicate nesting of tags.

PARTICIPATION
ACTIVITY

11.5.3: Indentation's effects on rendered HTML.

Pug

```
p First line  
span Second line  
div Third line
```

```
p First line
```

Rendered HTML

```
<p>First line</p>  
<span>Second line</span>  
<div>Third line</div>
```

```
<p>First line<span>Second line</span></p>
```

```
span Second line
div Third line
```

```
<div>Third line</div>
```

```
p First line
  span Second line
    div Third line
```

```
<p>First line<span>Second line
  <div>Third line</div></span></p>
```

Animation content:

3 examples of Pug being rendered to HTML are shown. The first block of Pug:

```
p First line
span Second line
div Third line
```

And the first rendered HTML:

```
<p>First line</p>
<span>Second line</span>
<div>Third line</div>
```

The second block of Pug:

```
p First line
  span Second line
div Third line
```

And the second rendered HTML:

```
<p>First line<span>Second line</span></p>
<div>Third line</div>
```

The third block of Pug:

```
p First line
  span Second line
    div Third line
```

And the third rendered HTML:

```
<p>First line<span>Second line<div>Third line</div></span></p>
```

Animation captions:

1. The first word on a line is rendered as an HTML tag surrounding the following text.
2. When a line in Pug is indented, the rendered tag is nested in the tag above.
3. Each level of indentation nests the rendered tag one more level in the HTML.

Table 11.5.1: Summary of Pug syntax.

Description	Pug example	Rendered HTML
HTML doctype	<code>doctype</code>	<code><!DOCTYPE html></code>
Nesting tags	<pre>ol li Pug uses indentation em to embed tags inside li each other.</pre>	<pre> Pug uses indentation to embed tags inside each other. </pre>
Displaying text	<pre>p A pipe at the beginning of each line displays raw text.</pre>	<pre><p>A pipe at the beginning of each line displays raw text.</p></pre>
Displaying lots of text	<pre>p. The period is the simplest way to include multiple lines of text because no pipe is required before each line.</pre>	<pre><p>The period is the simplest way to include multiple lines of text because no pipe is required before each line. </p></pre>
Tag attributes	<pre>div(style="width:400px") img(src="dog.jpg", alt="Barking dog")</pre>	<pre><div style="width:400px"> </div></pre>
ID attribute	<pre>h1#headline Pug is everywhere! #options div is the default tag</pre>	<pre><h1 id="headline">Pug is everywhere!</h1> <div id="options">div is the default tag</div></pre>
Class attribute	<pre>span.important Read this first! div.highlight.pull-right</pre>	<pre>Read this first! <div class="highlight pull- right"></div></pre>

Comments

```
// Comment is visible in  
HTML  
//- Invisible comment
```

```
<!--Comment is visible in  
HTML-->
```

**PARTICIPATION
ACTIVITY**

11.5.4: Pug syntax basics.

Select the HTML rendered from the Pug code.

1) `h2 Learning
span Pug`

- ☐ `<h2>Learning</h2>
Pug`
- ☐ `<h2>Learning span
Pug</h2>`
- ☐ `<h2>Learning
Pug</h2>`

2) `p A
p B
p C`

- ☐ `<p>A<p>B<p>C</p></p>
</p>`
- ☐ `<p>A</p><p>B</p>
<p>C</p>`
- ☐ `<p>A</p><p>B<p>C</p>
</p></h2>`

3) `span This
strong is a
| test`

- ☐ `Thisis
atest`
- ☐ `Thisis
atest`
- ☐ `Thisis a
test`

4) `.good This is
#better a test`

- ☐ `<div id="good">This
is</div>
<div class="better">a
test</div>`
- ☐ `<div class="good"
id="better">This is a
test</div>`
- ☐ `<div class="good">This
is</div>
<div id="better">a
test</div>`

5) `input(type="radio",
checked=true)`

- ☐ `<input type="radio">`
- ☐ `<input type="radio"
checked>`
- ☐ `<input type="radio"
checked="true">`

6) `script.`
`console.log("Pug");`
`let x = 10;`

- ☐ `<script>console.log("Pug");`
`</script>`
`<script>let x = 10;</script>`
- ☐ `<script>console.log("Pug");`
`let x = 10;`
`</script>`
- ☐ `<script>console.log("Pug");let`
`x = 10;`
`</script>`

JavaScript in Pug

A developer can use JavaScript in a Pug template that is executed on the web server when the template is rendered into HTML. JavaScript code is prefaced with a dash `-`.

Four techniques exist to display JavaScript variables:

- `tag= variable` - Escape any HTML in the variable and output the result.
- `tag!= variable` - Output the variable without escaping the HTML.
- `#{variable}` - Escape any HTML in the variable and output the result.
- `!{variable}` - Output the variable without escaping the HTML.

Attribute values use JavaScript variables without the need for special characters.

Figure 11.5.1: JavaScript in a Pug template.

Pug example	Rendered HTML
<pre>- const homeTeams = ["Broncos", "Nuggets", "Rockies"] ul - for (let i = 0; i < homeTeams.length; i++) li= homeTeams[i]</pre>	<pre> Broncos Nuggets Rockies </pre>
<pre>- let rand = Math.floor(Math.random() * 100) - let color = "red" p My favorite number is span(style="color:" + color) #{rand} .</pre>	<pre><p>My favorite number is 8. </p></pre>

JavaScript data can be passed to a Pug template using locals. A **local** is a JavaScript variable defined outside of a template whose value is made available to a template. Locals are passed to the Pug template through the `res.render()` method.

PARTICIPATION
ACTIVITY

11.5.5: Locals passed to the Pug template.

<pre>// From an Express route let locals = { title: "Intro to Pug", name: "Sonya", email: "sonya@gmail.com" }; res.render("index", locals);</pre>	
<pre>h3= title a(href="mailto:" + email) #{name}</pre>	<pre><h3>Intro to Pug</h3> Sonya</pre>
index.pug	Rendered HTML

Animation content:

3 blocks of code are shown. The JavaScript:

```
// From an Express route
```

```
let locals = { title: "Intro to Pug",  
  name: "Sonya", email: "sonya@gmail.com" };
```

```
res.render("index", locals);
```

The Pug called index.pug:

```
h3= title
```

```
a(href="mailto:" + email) #{name}
```

And the rendered HTML:

```
<h3>Intro to Pug</h3>
```

```
<a href="mailto:sonya@gmail.com">Sonya</a>
```

Animation captions:

1. In the Express application, "locals" is assigned an object containing information needed in the Pug template.
2. The "locals" object is passed to the index.pug template.
3. The locals are substituted for title, name, and email in the Pug template and rendered in the HTML.

PARTICIPATION ACTIVITY

11.5.6: JavaScript in Pug templates.

Select the Pug code that renders the HTML.

- 1) `There are 5 problems.`
- ☒ `- let c = 5
span There are
 | c
 | problems.

let c = 5
span There are #{c}
problems.`
- ☐ `- let c = 5
span There are #{c}
 | problems.`

2) `<p><i>sugar</i></p>`

```

- let ingredients = [
  "milk", "sugar",
  "eggs" ]
p
  i= ingredients[1]

- let ingredients = [
  "milk", "sugar",
  "eggs" ]
p= ingredients[1]

- let ingredients = [
  "milk", "sugar",
  "eggs" ]
p
  i= ingredients

```

3) `<p>Test
this!</p>
Test this!
`

```

- let msg = "<em>Test  
this!</em>"
p= msg
span= msg

- let msg = "<em>Test  
this!</em>"
p= msg
span #{msg}

- let msg = "<em>Test  
this!</em>"
p= msg
span!= msg

```

4) `<p>FIVE;five.</p>`

- ☐ `- let msg = "`
`five"`
`p #{msg.toUpperCase() }`
`!{msg}.`
- ☐ `- let msg = "`
`five"`
`p #{msg} !{msg}.`
- ☐ `- let msg = "`
`five"`
`p #{msg.toUpperCase() }`
`#{msg}.`

5) Assume locals `name` and `age` are passed to the template.

`<div id="Carlos">20</div>`

- ☐ `div(id=#{name}) #{age}`
- ☐ `div(id=name) #{age}`
- ☐ `div#name #{age}`

Conditions and loops

Pug has built-in constructs for conditions and loops that work with JavaScript variables.

Table 11.5.2: Summary of Pug conditions and loops.

Description	Pug example	Rendered HTML
if condition	<pre>- let answer = "Washington" if answer p The answer is: #{answer}. else p There is no answer.</pre>	<pre><p>The answer is: Washington.</p></pre>

unless condition	<pre>- let user = { role: "regular" } unless user.role === "admin" p Hello, regular user.</pre>	<pre><p>Hello, regular user.</p></pre>
case (switch)	<pre>- let points = 5 case points when 0 p No points. when 1 p One point. default p #{points} points.</pre>	<pre><p>5 points.</p></pre>
each loop with array	<pre>- let capitals = ["Austin", "Little Rock", "Denver"] ul each city in capitals li= city</pre>	<pre> Austin Little Rock Denver </pre>
each loop with object	<pre>- let capitals = { "TX": "Austin", "AR": "Little Rock", "CO": "Denver" } ul each city, state in capitals li= city + ", " + state</pre>	<pre> Austin, TX Little Rock, AR Denver, CO </pre>
while loop	<pre>- let n = 1 while n <= 4 em= n++</pre>	<pre>12 34</pre>

PARTICIPATION ACTIVITY

11.5.7: Pug conditions and loops.



- 1) Complete the condition to determine if `num` is positive, negative, or zero.



```
if _____  
    p #{num} is positive.  
else if num < 0  
    p #{num} is negative.  
else  
    p #{num} is zero.
```

[Check](#)[Show answer](#)

- 2) Write a while loop to display the numbers from 10 to 1.



```
- let n = 10  
_____  
p= n--
```

[Check](#)[Show answer](#)

- 3) Complete the code to display the average of the `scores` array.



```
- let scores = [86, 90,  
75, 62, 93], total = 0  
each score in scores  
    -total += score  
p Average is _____
```

[Check](#)[Show answer](#)

Mixins

A **mixin** is a Pug function that is defined with the **mixin** keyword. Ex: **mixin myMixin** followed by a block of Pug code defines a mixin. A mixin is called with a plus sign preceding the mixin name. Ex: **+myMixin** calls the mixin named myMixin. Mixins can be defined and called from anywhere within a Pug template.

A mixin can display a code block that is specified under a mixin call by using the **block** keyword inside the function.

Table 11.5.3: Pug mixins.

Description	Pug example	Rendered HTML
No parameters	<pre>mixin list ul li apples li bananas li oranges //- Calling the mixin +list</pre>	<pre> apples bananas oranges </pre>
Using parameters	<pre>mixin item(itemName, className) if className li(class=className)= itemName else li= itemName ul +item("apples") +item("bananas", "important") +item("oranges")</pre>	<pre> apples <li class="important">bananas oranges </pre>

Rest parameters	<pre>//- Accept any number of arguments mixin list(id, ...items) ol(id=id) each item in items li= item +list("groceries", "apples", "bananas", "oranges")</pre>	<pre><ol id="groceries"> apples bananas oranges </pre>
Mixin blocks	<pre>mixin ask(question) p= question //- Insert code block from under mixin call block //- No block +ask("What is 2 + 3?") //- With block +ask("Who is Blaise Pascal?") .note You may consult your textbook.</pre>	<pre><p>What is 2 + 3?</p> <p>Who is Blaise Pascal?</p> <div class="note">You may consult your textbook.</div></pre>

**PARTICIPATION
ACTIVITY**

11.5.8: Pug mixins.



Given the Pug below, match the Pug snippet with the HTML the snippet would render.

```

mixin greet(name)
  if name
    p Greetings, #{name}!
  block
  else
    p Hello, Anonymous!

mixin displayReverse(students)
  - let n = students.length - 1
  ul
    while n >= 0
      li= students[n].name + " - " + students[n].grade
      - n--

mixin displayGrades(students)
  ul
    each student, index in students
      li= student.name + " - " + student.grade

-
const students = [
  { name: "Anna", grade: "B" },
  { name: "Gary", grade: "A" },
  { name: "Maria", grade: "C" } ]

```

If unable to drag and drop, refresh the page.

+greet

+displayGrades(students)

+displayGrades([])

+greet(students[0].name)

+greet(students[1].name)
p How are you?

+displayReverse(students)

<p>Greetings, Anna!</p>

<p>Hello, Anonymous!</p>

<p>Greetings, Gary!</p>
<p>How are you?</p>

```
<ul>
  <li>Anna - B</li>
  <li>Gary - A</li>
  <li>Maria - C</li>
</ul>
```

```
<ul>
  <li>Maria - C</li>
  <li>Gary - A</li>
  <li>Anna - B</li>
</ul>
```

```
<ul></ul>
```

[Reset](#)

Inheritance and includes

Pug allows one template to inherit the content of another template. Developers use inheritance to provide a consistent structure to webpages without repeating shared HTML content in multiple places.

The **block** statement names a block of Pug content that may be replaced by inherited content. The **append** and **prepend** statements append and prepend, respectively, a Pug block to the inherited Pug block. A Pug template that inherits from another template uses the **extend** statement.

In the figure below, layout.pug serves as the parent template for all webpages. If the path to jquery.js needs to be changed or another JavaScript library needs to be included, only layout.pug would need modifying.

Figure 11.5.2: Pug inheritance example.

layout.pug	review.pug	Rendered HTML
<pre> doctype html block title title Default Title script(src='/scripts/jquery.js') body header block header h1 Movie Reviews #content block content </pre>	<pre> extend layout block title title Movie Review append header h3 Popular Movies prepend content p Movie reviewer: Joe Smith block content p Today we review <cite>The Martian</cite>. </pre>	<pre> <!DOCTYPE html> <html> <head> <title>Movie Review</title> <script src="/scripts/jquery.js"> </script> </head> <body> <header> <h1>Movie Reviews</h1> <h3>Popular Movies</h3> </header> <div id="content"> <p>Movie reviewer: Joe Smith</p> <p>Today we review <cite>The Martian</cite>.</p> </div> </body> </html> </pre>

The **include** statement allows a Pug template to import the contents of a text file. The text file may contain a Pug template, JavaScript, CSS, HTML, or any textual content. Unlike inheritance, included content cannot be replaced with other content.

Figure 11.5.3: Pug include example.

example.pug	test.pug	Rendered HTML
<pre>p Before the include. include test.pug p After the include.</pre>	<pre>p This is a test.</pre>	<pre><p>Before the include. </p> <p>This is a test.</p> <p>After the include. </p></pre>

**PARTICIPATION
ACTIVITY**

11.5.9: Pug inheritance and includes.

Refer to the file master.pug below.

```
.news
  block news
    p Gas prices are falling.
```

- 1) What Pug command makes a Pug template inherit from master.pug?
- ☐ include master.pug
 - ☐ inherit master
 - ☐ extend master

- 2) Assume the Pug file below inherits master.pug. What is the rendered HTML?



```
block news
  p Election is heating up.
```

- ☐

```
<div class="news">
  <p>Election is
heating up.</p>
</div>
```
- ☐

```
<div class="news">
  <p>Election is
heating up.</p>
  <p>Gas prices are
falling.</p>
</div>
```
- ☐

```
<div class="news">
  <p>Gas prices are
falling.</p>
</div>
<div class="news">
  <p>Election is
heating up.</p>
</div>
```

3) Assume the Pug file below inherits master.pug. What is the rendered HTML?



```
prepend news
p Election is heating up.
```

- ☐

```
<div class="news">
  <p>Election is
heating up.</p>
</div>
```
- ☐

```
<div class="news">
  <p>Election is
heating up.</p>
  <p>Gas prices are
falling.</p>
</div>
```
- ☐

```
<div class="news">
  <p>Gas prices are
falling.</p>
  <p>Election is
heating up.</p>
</div>
```


4) What is the rendered HTML for the Pug below?



```
include master.pug
p Election is heating up.
```

- ☐

```
<div class="news">
  <p>Gas prices are
  falling.</p>
</div>
```
- ☐

```
<div class="news">
  <p>Gas prices are
  falling.</p>
  <p>Election is
  heating up.</p>
</div>
```
- ☐

```
<div class="news">
  <p>Gas prices are
  falling.</p>
</div>
<p>Election is heating
up.</p>
```

Exploring further:

- [Pug reference](#)
- [Mustache templates with JavaScript](#)
- [EJS: Embedded JavaScript templates](#)

11.6 Relational databases and SQL (Node)

Introduction to relational databases

Web applications often store data in relational or non-relational databases. Relational databases have been popular since the 1970s due to the relative simplicity and efficiency that relational databases offer. This section provides an overview of relational databases and SQL; more in-depth coverage is found elsewhere in this material.

Relational databases store data in "relations", which are visualized as tables. A **table** is a collection of related data that is organized into columns and rows similar to a spreadsheet. The animation below shows a students table that stores information about three students: Susan, Billy, and Alice.

Developers use the **Structured Query Language (SQL)** to manipulate data in a relational database. A **relational database management system (RDBMS)** is a system that manages the data for relational databases and executes SQL statements. Popular RDBMSs include MySQL, PostgreSQL, Oracle, and SQL Server.

PARTICIPATION ACTIVITY

11.6.1: Students table with three rows.

students table

stuld	name	gpa
123	Susan	3.1
456	Billy	2.5
789	Alice	4.0

← columns
} rows

Animation content:

A table called students is shown. The column names are stuld, name, and gpa. The first row contains 123 under stuld, Susan under name, and 3.1 under gpa. The second row contains 456 in the first column, Billy in the second column, and 2.5 in the third column. The third row contains 789 in the first column, Alice in the second column, and 4.0 in the third column.

Animation captions:

1. A table to store student information is called students. The table has three columns for storing student IDs, names, and GPAs.
2. Each student's information is stored in a single row.

**PARTICIPATION
ACTIVITY**

11.6.2: Introduction to relational databases.



Refer to the students table above.

1) What is Billy's GPA?



- ☐ 3.1
- ☐ 2.5
- ☐ 4.0

2) How many rows would the students table have if a new student were added to the table?



- ☐ 2
- ☐ 3
- ☐ 4

3) A new ____ is needed to store each student's address.



- ☐ table
- ☐ column
- ☐ row

4) What language is used to add new students to the students table?



- ☐ SQL
- ☐ RDBMS
- ☐ MySQL

CREATE TABLE statement

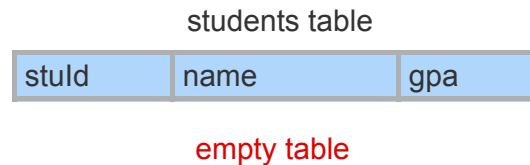
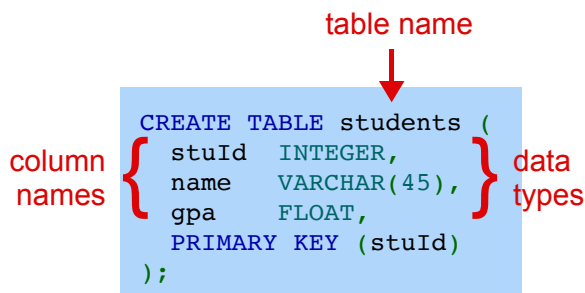
An **SQL statement** is a complete command and is terminated with a semicolon. SQL uses keywords like CREATE, SELECT, WHERE, and others that are often capitalized but are not case sensitive.

The **CREATE TABLE** statement is an SQL statement that creates a table by specifying the table

name, column names, data types, and primary key. The **primary key** is the column or columns that uniquely identify each row in the table. Ex: `stuId` is the primary key of the students table in the animation below because no two students can share the same ID.

**PARTICIPATION
ACTIVITY**

11.6.3: Creating a students table.

**Animation content:**

The following SQL code is shown:

```
CREATE TABLE students (  
  stuld INTEGER,  
  name VARCHAR(45),  
  gpa FLOAT,  
  PRIMARY KEY (stuld)  
);
```

The SQL code creates an empty table called students with column names stuld, name, and gpa.

Animation captions:

1. The CREATE TABLE statement creates a table named "students".
2. The column names and data types are separated by commas.
3. The keyword PRIMARY KEY names the column that must contain unique values.
4. The table contains no rows after being created.

Table 11.6.1: Common data types in SQL.

Data type	Description	Example
CHAR (n)	Fixed-length character string of length n	'CODE123 '
VARCHAR (n)	Variable-length character string with maximum length n	'Super Bowl '
BOOLEAN	TRUE or FALSE	TRUE
INTEGER	Integer number (-2,147,483,648 to 2,147,483,647)	12345
FLOAT	Approximate number with mantissa precision 16	3.14
DATE	Year, month, day: YYYY-MM-DD	'2010-12-25 '
TIME	Hour, minute, second: HH:MM:SS	'14:06:00 '
DATETIME	Year, month, day, hour, minute, second: YYYY-MM-DD HH:MM:SS	'2010-12-25 14:06:00 '

**PARTICIPATION
ACTIVITY**

11.6.4: SQL and creating tables.

1) The words "CREATE TABLE" must be capitalized in a CREATE TABLE statement.

- ☐ True
- ☐ False

2) A table typically has at least one column that uniquely identifies each row.

- ☐ True
☐ False

3) The students table may contain two students with the same stuld.

- ☐ True
☐ False

4) The students table may contain two students with the same name.

- ☐ True
☐ False

5) The number 5 billion may be stored in an INTEGER column.

- ☐ True
☐ False

6) The ____ data type is ideal for creating a column that stores a student's date of birth.

- ☐ DATETIME
☐ DATE

7) The students table is empty after the table is created.

- ☐ True
☐ False

Auto-increment columns

ID columns are frequently created as auto-increment columns. An **auto-increment column** is a column that is assigned an automatically incrementing value. Ex: An auto-incrementing column may be assigned 1, 2, 3, etc. for each row that is inserted into the table.

Each RDMS defines auto-increment columns differently. MySQL uses the **AUTO_INCREMENT** keyword to define an auto-increment column. The example below creates a students table with stuld as an auto-increment column.

```
CREATE TABLE students (  
  stuId INTEGER AUTO_INCREMENT,  
  name VARCHAR(45),  
  gpa FLOAT,  
  PRIMARY KEY (stuId));
```

INSERT statement

SQL provides four statements (or commands) that perform CRUD operations on a table:

- Create - Create new rows in a table with an INSERT statement
- Retrieve - Retrieve all rows that meet some criteria with a SELECT statement
- Update - Update the data in a row with an UPDATE statement
- Delete - Delete rows from a table with a DELETE statement

An **INSERT** statement inserts one or more rows into a table. The values composing the rows are listed after the VALUES keyword in parentheses. String values must be delimited with single quotation marks.

PARTICIPATION ACTIVITY

11.6.5: INSERT statement.



```
INSERT INTO students VALUES  
(123, 'Susan', 3.1),  
(456, 'Billy', 2.5),  
(987, 'Alice', 4.0);
```

students table

stuld	name	gpa
123	Susan	3.1
456	Billy	2.5

987

Alice

4.0

Animation content:

The following SQL code is shown:

```
INSERT INTO students VALUES  
  (123, 'Susan', 3.1),  
  (456, 'Billy', 2.5),  
  (987, 'Alice', 4.0);
```

The code adds three rows to the students table. The column names are stuld, name, and gpa. The first row contains 123, Susan, and 3.1. The second row contains 456, Billy, and 2.5. The third row contains 789, Alice, and 4.0.

Animation captions:

1. INSERT statement indicates the table that will receive the data.
2. Row values to insert are listed in the same order as the table's columns.
3. Each set of values is inserted as a row in the table.

PARTICIPATION ACTIVITY

11.6.6: Insert your favorite movie.



The given SQL creates a movie table and inserts some movies. The SELECT statement selects all movies from the movie table.

Press the Run button to produce a result table. Verify the result table displays five movies.

Modify the INSERT statement to add your favorite movie with the ID 6. Then run the SQL again and verify your new movie appears in the result table.


```
1 CREATE TABLE movie (  
2   id INTEGER,  
3   title VARCHAR(100),  
4   rating VARCHAR(5),  
5   release_date DATE  
6 );  
7  
8 -- Add your favorite movie  
9 INSERT INTO movie VALUES  
10  (1, 'Rogue One: A Star Wars Story', 'PG-13', '2016-12-10'),  
11  (2, 'Hidden Figures', 'PG', '2017-01-06'),  
12  (3, 'Toy Story', 'G', '1995-11-22'),  
13  (4, 'Avengers: Endgame', 'PG-13', '2019-04-26'),  
14  (5, 'The Godfather', 'R', '1972-03-14');  
15  
16 SELECT *  
17 FROM movie;  
18
```

[Run](#)[Reset code](#)[► View solution](#)**PARTICIPATION
ACTIVITY**

11.6.7: INSERT statement.



All but one of the INSERT statements below has a syntax error. Match the INSERT statement with the statement's error description.

If unable to drag and drop, refresh the page.

```
INSERT INTO students VALUES ('Ebony', 555, 3.9);
```

```
INSERT INTO students VALUES (777, 'O'Malley', 3.0);
```

```
INSERT INTO students VALUES 222, 'Darnell', 3.0;
```

```
INSERT INTO students (stuld, name) VALUES (888, 'Wang');
```

```
INSERT INTO students VALUES (444, Zack, 3.2);
```

Missing single quotation marks.

Data is not in the correct order.

Missing parenthesis around data values.

Illegal use of single quotation mark.

Syntactically correct statement.

Reset

Inserting with auto-incrementing columns

If a table uses an auto-incrementing column, the INSERT statement should not specify a value for the auto-increment column. The example below assumes the stuld column is auto-incrementing, so no stuld is specified. The stuld will be assigned the next unused integer.

```
INSERT INTO students (name, gpa)
VALUES ('MingMing', 2.9);
```

SELECT statement

The **SELECT** statement retrieves data from a table. The column names listed after "SELECT" are included in the retrieval result. If "*" is given as a column name then all columns are selected.

The SELECT statement retrieves all rows by default, but the WHERE clause limits the rows that are retrieved. The **WHERE** clause works like an if statement in a programming language, specifying conditions that must be true for a row to be selected.

A SELECT statement may use the **ORDER BY** clause to sort selected rows in ascending (alphabetic) order. SQL also defines a number of functions that may be used in a SELECT statement. Ex: The **COUNT()** function counts how many rows are returned by a SELECT statement.

PARTICIPATION ACTIVITY

11.6.8: SELECT statement.



```
SELECT *  
FROM students;
```

Selected rows

stuld	name	gpa
123	Susan	3.1
456	Billy	2.5
987	Alice	4.0

```
SELECT name, gpa  
FROM students  
WHERE gpa > 3;
```

Selected rows

name	gpa
Susan	3.1
Alice	4.0

students table

stuld	name	gpa
123	Susan	3.1
456	Billy	2.5
987	Alice	4.0

```
SELECT COUNT(*)  
FROM students  
WHERE name != 'Alice';
```

Selected rows

COUNT(*)
2

Animation content:

A table called students is shown with columns stuld, name, and gpa. The first row contains 123, Susan, and 3.1. The second row contains 456, Billy, and 2.5. The third row contains 789, Alice, and 4.0. Three blocks of SQL code are shown. The first:

```
SELECT *  
FROM students;
```

This code block retrieves all rows from the students table. The second code block:

```
SELECT name, gpa  
FROM students  
WHERE gpa > 3;
```

This code block selects two rows from students. The first row contains Susan under name and 3.1 under gpa. The second row contains Alice in the first column and 4.0 in the second column. The third code block:

```
SELECT COUNT(*)  
FROM students  
WHERE name != 'Alice';
```

This code block examines rows in students and adds to a count if the name is not Alice. 2 is returned in a row.

Animation captions:

1. The SELECT statement has an asterisk, which selects data from all columns. The FROM clause names the table from which data is selected.
2. All rows are retrieved from the table by default.
3. The SELECT statement selects only the name and gpa columns. The WHERE clause selects data from rows that have a gpa > 3.
4. Each row in the students table is examined, and only Susan and Alice have a gpa > 3.0.
5. The COUNT(*) function counts how many rows are selected.
6. Two rows do not have "Alice" in the name column, so a single row with the count is returned.

Table 11.6.2: WHERE condition operators.

Operator	Description	Example
=	Compares two values for equality	1 = 2 is false
!=	Compares two values for inequality	1 != 2 is true
<	Compares two values with <	2 < 2 is false
<=	Compares two values with ≤	2 <= 2 is true
>	Compares two values with >	3 > 3 is false
>=	Compares two values with ≥	3 >= 3 is true
AND	Performs logical AND between two conditions	1 > 2 AND 2 < 3 is false
OR	Performs logical OR between two conditions	1 > 2 OR 2 < 3 is true

**PARTICIPATION
ACTIVITY**

11.6.9: Select movies with WHERE clause.



The given SQL creates a movie table and inserts some movies. The SELECT statement selects all movies released before January 1, 2000.

Modify the SELECT statement to select the title and release date of PG-13 movies that were released after January 1, 2008.

Run your solution and verify the result table shows just the title and release date columns for *The Dark Knight* and *Crazy Rich Asians*.

```
1 CREATE TABLE movie (  
2     movie_id INT AUTO_INCREMENT,  
3     title VARCHAR(100),  
4     rating CHAR(5),  
5     release_date DATE,  
6     PRIMARY KEY (movie_id )  
7 );  
8  
9 INSERT INTO movie (title, rating, release_date) VALUES  
10 ('Casablanca', 'PG', '1943-01-23'),  
11 ('Bridget Jones\'s Diary', 'PG-13', '2001-04-13'),  
12 ('The Dark Knight', 'PG-13', '2008-07-18'),  
13 ('Hidden Figures', 'PG', '2017-01-06'),  
14 ('Toy Story', 'G', '1995-11-22'),  
15 ('Rocky', 'PG', '1976-11-21'),  
16 ('Crazy Rich Asians', 'PG-13', '2018-08-15');  
17  
18 -- Modify the SELECT statement:  
19 SELECT *  
20 FROM movie  
21 WHERE release_date < '2000-01-01';  
22
```

[Run](#)[Reset code](#)[► View solution](#)**PARTICIPATION
ACTIVITY**

11.6.10: SELECT statement.



Refer to the shows table created below.

```
CREATE TABLE shows (  
    showId INTEGER,  
    title VARCHAR(45),  
    seasons INTEGER,  
    releaseDate DATE,  
    PRIMARY KEY (showId)  
);  
  
INSERT INTO shows VALUES  
    (111, 'Grey\'s Anatomy', 12, '2005-03-27'),  
    (222, 'Arrow', 5, '2012-10-10'),  
    (333, 'The Blacklist', 4, '2013-09-23'),  
    (444, 'Parks and Recreation', 7, '2009-04-09');
```

- 1) What shows are selected by the following query?



```
SELECT *  
FROM shows;
```

- ☐ All shows
- ☐ Arrow
- ☐ No shows

- 2) In what order are the shows retrieved by the following query?



```
SELECT *  
FROM shows  
ORDER BY title;
```

- ☐ Grey's Anatomy, Arrow, The Blacklist, Parks and Recreation
- ☐ Arrow, Grey's Anatomy, Parks and Recreation, and The Blacklist
- ☐ The Blacklist, Parks and Recreation, Grey's Anatomy, Arrow

- 3) What shows are selected by the following query?



```
SELECT *  
FROM shows  
WHERE seasons < 5;
```

- ☐ All shows
- ☐ Arrow and The Blacklist
- ☐ The Blacklist

- 4) What shows are selected by the following query?



```
SELECT *  
FROM shows  
WHERE seasons <= 10 AND  
seasons >= 5;
```

- ☐ All shows
 - ☐ Arrow and Parks and Recreation
 - ☐ Grey's Anatomy
- 5) What number is returned by the following query?



```
SELECT COUNT(*)  
FROM shows;
```

- ☐ 0
 - ☐ 4
 - ☐ 1
- 6) The `DATEDIFF()` function returns the number of days between two dates. What shows are selected by the following query?



```
SELECT *  
FROM shows  
WHERE DATEDIFF('2016-08-01',  
releaseDate) > 365 * 10;
```

- ☐ All shows
- ☐ No shows
- ☐ Grey's Anatomy

7) The **YEAR()** function extracts the year from the given date. What is selected by the following query?

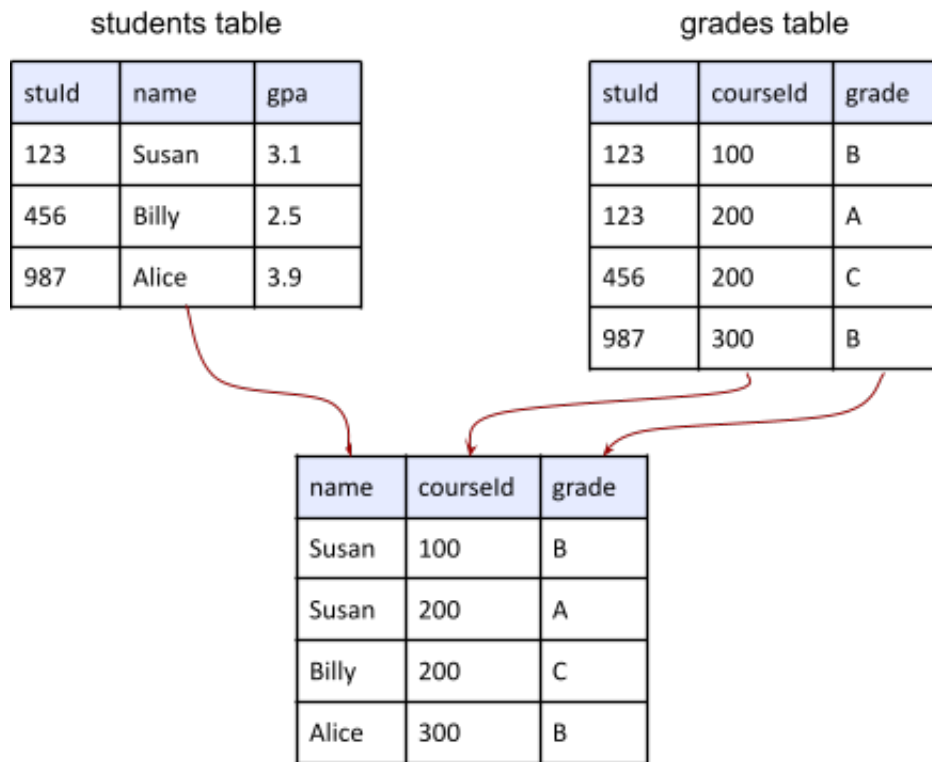


```
SELECT title,  
YEAR(releaseDate)  
FROM shows;
```

- ☐ Grey's Anatomy 2005
- ☐ 2005, 2012, 2013, 2009
- ☐ Grey's Anatomy 2005, Arrow
- ☐ 2012, The Blacklist 2013, Parks and Recreation 2009

Joining data from multiple tables.

Most databases are comprised of multiple tables with relationships between the tables. Ex: The grades table below assigns letter grades to students in each course identified by a courseId. To retrieve the student's name, course ID, and grade of all students requires the students and grades tables be joined. The **SELECT** statement uses a **JOIN** clause with an **ON** clause to join data in multiple tables.



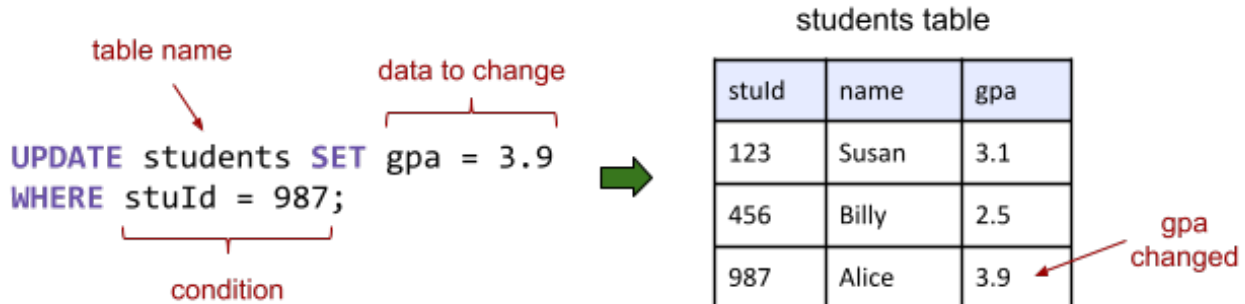
```
SELECT name, courseId, grade
FROM students
JOIN grades ON students.stuId = grades.stuId;
```

The example above joins the name column from the students table and the courseId and grade columns from the grades table into a single table that matches the stuld columns from both tables. Joins allow developers to write complex queries that answer questions like, "What is the name of the student who performed the best in all courses?" and "Which students were not assigned a passing grade?" Joins are covered in more detail elsewhere in this material.

UPDATE and DELETE statements

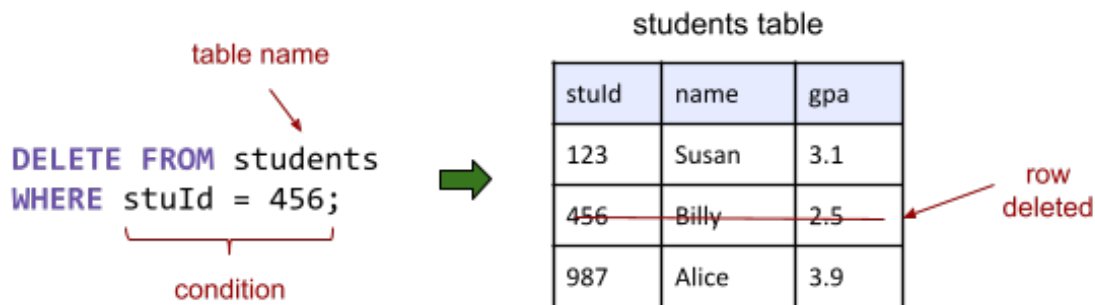
The **UPDATE** statement changes the values of existing rows in a table. The WHERE clause determines which rows are updated.

Figure 11.6.1: UPDATE statement changes Alice's GPA to 3.9.



The **DELETE** statement removes rows from a table. The WHERE clause determines which rows are removed.

Figure 11.6.2: DELETE statement removes Billy from the students table.



PARTICIPATION ACTIVITY

11.6.11: Update the song table.

The given SQL creates a song table and inserts three songs.

Write three UPDATE statements to make the following changes:

- Change the title from 'One' to 'With Or Without You'.
- Change the artist from 'The Righteous Brothers' to 'Aretha Franklin'.
- Change the release years of all songs after 1990 to 2021.

Run your solution and verify the songs in the result table reflect the changes above.

```
1 CREATE TABLE song (  
2   song_id INT,  
3   title VARCHAR(60),  
4   artist VARCHAR(60),  
5   release_year INT,  
6   PRIMARY KEY (song_id)  
7 );  
8  
9 INSERT INTO song VALUES  
10  (100, 'Blinding Lights', 'The Weeknd', 2019),  
11  (200, 'One', 'U2', 1991),  
12  (300, 'You\'ve Lost That Lovin\' Feeling', 'The Righteous Brothers'  
13  (400, 'Johnny B. Goode', 'Chuck Berry', 1958);  
14  
15  -- Write your UPDATE statements here:  
16  
17  
18  
19 SELECT *  
20 FROM song;
```

[Run](#)[Reset code](#)[► View solution](#)

PARTICIPATION ACTIVITY

11.6.12: UPDATE and DELETE statement.



Use the table below to match the UPDATE or DELETE statement with the statement's effect.

shows table

showId	title	seasons	releaseDate
555	Lost	6	2004-11-22
666	The X-Files	10	1993-11-10
777	Seinfeld	9	1989-07-05
888	Arrested Development	4	2003-11-02

If unable to drag and drop, refresh the page.

UPDATE shows SET title = 'The Office' WHERE showId = 999;

UPDATE shows SET title = 'The Office' WHERE seasons > 6;

UPDATE shows SET title = 'The Office', seasons = 9;

DELETE FROM shows WHERE MONTH(releaseDate) = 7;

DELETE FROM shows;

DELETE FROM shows WHERE title LIKE '%Files';

The X-Files and Seinfeld are assigned the title "The Office".

No shows are updated.

All shows are assigned the title "The Office" and season set to 9.

All shows are removed.

Seinfeld is removed from the table.

The X-Files is removed from the table.

Reset

Exploring further:

- [SQL Cheat Sheet](#)
- [SQL LIKE operator](#)

11.7 MySQL (Node)

MySQL Introduction

MySQL is a popular RDBMS that offers an open source version and a few commercial versions that provide additional functionality.

Developers often install one RDBMS on a local machine for development and another RDBMS on a web server for the deployed website. MySQL can be downloaded from mysql.com and installed on a variety of operating systems, including Unix, Linux, Windows, and OS X. Detailed instructions for installing MySQL are provided on the MySQL website.

MySQL server has a user account assigned administrative roles that allows the **database administrator** to create user accounts and databases and perform other administrative activities. MySQL server normally runs as a service or daemon, so MySQL is always running.

PARTICIPATION ACTIVITY

11.7.1: Introduction to MySQL.

1) MySQL is a popular choice for web hosting companies that provide an RDBMS for their customers.

- ☐ True
- ☐ False

2) Web applications typically use the same physical database in both the development environment and deployment environment .

- ☐ True
- ☐ False

3) Each MySQL user is assigned a username and password.

- ☐ True
- ☐ False

mysql tool

The **mysql command-line tool** (sometimes called the "terminal monitor" or "monitor") is a command-line program that allows developers to connect to a MySQL server, perform administrative functions, and execute SQL statements.

The mysql tool attempts by default to connect to the MySQL server running on the same machine. The `-u` parameter indicates the username, and `-p` indicates that the user will enter a password. The connection to the local MySQL server is only established if the proper username and password are entered.

PARTICIPATION ACTIVITY

11.7.2: Using mysql tool to connect to MySQL server.

```
$ mysql -u bsmith -p
Enter password:*****
Welcome to the MySQL monitor.
...
Type 'help;' or '\h' for help.
mysql>
```

username: bsmith
password: mypass

bsmith

← mysql tool
Successful
login!

MySQL server

Animation content:

MySQL server is shown with the following information:

username: bsmith

password: mypass

A console is shown where the user runs the mysql tool and inputs their username and password with the following command line prompt:

```
mysql -u bsmith -p
```

The user is then prompted to enter their password. The username and password are sent to MySQL server, which indicates the login is successful. Then the console displays:
Welcome to the MySQL monitor.

...

Type 'help;' or '\h' for help.

```
mysql<
```

Animation captions:

1. The MySQL database administrator creates a user account with a username and password.
2. The user bsmith opens a command-line window on the machine running MySQL and runs the mysql tool.
3. The mysql tool prompts the user for bsmith's password.
4. Using the username and password, the mysql tool attempts to connect to the MySQL server.
5. If the correct username and password are given, the mysql tool successfully connects and displays the welcome screen and mysql prompt.

PARTICIPATION ACTIVITY

11.7.3: Connecting with mysql tool.



- 1) The mysql tool must be supplied a username and password to connect to MySQL server.



- ☐ True
☐ False

- 2) What does a developer with username "manning" type to start the mysql tool?



- ☐ mysql -p manning
☐ mysql -u manning -p

3) The mysql tool attempts to connect to MySQL server running on the same machine as the tool.

- ☐ True
- ☐ False

Creating and using a database

A **MySQL database** is a collection of tables. The database administrator can create databases for individual applications and can give individual users permission to create databases. Various mysql tool commands work with databases:

- The **CREATE DATABASE** command creates a new database.
- The **SHOW DATABASES** command shows all the databases.
- The **USE** command selects a database to use.

Commands are case insensitive and must be terminated by a semicolon.

Figure 11.7.1: Creating a database called "test" and using the "test" database.

```
mysql> create database test;
Query OK, 1 row affected (0.00
sec)

mysql> show databases;
+-----+
| Database |
+-----+
| test     |
+-----+
1 rows in set (0.01 sec)

mysql> use test;
Database changed
```

Once a database is selected, a table can be created for the database using the CREATE SQL command. The **SHOW TABLES** command shows all the tables in the database.

Figure 11.7.2: Creating a table and showing all the tables in the test database.

```
mysql> create table students (stuId int, name varchar(45), gpa float, primary key
(stuId));
Query OK, 0 rows affected (0.37 sec)

mysql> show tables;
+-----+
| Tables_in_test |
+-----+
| students       |
+-----+
1 row in set (0.00 sec)
```

Other SQL commands like SELECT, INSERT, UPDATE, and DELETE may be executed on the database's tables.

Figure 11.7.3: Inserting, selecting, and updating a student.

```
mysql> insert into students values (123, 'Susan',
3.1);
Query OK, 1 row affected (0.08 sec)

mysql> select * from students;
+-----+-----+-----+
| stuId | name  | gpa  |
+-----+-----+-----+
| 123   | Susan | 3.1  |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> update students set gpa = 2.5 where stuId =
123;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

The **QUIT** command exits the mysql tool. All modifications to the database are saved, so the user does not need to re-create the database or tables when running the mysql tools again.

PARTICIPATION ACTIVITY

11.7.4: Entering commands.



Assume a music database contains a songs table with columns songId, title, artist, and year.

- 1) Enter the command to select the music database.

**Check**[Show answer](#)

- 2) Enter the command to display all the tables in the music database.

**Check**[Show answer](#)

- 3) Complete the SQL command to display all the songs in the songs table.



_____ * from songs;

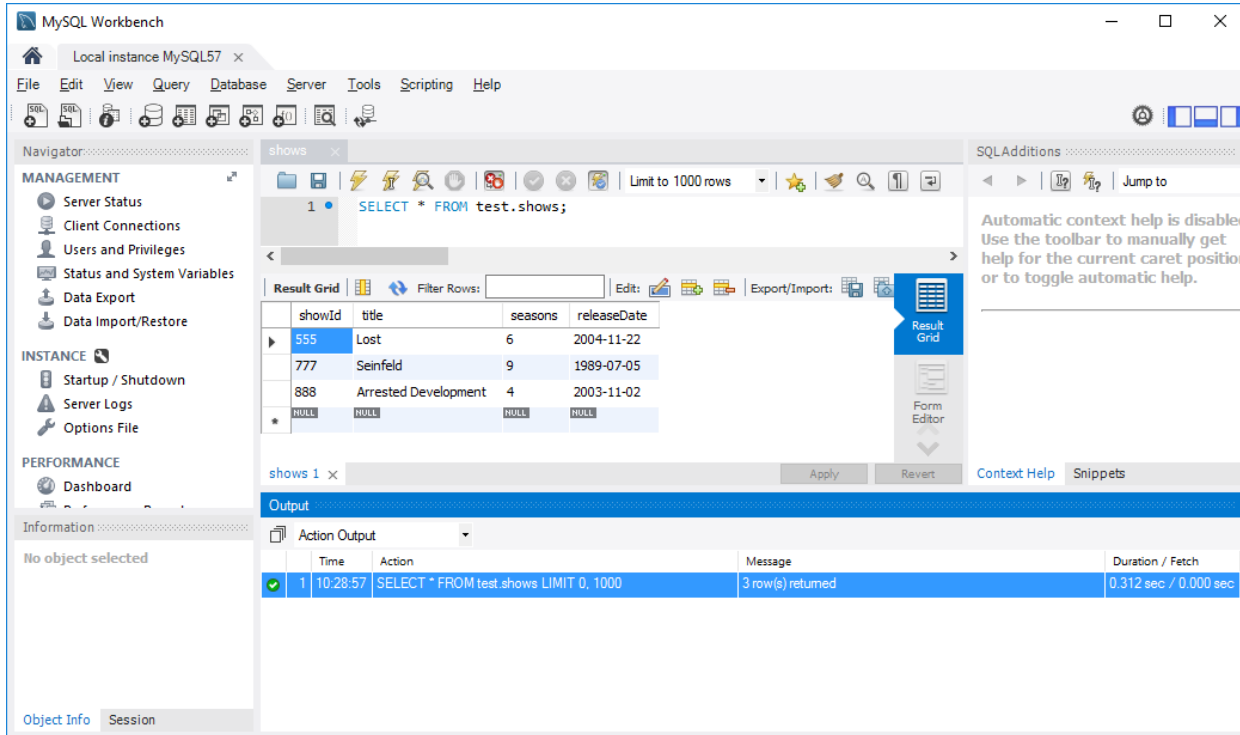
Check[Show answer](#)

- 4) Enter the command to exit the mysql tool.

**Check**[Show answer](#)

MySQL Workbench

Some developers prefer to use a GUI application to interact with a MySQL server. MySQL Workbench allows developers to execute SQL commands using an SQL editor.



Errors

MySQL server returns an **error code** and description when a SQL statement is syntactically incorrect or the database cannot execute the statement.

Figure 11.7.4: Improperly formatted SELECT returns error code 1064, and INSERT with duplicate student ID returns error code 1062.

```
mysql> select from students;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to
your MySQL server version for the right syntax to use near 'from students' at line 1

mysql> insert into students values (123, 'George', 2.9);
ERROR 1062 (23000): Duplicate entry '123' for key 'PRIMARY'
```

**PARTICIPATION
ACTIVITY**

11.7.5: Common MySQL error codes.



The SQL statements operate on the shows table below. Match the SQL statement with the resulting MySQL error code.

shows table

showId	title	seasons	releaseDate
555	Lost	6	2004-11-22
666	The X-Files	10	1993-11-10
777	Seinfeld	9	1989-07-05
999	Arrested Development	4	2003-11-02

If unable to drag and drop, refresh the page.

INSERT into shows VALUES (999, 'Stranger Things', 2, '2016-07-15');

UPDATE shows SET genre = 'reality' WHERE seasons > 6;

INSERT into shows VALUES (888, 'Stranger Things', 2);

UPDATE title = 'The Office' WHERE showId = 999;

INSERT into showsTable VALUES (888, 'Stranger Things', 2, '2016-07-15');

1054: Unknown column name

1064: Parsing error, bad syntax

1136: Number of values given in
INSERT does not match number of
columns

1062: Duplicate entry for primary key

1146: Table doesn't exist

Reset

Exploring further:

- [MySQL Documentation](#) from MySQL
- [MySQL Workbench](#) from MySQL
- [MySQL error codes and messages](#) from MySQL

11.8 mysql module (Node)

Installing and connecting

The **mysql module** allows Node.js applications to interface with a MySQL database. The mysql module is installed with the command: `npm install mysql`.

The **mysql.createConnection()** method creates a connection object using an object parameter that specifies the hostname of the computer running MySQL, the developer's username and password, and the database to select.

The **connection object** provides methods to interact with the MySQL server. The connection object's **connect()** method attempts to establish the database connection. **connect()** has a callback function that is passed an error object set to a non-null value if the connection is unsuccessful. An unsuccessful connection occurs if the server is not running, the username/password are incorrect, the database does not exist, etc.

Figure 11.8.1: Connecting to a MySQL server running on localhost.

```
const mysql = require("mysql");

const conn = mysql.createConnection({
  host:      "localhost",
  user:      "myusername",
  password:  "mypassword",
  database:  "test"
});

conn.connect(function(err) {
  if (err) {
    console.log("Error connecting to MySQL:",
err);
  }
  else {
    console.log("Connection established");
  }
});
```

**PARTICIPATION
ACTIVITY**

11.8.1: Connecting to MySQL server.

Refer to the figure above.

1) The require statement won't work unless mysql module is first installed.

- ☐ True
☐ False

2) Calling `mysql.createConnection()` establishes a connection to a MySQL server.

- ☐ True
☐ False

3) The `connect ()` is passed a non-null error object if the database "test" does not exist.

- ☐ True
- ☐ False

SELECT statement

After establishing a connection with the MySQL server, SQL statements can be executed using the connection object's **`query()`** method, which has two parameters:

1. An SQL statement
2. A callback function with two parameters:
 1. An error object that is set if an error occurs
 2. Results of the SQL statement

PARTICIPATION ACTIVITY

11.8.2: Selecting rows from the students table.

```
const sql = "SELECT stuId, name, gpa FROM students";
conn.query(sql, function(err, rows) {
  if (err) {
    console.log("ERROR:", err);
  }
  else {
    // Loop through all the rows returned from SELECT statement
    rows.forEach(function(row) {
      console.log(row.name + " = " + row.gpa);
    });
  }
});
```

console

Susan = 3.1
Billy = 2.5
Alice = 4.0

MySQL server

students		
stuld	name	gpa
123	Susan	3.1
456	Billy	2.5
789	Alice	4.0

conn

rows

```
[0] { stuld:123, name:"Susan", gpa:3.1 }
[1] { stuld:456, name:"Billy", gpa:2.5 }
[2] { stuld:789, name:"Alice", gpa:4.0 }
```


Animation content:

A block of JavaScript code is shown:

```
const sql = "SELECT stuld, name, gpa FROM students";
conn.query(sql, function(err, rows) {
  if (err) {
    console.log("ERROR:", err);
  }
  else {
    // Loop through all the rows returned from SELECT statement
    rows.forEach(function(row) {
      console.log(row.name + " = " + row.gpa);
    });
  }
});
```

MySQL server is shown with a students table and columns stuld, name, and gpa. The first row is 123, Susan, and 3.1. The second row is 456, Billy, and 2.5. The third row is 789, Alice, and 4.0. The connection object conn is displayed that selects rows from the table. The following data is returned:

```
[0] { stuld:123, name:"Susan", gpa:3.1 }
[1] { stuld:456, name:"Billy", gpa:2.5 }
[2] { stuld:789, name:"Alice", gpa:4.0 }
```

When the else block executes, the following is printed to the console:

```
Susan = 3.1
Billy = 2.5
Alice = 4.0
```

Animation captions:

1. MySQL server has a students table with three rows.
2. conn is the connection object, which is used to interact with the MySQL database. query() is passed a string with a SELECT statement that selects all rows from the students table.
3. The students table's three rows are returned to the query() callback function as an array of objects.
4. The query is free of errors and executed correctly, so err is null, and the else block executes.
5. The forEach() loops through the rows array and outputs each student's name and GPA to the console.

**PARTICIPATION
ACTIVITY**

11.8.3: SELECT query.



Refer to the animation above.

1) Which of the following is a reason the `query()` method could execute the callback function with an error object that is not null?



- ☐ No rows exist in the students table.
- ☐ No table named "students" exists.
- ☐ The students table contains too many rows.

2) If the students table contains 5 students, how many lines of console output does the `query()` callback function output?



- ☐ 0
- ☐ 3
- ☐ 5

3) What statement could be added to the for each loop to output each student's ID?



- ☐ `console.log(stuId);`
- ☐ `console.log(rows.stuId);`
- ☐ `console.log(row.stuId);`

Connection pool

A Node.js application may keep a connection to the database open the entire time the Node.js application is running. However, database connections are often terminated by the MySQL server at periodic intervals or when the MySQL server is restarted. A developer can use a connection pool to circumvent connection timeouts and to enable Node.js applications to execute more than one SQL command at a time. More about the connection pool can be found in the [mysql module documentation](#).

Query with values array

The `query()` method accepts an optional values array, which holds values that are substituted for "?" characters in a SQL command.

PARTICIPATION ACTIVITY

11.8.4: Values substituted into SELECT query.



```
conn.query("SELECT name, gpa FROM students WHERE gpa >= ? OR name = ?", [3.0, "Bob"],  
          function(err, rows) {  
              3.0          'Bob'  
              ...  
          });
```

`SELECT name, gpa FROM students WHERE gpa >= 3.0 OR name = 'Bob'`

Animation content:

The following JavaScript code is shown:

```
conn.query("SELECT name, gpa FROM students WHERE gpa >= ? OR name = ?", [3.0, "Bob"],  
          function(err, rows) {  
              ...  
          });
```

The following SQL query is sent to MySQL:

`SELECT name, gpa FROM students WHERE gpa >= 3.0 OR name = 'Bob'`

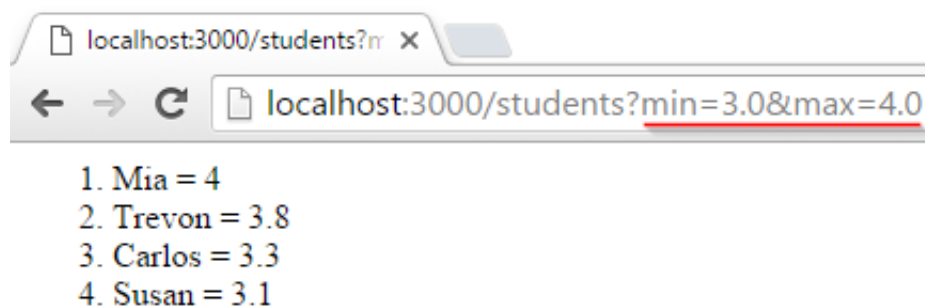
Animation captions:

1. The optional values array contains data to be inserted into the SQL statement.
2. Array values are matched to "?" characters in the SQL query.
3. SQL query with substituted values is sent to MySQL.

The ability to substitute values into SQL statements allows developers to execute SQL statements based on user input. In the figure below, the URL query string parameters are substituted into the SQL statement so only students who have a GPA between the **min** and **max** query string parameters are displayed.

Figure 11.8.2: Express route displaying students with GPAs between two values obtained from the query string.

```
app.get("/students", function(req, res) {  
    // Get values from the query string  
    const minGpa = req.query.min;  
    const maxGpa = req.query.max;  
  
    // Get ordered list of students by gpa  
    conn.query("SELECT name, gpa FROM students WHERE " +  
        "gpa >= ? and gpa <= ? ORDER BY gpa DESC",  
        [minGpa, maxGpa], function(err, rows) {  
  
        if (err) {  
            console.log("ERROR:", err);  
        }  
        else {  
            // Produce ordered list of students  
            let html = "<ol>";  
            rows.forEach(function(row) {  
                html += "<li>" + row.name + " = " + row.gpa + "  
</li>";  
            });  
            html += "</ol>";  
            res.send(html);  
        }  
    });  
});
```



SQL injection

Web applications that use input from a user in an SQL statement are susceptible to SQL injection attacks. A **SQL injection attack** is when a malicious user enters input into a web application that alters the intent of a SQL statement. SQL injection attacks can lead to sensitive data being divulged or deleted. The mysql module's behavior of substituting values into a SQL statement helps circumvent SQL injection attacks.

PARTICIPATION ACTIVITY

11.8.5: Select with values array.

- 1) If a `query()` SQL statement contains 3 question marks, the values array must contain 3 values.

- ☐ True
☐ False

- 2) The following `query()` call outputs Billy's GPA.

```
conn.query("SELECT gpa FROM
students WHERE name = '?'",
  ["Billy"], function(err,
rows) {

console.log(rows[0].gpa);
});
```

- ☐ True
☐ False

- 3) What does the following code display if the query string contains **name=Susan**, but Susan is not in the students table?



```
conn.query("SELECT gpa FROM
students WHERE name = ?",
[req.query.name],
function(err, rows) {
    if (err) {
        console.log("Error");
    }
    else if (rows.length >
0) {
        console.log(rows[0].gpa);
    }
    else {
        console.log("No");
    }
});
```

- ☐ Error
- ☐ No

INSERT statement

The `query()` method's values array may also be a map containing column names and values, which is helpful when executing an INSERT statement with a SET clause. After executing an INSERT, UPDATE, or DELETE statement, the property **result.affectedRows** indicates the number of rows the query inserted, updated, or deleted.

The figure below outputs to the console "Inserted 1 row" when the INSERT statement inserts Maria into the students table. If a student with ID 777 already exists, MySQL returns error code 1062, indicating that the INSERT failed because a duplicate value was used for the primary key.

Figure 11.8.3: query() uses INSERT with map.

```
const student = { stuId: 777, name: "Maria", gpa: 3.3 };
conn.query("INSERT INTO students SET ?", student, function(err, result)
{
  if (err) {
    if (err.errno === 1062) {
      console.log("Cannot insert duplicate ID " + student.stuId);
    }
    else {
      console.log("ERROR:", err);
    }
  }
  else {
    console.log("Inserted " + result.affectedRows + " row"); //
    Success!
  }
});
```

The students table used in the example above requires the INSERT statement to supply a unique student ID. However, many databases use tables where the primary key is automatically chosen by the database. An **auto-increment field** is a database field that increments by one each time a row is inserted into the table. MySQL defines an **AUTO_INCREMENT** attribute that is used in a CREATE statement to generate a unique ID for new rows.

Figure 11.8.4: CREATE with AUTO_INCREMENT attribute creates unique student IDs for INSERT.

```
mysql> create table students (stuId int AUTO_INCREMENT, name varchar(45), gpa float,
primary key (stuId));

mysql> insert into students (name, gpa) values ('Susan', 3.1), ('Trevon', 3.8),
('Carlos', 3.3);

mysql> select * from students;
```

stuId	name	gpa
1	Susan	3.1
2	Trevon	3.8
3	Carlos	3.3

When the primary key is an auto-incrementing ID field, the INSERT statement does not normally

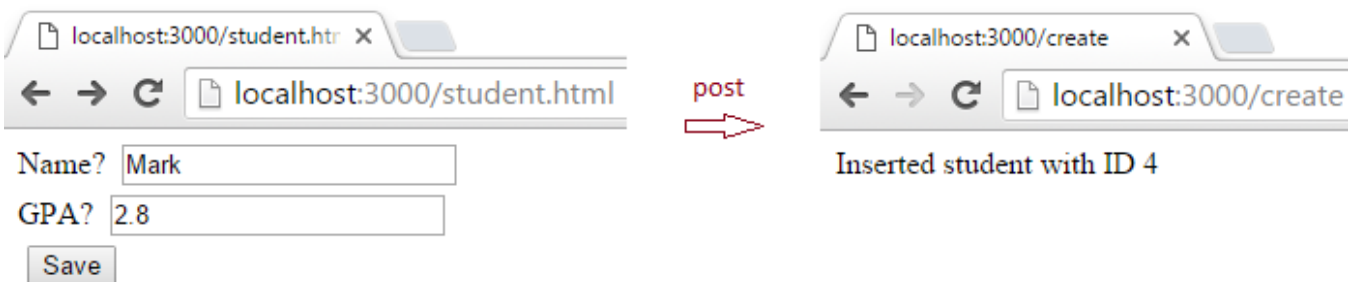
specify an ID. If a web application needs to know the value of the auto-incremented ID after the INSERT, the mysql module provides the new ID in ***result.insertId***.

Figure 11.8.5: Express route that inserts a new student into the students table and displays the student's ID.

```
<!-- student.html -->
<form method="post" action="/create">
  <p>
    <label>Name?</label>
    <input type="text" name="name">
  </p>
  <p>
    <label>GPA?</label>
    <input type="text" name="gpa">
  </p>
  <input type="submit" value="Save">
</form>
```

```
app.post("/create", function(req, res) {
  // Get values from the form
  const student = { name: req.body.name, gpa: req.body.gpa };

  conn.query("INSERT INTO students SET ?", student, function(err,
result) {
    if (err) {
      res.send(err);
    }
    else {
      res.send("Inserted student with ID " + result.insertId);
    }
  });
});
```



- 1) Which INSERT statement matches the query below?



```
conn.query("INSERT INTO
animals SET ?",
  { id: 101, name: "Zebra" },
  function(err, result) {
    ...
  });
```

- ☐ INSERT into animals SET (101, 'Zebra')
 - ☐ INSERT into animals SET id=101, name='Zebra'
 - ☐ INSERT into animals VALUES (101, 'Zebra')
- 2) The code below returns a 1062 error code. What should the developer do to fix the INSERT statement?



```
conn.query("INSERT INTO
animals VALUES (101,
'Zebra')",
  function(err, result) {
    ...
  });
```

- ☐ Add the missing values to the INSERT statement.
 - ☐ Change the order of the animal ID and name.
 - ☐ Use an ID that is not already in the animals table.
- 3) Can an INSERT statement specify an ID when the ID column is an auto-increment column?
- ☐ Yes
 - ☐ No



4) In the figure above, Mark is inserted into the students table and is assigned the ID 4. What will `result.insertId` be when the next student is inserted into the students table?

- ☐ 4
- ☐ 5
- ☐ 6

UPDATE and DELETE statements

When the `query()` method executes UPDATE and DELETE statements, `result.affectedRows` contains the number of rows that were affected by the UPDATE or DELETE.

Figure 11.8.6: `.query()` method executing an UPDATE statement.

```
conn.query("UPDATE students SET gpa = ? WHERE stuId = ?", [2.8, 777],
function(err, result) {
    if (err) {
        console.log("ERROR:", err);
    }
    else {
        console.log("Updated " + result.affectedRows + " rows");
    }
});
```

Figure 11.8.7: `.query()` method executing a DELETE statement.

```
conn.query("DELETE FROM students WHERE stuId = ?", [777], function(err,
result) {
    if (err) {
        console.log("ERROR:", err);
    }
    else {
        console.log("Deleted " + result.affectedRows + " rows");
    }
});
```

**PARTICIPATION
ACTIVITY**

11.8.7: UPDATE and DELETE statements.

- 1) `result.affectedRows` is 0 if the UPDATE or DELETE statement did not update or delete any rows.
- ☐ True
- ☐ False
- 2) `result.affectedRows` reports the number of rows inserted by an INSERT command.
- ☐ True
- ☐ False
- 3) `result.affectedRows` reports the number of rows selected by a SELECT command.
- ☐ True
- ☐ False

Exploring further:

- [MySQL Documentation](#)
- [MySQL error codes and messages](#)
- [node-mysql documentation \(GitHub\)](#)
- [SQL injection from W3Schools](#)

11.9 MongoDB

MongoDB document database

Node.js web applications may use relational or non-relational (NoSQL) databases to store web application data. **MongoDB** is the most popular NoSQL database used by Node.js developers. MongoDB stores data objects as documents inside a collection.

- A **document** is a single data object in a MongoDB database that is composed of field/value pairs, similar to JSON property/value pairs.
- A **collection** is a group of related documents in a MongoDB database.

MongoDB stores documents internally as BSON documents. A **BSON document** (Binary JSON) is a binary representation of JSON with additional type information. BSON types include string, integer, double, date, boolean, null, and others. A BSON document may not exceed 16 MB in size.

PARTICIPATION ACTIVITY

11.9.1: Documents and collections.



document

```
{
  name: "Sue",
  gpa: 3.1,
  interests: ["biking", "reading"]
}
```

field

value

```
{
  name: "Sue",
  gpa: 3.1,
  interests: ["biking", "reading"],
  address: {
    city: "Dallas",
    state: "TX"
  }
}
```

} nested
document

collection

```
{
  name: "Sue",
  gpa: 3.1,
  interests: ["biking", "reading"]
}
```

```
{
  name: "Larry",
  gpa: 2.5,
  interests: ["RPGs", "chess"]
}
```

```
{
  name: "Anne",
  gpa: 4.0,
  interests: ["coding", "Pilates"]
}
```

Animation content:

A document shows a single student:

```
{
  name: "Sue",
```

```
gpa: 3.1,  
interests: ["biking", "reading"]  
}
```

On the last line, interests is the field and ["biking", "reading"] is the value.

Another document with a nested document is shown:

```
{  
  name: "Sue",  
  gpa: 3.1,  
  interests: ["biking", "reading"],  
  address: {  
    city: "Dallas",  
    state: "TX"  
  }  
}
```

The address block is the nested document. Next a collection of a group of student documents is shown:

```
{  
  name: "Sue",  
  gpa: 3.1,  
  interests: ["biking", "reading"]  
}  
{  
  name: "Larry",  
  gpa: 2.5,  
  interests: ["RPGs", "chess"]  
}  
{  
  name: "Anne",  
  gpa: 4.0,  
  interests: ["coding", "Pilates"]  
}
```

Animation captions:

1. A single student is represented as a document with field:value pairs. The name field is assigned a BSON string, gpa is a double, and interests is an array.

2. Documents may be nested. The student document contains a nested address document.
3. MongoDB organizes documents into collections. A group of students is stored in a single collection.

**PARTICIPATION
ACTIVITY**

11.9.2: MongoDB concepts.



1) Node.js only works with non-relational databases.



- ☐ True
- ☐ False

2) All MongoDB documents must be stored in a collection.



- ☐ True
- ☐ False

3) MongoDB documents may store nested documents.



- ☐ True
- ☐ False

4) Document fields do not require quotes, but all values do.



- ☐ True
- ☐ False

5) MongoDB stores documents in a binary-encoded format.



- ☐ True
- ☐ False

6) No size limit exists for a BSON document.

- ☐ True
- ☐ False

Installing MongoDB

MongoDB runs on a wide range of operating systems. Instructions for installing MongoDB Community Edition are provided on the [MongoDB website](#).

MongoDB Shell is a program for interacting with MongoDB. Instructions for installing MongoDB Shell are also available on the [MongoDB website](#).

MongoDB Shell

MongoDB Shell is a command-line interface for creating and deleting documents, querying, creating user accounts, and performing many other operations in MongoDB. The `mongosh` command starts MongoDB Shell.

PARTICIPATION ACTIVITY

11.9.3: Running MongoDB Shell.

```
$ mongosh
Current Mongosh Log ID: XYZ
Connecting to: mongodb://127.0.0.1:27017/...

test> use mydb
switched to db mydb

mydb> stu = { _id: 123, name: "Sue", gpa: 3.1 }
{ "_id" : 123, "name" : "Sue", "gpa" : 3.1 }

mydb> db.students.insertOne(stu)
{ acknowledged: true, insertedId: 123 }

mydb> db.students.find()
[ { "_id" : 123, "name" : "Sue", "gpa" : 3.1 } ]
```

MongoDB

mydb

students

```
{ _id: 123,
  name: "Sue",
  gpa: 3.1 }
```


Animation content:

The following console is shown:

```
$ mongosh
Current Mongosh Log ID: XYZ
Connecting to: mongodb://127.0.0.1:27017/...
test> use mydb
switched to db mydb
```

```
mydb> stu = { _id: 123, name: "Sue", gpa: 3.1 }
{ "_id" : 123, "name" : "Sue", "gpa" : 3.1 }
```

```
mydb> db.students.insertOne(stu)
{ acknowledged: true, insertedId: 123 }
```

```
mydb> db.students.find()
[ { "_id" : 123, "name" : "Sue", "gpa" : 3.1 } ]
```

The MongoDB shell is displayed next which holds the mydb database. mydb database contains the student collection with the following student information:

```
{ _id: 123,
  name: "Sue",
  gpa: 3.1 }
```

Animation captions:

1. Entering "mongosh" at the command line starts the MongoDB Shell and connects to the MongoDB instance running on the local computer.
2. The use command creates a new database called "mydb" since mydb does not exist.
3. stu is assigned a document and inserted into the students collection with insertOne().
4. The find() method retrieves the one student in the students collection.



Refer to the commands entered into the MongoDB Shell below.

```
test> use dealer
switched to db dealer

dealer> car = { _id: 200, make: "Ford", model: "Mustang" }
{ "_id" : 200, "make" : "Ford", "model" : "Mustang" }

dealer> db.autos.insertOne(car)
{ acknowledged: true, insertedId: 200 }

dealer> db.autos.find()
[ { "_id" : 200, "make" : "Ford", "model" : "Mustang" } ]

dealer> show dbs
admin      41 kB
config    111 kB
dealer    73.7 kB
local      41 kB
mydb      73.7 kB

dealer> help
Shell Help:

    use                Set current database
    show                'show databases'/'show dbs': Print a list of all available
databases.
                        'show collections'/'show tables': Print a list of all
collections for...
    ...

dealer> show collections
autos

dealer> db.autos.help()
Collection Class:

    aggregate          Calculates aggregate values for the data in a collection or a
view.
    bulkWrite           Performs multiple write operations with controls for order of
execution.
    count              Returns the count of documents that would match a find() query
for...
    ...

dealer> exit
```

If unable to drag and drop, refresh the page.

db.autos.insertOne(car)

show dbs

show collections

exit

use dealer

db.autos.help()

help

Displays a summary of all available shell commands.

Displays a summary of all collection methods.

Creates or selects a database called "dealer".

Quits the MongoDB Shell.

Displays a list of all databases.

Displays all collections in the current database.

Inserts a single document into the "autos" collection.

Reset

Inserting documents

The ***insertOne()*** collection method inserts a single document into a collection. The ***insertMany()*** collection method inserts multiple documents into a collection.

In the figure below, Sue is inserted into the students collection, then three students in the `students` array are inserted.

Figure 11.9.1: Inserting multiple students in bulk.

```
mydb> db.students.insertOne({ name: "Sue", gpa: 3.1 })
{
  acknowledged: true,
  insertedId: ObjectId("62794229fc4ebd4933877a9d")
}

mydb> students = [
... { name: "Maria", gpa: 4.0 },
... { name: "Xiu", gpa: 3.8 },
... { name: "Braden", gpa: 2.5 } ]

mydb> db.students.insertMany(students)
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("627942f6fc4ebd4933877a9e"),
    '1': ObjectId("627942f6fc4ebd4933877a9f"),
    '2': ObjectId("627942f6fc4ebd4933877aa0")
  }
}

mydb> db.students.find()
[
  { _id: ObjectId("62794229fc4ebd4933877a9d"), name: 'Sue', gpa: 3.1 },
  { _id: ObjectId("627942f6fc4ebd4933877a9e"), name: 'Maria', gpa: 4 },
  { _id: ObjectId("627942f6fc4ebd4933877a9f"), name: 'Xiu', gpa: 3.8 },
  { _id: ObjectId("627942f6fc4ebd4933877aa0"), name: 'Braden', gpa: 2.5 }
]
```

The ***_id*** field is automatically assigned to every document and is always the first field in the document. The ***_id*** acts as a primary key. A **primary key** is a field that uniquely identifies each document in a collection.

The ***_id*** may be assigned a unique value like a student ID number or use an auto-incrementing value. In the figure above, no ***_id*** field was assigned, so MongoDB automatically assigned an **ObjectId** to ***_id***. An **ObjectId** is a 12-byte BSON type that contains a unique value. An **ObjectId** is displayed as a hexadecimal number. Ex: 62794229fc4ebd4933877a9d.



- 1) The command below assigns `_id` an ObjectId.



```
db.students.insertOne({ _id:
123, "Ebony", gpa: 3.2 })
```

- ☐ True
- ☐ False

- 2) In the figure above, the inserted documents received similar ObjectIds.



- ☐ True
- ☐ False

- 3) The ObjectId is 12 characters long.



- ☐ True
- ☐ False

- 4) The `_id` field may use duplicate values.



- ☐ True
- ☐ False

Finding documents

The ***find()*** collection method returns all documents by default or documents that match an optional query parameter. The ***findOne()*** collection method returns only the first document matching the query. Both methods return null if the query matches no documents.

Figure 11.9.2: Find 'Sue' and students with GPA ≥ 3.0 .

```
mydb> db.students.find({ name: 'Sue' })
[
  { _id: ObjectId("62794229fc4ebd4933877a9d"), name: 'Sue', gpa:
3.1 }
]

mydb> db.students.find({ gpa: { $gte: 3.0 } })
[
  { _id: ObjectId("62794229fc4ebd4933877a9d"), name: 'Sue', gpa:
3.1 },
  { _id: ObjectId("627942f6fc4ebd4933877a9e"), name: 'Maria', gpa:
4 },
  { _id: ObjectId("627942f6fc4ebd4933877a9f"), name: 'Xiu', gpa:
3.8 }
]
```

The operators used in queries are summarized in the table below, but many more exist and are documented in the MongoDB manual.

Table 11.9.1: Common MongoDB query operators.

Operator	Description	Example
<code>field:value</code>	Matches documents with fields that are equal to the given value.	<pre>// Matches student with this _id { "_id" : ObjectId("62794229fc4ebd4933877a9d") }</pre>
<code>\$eq</code> <code>\$ne</code>	Matches values = or \neq to the given value.	<pre>// Matches all docs except Sue { name: { \$ne: "Sue" } }</pre>
<code>\$gt</code> <code>\$gte</code>	Matches values $>$ or \geq to the given value.	<pre>// Matches students with gpa > 3.5 { gpa: { \$gt: 3.5 } }</pre>
<code>\$lt</code> <code>\$lte</code>	Matches values $<$ or \leq to the given value.	<pre>// Matches students with gpa <= 3.0 { gpa: { \$lte: 3.0 } }</pre>
<code>\$in</code> <code>\$nin</code>	Matches values in or not in a given array.	<pre>// Matches Sue, Susan, or Susie { name: { \$in: ["Sue", "Susan", "Susie"] } }</pre>
<code>\$and</code>	Joins query clauses with a logical AND, returns documents that match both clauses.	<pre>// Matches student with gpa >= 3.0 // and gpa <= 3.5 { \$and: [{ gpa: { \$gte: 3.0 } }, { gpa: { \$lte: 3.5 } }] }</pre>
<code>\$or</code>	Joins query clauses with a logical OR, returns documents that match either clauses.	<pre>// Matches students with gpa >= 3.9 // or gpa <= 3.0 { \$or: [{ gpa: { \$gte: 3.9 } }, { gpa: { \$lte: 3.0 } }] }</pre>



Refer to the "autos" collection below, and choose the documents returned by each query.

```
[
  {
    "_id" : 100,
    "make" : "Ford",
    "model" : "Fusion",
    "year" : 2014,
    "options" : [ "engine start", "moon roof" ],
    "price" : 13500
  },
  {
    "_id" : 200,
    "make" : "Honda",
    "model" : "Accord",
    "year" : 2013,
    "options" : [ "spoiler", "alloy wheels", "sunroof" ],
    "price" : 16900
  },
  {
    "_id" : 300,
    "make" : "Dodge",
    "model" : "Avenger",
    "year" : 2012,
    "options" : [ "leather seats" ],
    "price" : 10800
  },
  {
    "_id" : 400,
    "make" : "Toyota",
    "model" : "Corolla",
    "year" : 2013,
    "options" : [ "antitheft" ],
    "price" : 13400
  }
]
```

1) `db.autos.find({})`

- ☐ `_id 100`
- ☐ All documents
- ☐ null



- 2) `db.autos.find({ year: { $gte: 2013 } })`
- ☐ `_id 100`
 - ☐ `_id 100, 200, 400`
 - ☐ `_id 300`
- 3) `db.autos.findOne({ year: { $gte: 2013 } })`
- ☐ `_id 100`
 - ☐ `_id 100, 200, 400`
 - ☐ `_id 300`
- 4) `db.autos.findOne({ year: { $gte: 2016 } })`
- ☐ `_id 100`
 - ☐ Run-time error
 - ☐ null
- 5) `db.autos.find({ $and: [{price: { $lte: 15000 } }, { options: { $in: ["sunroof", "antitheft", "moonroof"] } }] })`
- ☐ `_id 100`
 - ☐ `_id 100, 400`
 - ☐ `_id 100, 200, 400`
- 6) `db.autos.find({ $or: [{ "make": "Honda" }, { year: { $ne: 2013 } }] })`
- ☐ All documents
 - ☐ `_id 100, 300`
 - ☐ `_id 100, 200, 300`

Updating documents

The **updateOne()** collection method modifies a single document in a collection. The **updateMany()** collection method modifies multiple documents in a collection. The methods have two required parameters:

1. **query** - The query to find the document(s) to update. An empty query `{ }` matches all documents.
2. **update** - The modification to perform on matched documents using an update operator like `$inc`, `$set`, and `$unset`.

In the example below, the calls to `updateOne()` and `updateMany()` return the `matchedCount` property indicating how many documents matched the query, and the `modifiedCount` property indicating the number of documents modified.

Figure 11.9.3: Change Sue's GPA to 3.3, and set all students with GPA > 3 to 1.

```
mydb> db.students.updateOne({ name: 'Sue' }, { $set: { gpa: 3.3 } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}

mydb> db.students.find({ name: 'Sue' })
{ "_id" : ObjectId("5e600d18bbd10ee972f6ed9a"), "name" : "Sue", "gpa" : 3.3 }

mydb> db.students.updateMany({ gpa: { $gt: 3 } }, { $set: { gpa: 1 } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 3,
  modifiedCount: 3,
  upsertedCount: 0
}

mydb> db.students.find()
[
  { _id: ObjectId("62794229fc4ebd4933877a9d"), name: 'Sue', gpa: 1 },
  { _id: ObjectId("627942f6fc4ebd4933877a9e"), name: 'Maria', gpa: 1 },
  { _id: ObjectId("627942f6fc4ebd4933877a9f"), name: 'Xiu', gpa: 1 },
  { _id: ObjectId("627942f6fc4ebd4933877aa0"), name: 'Braden', gpa: 2.5 }
]
```

Table 11.9.2: Common MongoDB update operators.

Operator	Description	Example
\$currentDate	Sets a field's value to the current date/time	<pre>// Sue's "birthDate" : ISODate("2022-05- 09T16:39:00.121Z") db.students.updateOne({ name: 'Sue' }, { \$currentDate: { birthDate: true } })</pre>
\$inc	Increments a field's value by the specified amount	<pre>// Sue's "gpa" incremented from 3.1 to 3.2 db.students.updateOne({ name: 'Sue' }, { \$inc: { gpa: 0.1 } })</pre>
\$rename	Renames a field	<pre>// Sue's "name" is now "firstName" db.students.updateOne({ name: 'Sue' }, { \$rename: { name: 'firstName' } })</pre>
\$set	Sets a field's value	<pre>// Sue's "gpa" : 4.0 db.students.updateOne({ name: 'Sue' }, { \$set: { gpa: 4.0 } })</pre>
\$unset	Removes a field	<pre>// Removes Sue's "gpa" and "birthDate" fields db.students.updateOne({ name: 'Sue' }, { \$unset: { gpa: "", birthDate: "" } })</pre>

PARTICIPATION ACTIVITY

11.9.7: Updating documents in the 'autos' collection.



Refer to the "autos" collection below, and choose the result of each command.

```
[
  {
    "_id" : 100,
    "make" : "Ford",
    "model" : "Fusion",
    "year" : 2014,
    "options" : [ "engine start", "moon roof" ],
    "price" : 13500
  },
  {
    "_id" : 200,
    "make" : "Honda",
    "model" : "Accord",
    "year" : 2013,
    "options" : [ "spoiler", "alloy wheels", "sunroof" ],
    "price" : 16900
  }
]
```

1) `db.autos.updateOne({ price: { $gt: 10000 } }, { $set: { year: 2000, options: [] } })`

- Only auto with _id 100 has year
- ☐ set to 2000 and options removed.
 - ☐ Both autos have year set to 2000 and options removed.
 - ☐ No autos are updated.

2) `db.autos.updateMany({ price: { $gt: 10000 } }, { $set: { year: 2000, options: [] } })`

- Only auto with _id 100 has year
- ☐ set to 2000 and options removed.
 - ☐ Both autos have year set to 2000 and options removed.
 - ☐ No autos are updated.

3) `db.autos.updateOne({ price: { $gt: 10000 } }, { $set: { sold: true } })`

- No autos are updated because
- ☐ the autos do not have a "sold" field.
 - ☐ Auto with `_id` 100 has new field "sold" set to true.
 - ☐ Both autos have a new field "sold" set to true.

4) `db.autos.updateOne({ _id: 100 }, { $currentDate: { soldDate: true } })`

- ☐ Auto with `_id` 100 has new field "soldDate" set to true.
- ☐ Auto with `_id` 100 has new field "soldDate" set to the Unix epoch (January 1, 1970).
- ☐ Auto with `_id` 100 has new field "soldDate" set to the current date and time.

5) `db.autos.updateOne({ _id: 100 }, { $inc: { price: -500, year: 2 } })`

- ☐ Auto with `_id` 100 has price reduced by 500 and year increased by 2.
- ☐ Auto with `_id` 100 has price set to -500 and year set to 2.
- ☐ Auto with `_id` 100 has price and year fields removed.

Deleting documents

The **`deleteOne()`** collection method deletes a single document from a collection. The **`deleteMany()`** collection method deletes multiple documents from a collection. The methods have a required query parameter that matches documents to delete.

In the example below, the calls to **`deleteOne()`** and **`deleteMany()`** return a **`deletedCount`** property indicating how many documents were deleted.

Figure 11.9.4: Delete the first student with GPA < 3.5 (Sue), and delete all students with GPA > 3.5 (Maria and Xiu).

```
mydb> db.students.deleteOne({ gpa: { $lt:3.5 } })
{ acknowledged: true, deletedCount: 1 }

mydb> db.students.find()
[
  { _id: ObjectId("627942f6fc4ebd4933877a9e"), name: 'Maria', gpa: 4 },
  { _id: ObjectId("627942f6fc4ebd4933877a9f"), name: 'Xiu', gpa: 3.8 },
  { _id: ObjectId("627942f6fc4ebd4933877aa0"), name: 'Braden', gpa: 2.5 }
]

mydb> db.students.deleteMany({ gpa: { $gt:3.5 } })
{ acknowledged: true, deletedCount: 2 }

mydb> db.students.find()
[
  { _id: ObjectId("627942f6fc4ebd4933877aa0"), name: 'Braden', gpa: 2.5 }
]
```

PARTICIPATION ACTIVITY

11.9.8: Deleting documents.

- 1) Delete all documents from the "autos" collection.

`db.autos.`

Check

Show answer

- 2) Delete all documents with a price more than \$10,000.



db.autos.

Check

Show answer

- 3) Delete only the first document with the year 2020.



db.autos.

Check

Show answer

Exploring further:

- [MongoDB manual](#)
- [Installing MongoDB](#)
- [Query and Projection Operators](#)
- [SQL to MongoDB Mapping Chart](#)

11.10 Mongoose

Schemas and models

A Node.js application using MongoDB may use the ***mongodb module*** to interact with a MongoDB database using many of the same functions supported in the mongo shell. However, many developers prefer to use Mongoose, which simplifies some MongoDB operations. The ***mongoose module*** provides object data mapping (ODM) and structured schemas to MongoDB collections. ***Object data mapping (ODM)*** is the conversion of data from a database into a JavaScript object. Mongoose is installed using: `npm install mongoose`.

Mongoose uses schemas and models to interact with MongoDB databases. A **schema** defines the structure of documents within a MongoDB collection. The **mongoose.Schema()** method creates a new **Schema** object that defines the properties and data types of a document. The supported data types are:

- **String** - "string"
- **Number** - Integers or floating-point numbers
- **Date** - Date and time
- **Buffer** - Binary data
- **Boolean** - `true` or `false`
- **Mixed** - Any kind of data
- **ObjectId** - MongoDB `ObjectId`
- **Array** - Array of any data type. Ex: `[Number]`.

Figure 11.10.1: Mongoose schema for a student.

```
const mongoose = require("mongoose");

const studentSchema = new mongoose.Schema({
  name:      String,
  gpa:       { type: Number, min: 0, max: 4 },
  birthDate: { type: Date, default: Date.now },
  interests: [ String ]
});
```

A **model** is a constructor compiled from a schema. A model instance represents a MongoDB document that can be saved to or retrieved from a MongoDB database. Models are created from schemas using the **mongoose.model()** method, which has two parameters:

1. **modelName** - Singular name of the model's collection. Ex: "Card" for the collection "cards". Mongoose automatically uses collections that are the plural of the model name.
2. **schema** - Previously defined schema.

Good practice is to use model names and model variables that have the first letter capitalized. Ex: "Card" instead of "card".

Figure 11.10.2: Mongoose model for a 'students' collection.

```
// Create a "Student" model from the studentSchema
schema
const Student = mongoose.model("Student",
studentSchema);

// Create a student document from the Student model
const stu = new Student({
  name: "Sue Black",
  gpa: 3.1,
  birthDate: new Date(1999, 11, 2),
  interests: ["biking", "reading"]
});
```

**PARTICIPATION
ACTIVITY**

11.10.1: Mongoose concepts.

1) Mongoose is an essential package for any Node.js application that needs to interface with MongoDB.

- ☐ True
☐ False

2) Mongoose forces all documents within a collection to match a predefined schema.

- ☐ True
☐ False

3) The second argument to the `mongoose.model()` method may be a schema or a map of properties and data types.

- ☐ True
☐ False

4) The call `mongoose.model("Dog", dogSchema)` creates a constructor for creating objects for a collection called "Dog".

- ☐ True
- ☐ False

5) The `studentSchema` defined in the figure above requires the `gpa` field to be assigned only values between 0 and 4.

- ☐ True
- ☐ False

Connecting to MongoDB

Mongoose must establish a connection with MongoDB before executing any database operations. The **`mongoose.connect()`** method takes a URL parameter indicating the machine on which MongoDB is running and the name of the database to use. If the specified database does not exist, MongoDB creates the database.

Figure 11.10.3: Connecting to the 'mydb' database on MongoDB.

```
// Connect to MongoDB running on 127.0.0.1
mongoose.connect("mongodb://localhost/mydb");
```

Saving documents

Mongoose provides a collection of methods for implementing CRUD operations. Two Mongoose methods can be used to implement the Create operation:

- The **`Document.save()`** method saves the document to a MongoDB database. The given callback function is executed after the save operation has completed.
- The **`Model.create()`** method saves a single document or array of documents to a MongoDB database. The given callback function is executed after the save operation has completed.

**PARTICIPATION
ACTIVITY**

11.10.2: Mongoose saving a student to MongoDB.



```
const express = require("express");
const mongoose = require("mongoose");

mongoose.connect("mongodb://localhost/mydb");

const studentSchema = new mongoose.Schema({
  name: { type: String, required: true },
  gpa: { type: Number, min: 0, max: 4 },
  birthDate: { type: Date, default: Date.now },
  interests: [ String ]
});

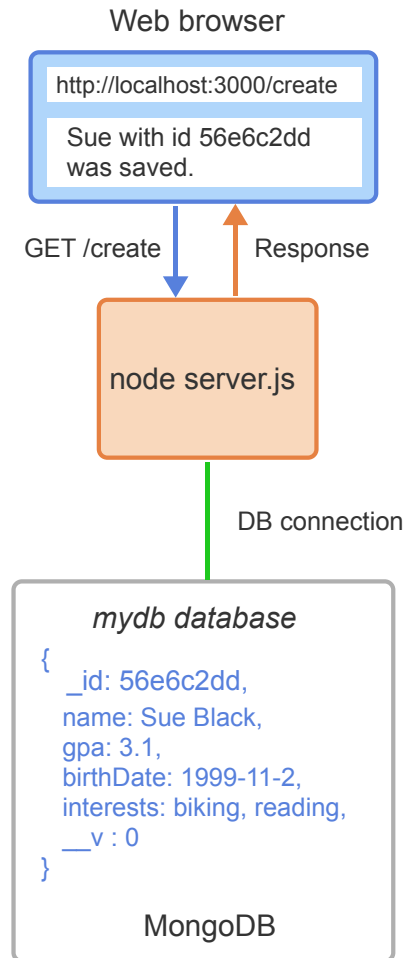
const Student = mongoose.model("Student", studentSchema);

const app = express();

app.get("/create", function(req, res) {
  const stu = new Student({
    name: "Sue Black",
    gpa: 3.1,
    birthDate: "1999-11-02",
    interests: ["biking", "reading"]
  });

  stu.save(function(err, stu) {
    res.send("Sue with id " + stu._id +
      " was saved.");
  });
});

app.listen(3000);
```


Animation content:

A block of code is displayed:

```
const express = require("express");
const mongoose = require("mongoose");
mongoose.set("useUnifiedTopology", true);
mongoose.connect("mongodb://localhost/mydb");
const studentSchema = new mongoose.Schema({
  name: { type: String, required: true },
  gpa: { type: Number, min: 0, max: 4 },
  birthDate: { type: Date, default: Date.now },
```

```
    interests: [ String ]
  });
const Student = mongoose.model("Student", studentSchema);
const app = express();
app.get("/create", function(req, res) {
  const stu = new Student({
    name: "Sue Black",
    gpa: 3.1,
    birthDate: "1999-11-02",
    interests: ["biking", "reading"]
  });
  stu.save(function(err, stu) {
    res.send("Sue with id " + stu._id +
      " was saved.");
  });
});
app.listen(3000);
```

A connection between the web server called node server.js and the MongoDB database called mydb database is established. A web browser with the URL <http://localhost:3000/create> sends the following GET request to the web server:

GET/create

The GET request creates a student in mydb database with the following information:

```
{
  name: Sue Black,
  gpa: 3.1,
  birthDate: 1999-11-2,
  interests: biking, reading,
  __v : 0
}
```

The database sends back this information and the following is displayed on the web browser:
Sue with id 56e6c2dd was saved.

Animation captions:

1. The express and mongoose modules are loaded.
2. Mongoose connects to the "mydb" database on MongoDB running on the local machine.
3. The student schema and model are created.
4. Express app defines the "/create" route and starts listening on port 3000.
5. When the browser requests the "/create" route, the server creates a new Student object

from the Student model.

6. `stu.save()` saves the student object as a document in the mydb database where a new ObjectId is assigned to `_id`.
7. The Express server sends a response to the browser with Sue's ID.

PARTICIPATION ACTIVITY

11.10.3: Saving documents to MongoDB using Mongoose.

Refer to the animation above.

- 1) What URL does `mongoose.connect()` use to connect to the MongoDB database called "students" instead of "mydb"?
 - ☐ `mongodb://localhost/mydb`
 - ☐ `http://localhost/students`
 - ☐ `mongodb://localhost/students`
- 2) What method could be used in place of `stu.save()` in the example above?
 - ☐ `Student.create()`
 - ☐ `Student.updateOne()`
 - ☐ `Student.deleteOne()`
- 3) When is the new student's `_id` created in the animation above?
 - ☐ When assigning `stu = new Student(...)`.
 - ☐ When `stu.save()` is called.
 - ☐ When the `studentSchema` is created.

4) What does the `__v` field in the student document represent?

- ☐ versionKey
- ☐ document's ID
- ☐ Mongoose version

5) According to the student schema, what `birthDate` is assigned to the student below when the student is saved to MongoDB?

```
const stu = new Student({  
  name: "Jack White",  
  gpa: 2.5,  
  interests: ["football",  
    "piano"]  
});
```

- ☐ null
- ☐ The current date and time.
- ☐ No birthDate is assigned
- ☐ because an exception is thrown.

- 6) According to the student schema, what happens when the student below is saved to MongoDB?



```
const stu = new Student({  
  gpa: 2.5,  
  interests: ["football",  
    "piano"]  
});
```

- ☐ A student with no name field is saved.
- ☐ A student with an empty string assigned to the name field is saved.
- ☐ An error occurs, and no student is saved.

Finding documents

The Mongoose methods for retrieving documents work similar to MongoDB's `find()` and `findOne()` methods. The Mongoose find methods support MongoDB query operators like `$in` and `$gt`.

Table 11.10.1: Common Mongoose find methods.

Method	Description	Example
<code>Model.find(conditions, [callback])</code>	Finds all documents that match the conditions.	<pre>// Find all students with Student.find({ gpa: { \$gt function(err, students console.log(student } });</pre>
<code>Model.findOne(conditions, [callback])</code>	Finds the first document that match the conditions.	<pre>// Find first student with in the name Student.findOne({ name: r "i" }) }, function(err, stu) { if (stu === null) { console.log("No } else { console.log(stu) } });</pre>
<code>Model.findById(id, [callback])</code>	Finds the document that matches the id.	<pre>// Find the student with Student.findById("56e0ad2 function(err, stu) { if (stu === null) { console.log("No } else { console.log(stu) } });</pre>

All Mongoose find methods return a **Query** object. The **Query** object has a number of methods that may be chained to create a more readable query. Example methods include:

- The **Query.where()** method specifies a path to filter documents.
- The **Query.sort()** method sets the sort order.
- The **Query.limit()** method limits the number of documents returned.
- The **Query.exec()** method is called last to execute the query and provides a callback function that is called after the query executes.

PARTICIPATION
ACTIVITY

11.10.4: Finding students by chaining Query methods.



```
Student.find({ gpa: { $gte: 3 }})
  .where("interests").in(["fishing", "biking"])
  .sort({ "birthDate": "desc" })
  .limit(2)
  .exec(function(err, students) {
    students.forEach(function(stu) {
      console.log(stu.name);
    });
  });
```

Dana
Sue

```
{ _id: 9ed, name: Dana,
  gpa: 3.3, birthDate: 2000-1-15,
  interests: gardening, biking }
{ _id: 5ab, name: Sue,
  gpa: 3.1, birthDate: 1999-11-2,
  interests: biking, reading }
{ _id: 3ff, name: Greg,
  gpa: 3.2, birthDate: 1998-6-30,
  interests: fishing, tennis }
{ _id: 7ac, name: Cindy,
  gpa: 4.0, birthDate: 2001-2-6,
  interests: rock climbing }
{ _id: 3ca, name: Laura,
  gpa: 2.6, birthDate: 2002-5-6,
  interests: dancing, coding }
```

Animation content:

A block of Node.js is shown:

```
Student.find({ gpa: { $gte: 3 }})
  .where("interests").in(["fishing", "biking"])
  .sort({ "birthDate": "desc" })
  .limit(2)
  .exec(function(err, students) {
    students.forEach(function(stu) {
      console.log(stu.name);
    });
  });
```

The collection of students is shown:

```
{ _id: 9ed, name: Dana,
  gpa: 3.3, birthDate: 2000-1-15,
  interests: gardening, biking }
{ _id: 5ab, name: Sue,
  gpa: 3.1, birthDate: 1999-11-2,
  interests: biking, reading }
{ _id: 3ff, name: Greg,
```

```
gpa: 3.2, birthDate: 1998-6-30,  
interests: fishing, tennis }  
{ _id: 7ac, name: Cindy,  
  gpa: 4.0, birthDate: 2001-2-6,  
  interests: rock climbing }  
{ _id: 3ca, name: Laura,  
  gpa: 2.6, birthDate: 2002-5-6,  
  interests: dancing, coding }
```

Sue, Dana, Cindy, and Greg have a GPA ≥ 3.0 , so they are selected. Of those with a GPA ≥ 3.0 , Dana, Sue, and Greg have interests in fishing or biking, so they are selected. When sorted by birth date and limited to 2 students, only Dana and Sue are selected and output to the console.

Animation captions:

1. The query first finds all students with $\text{gpa} \geq 3$.
2. `.where()` selects students that have an interest of fishing or biking.
3. `.sort()` sorts the matching documents by `birthDate` in descending order.
4. `.limit()` limits the number of results to the first 2 documents.
5. `.exec()` executes the query, and returns the students' names.

PARTICIPATION ACTIVITY

11.10.5: Mongoose find methods.



Given the model and schema below and the collection of super heroes, match each query with the query's results.

```
const SuperHero = mongoose.model("SuperHero", {
  _id:      Number,
  universe: { type: String, enum: [ "Marvel", "DC", "other" ], default:
"other" },
  name:     { type: String, required: true },
  stamina:  { type: Number, min: 1, max: 10 },
  powers:   [ String ],
  origin:   {
    comic: String,
    year:  Number
  }
});

const heroes = [
  {
    _id: 111,
    universe: "DC",
    name: "Wonder Woman",
    stamina: 9,
    powers: ["strength", "combat"],
    origin: { comic: "All Star Comics #8", year: 1941 }
  },
  {
    _id: 222,
    universe: "DC",
    name: "Green Lantern",
    stamina: 5,
    powers: ["flying", "power ring"],
    origin: { comic: "All-American Comics #16", year: 1940 }
  },
  {
    _id: 333,
    universe: "Marvel",
    name: "Spider-Man",
    stamina: 6,
    powers: ["strength", "agility", "spider-sense"],
    origin: { comic: "Amazing Fantasy #15", year: 1962 }
  },
  {
    _id: 444,
    universe: "Marvel",
    name: "Storm",
    stamina: 4,
    powers: ["control weather"],
    origin: { comic: "Giant-Size X-Men #1", year: 1976 }
  }
];

SuperHero.create(heroes, function(err, heroes) {
  console.log("Created");
});
```

If unable to drag and drop, refresh the page.

IDs 222 and 333

ID 111

IDs 111 and 333

IDs 222 and 444

2

```
SuperHero.find({ stamina: { $gte: 6 } },  
function(err, heroes) { ... })
```

```
SuperHero.findOne(  
  { powers: {$in:["strength", "shrinking"  
function(err, hero) { ... })
```

```
SuperHero.find({})  
  .where("powers").in(["strength", "flyin  
  .sort({ "stamina": "asc" })  
  .limit(2)  
  .exec(function(err, heroes) { ... });
```

```
SuperHero.find({})  
  .or([ { "origin.year": { $gt: 1970 } }  
        { name: "Green Lantern" } ])  
  .exec(function(err, heroes) { ... })
```

```
SuperHero.where({ stamina: { $lte: 5 } })  
  .count(function(err, count) { ... })
```

Reset

Finding with await operator

Mongoose find methods can be called with a callback function or from an **async** function using the **await** operator. Both functions in the figure below find all students with a GPA > 3 and output the students to the console. The **findStudentsCallback()** function uses a callback function, but **findStudentsAsync()** awaits the results returned by **find()** and **sort()**.

Figure 11.10.4: Two ways to find documents.

```
function findStudentsCallback() {
  Student.find({ gpa: { $gte: 3 }})
    .sort({ "gpa": "desc" })
    .exec(function(err, students) {
      for (let stu of students) {
        console.log(stu.name + " " + stu.gpa);
      }
    });
}

async function findStudentsAsync() {
  const students = await Student.find({ gpa: { $gte: 3 }}).sort({ "gpa":
"desc" });
  for (let stu of students) {
    console.log(stu.name + " " + stu.gpa);
  }
}
```

**PARTICIPATION
ACTIVITY**

11.10.6: Asynchronous finding.

- 1) Calling find methods with `await` generally results in simpler code.

☐ True

☐ False
- 2) Calling find methods with `await` generally results in faster queries.

☐ True

☐ False

- 3) The code below is guaranteed to show the results from query 1 before showing the results from query 2.



```
// Query 1
Student.find({ gpa: { $gte: 3
}})
  .exec(function(err,
students) {
    console.log(students);
  });

// Query 2
Student.find({ gpa: { $lte: 3
}})
  .exec(function(err,
students) {
    console.log(students);
  });
```

- ☐ True
- ☐ False

- 4) The code below is guaranteed to show the results from query 1 before showing the results from query 2.



```
// Query 1
let students = await
Student.find({ gpa: { $gte: 3
}})
  .sort({ "gpa": "desc" });
console.log(students);

// Query 2
students = await
Student.find({ gpa: { $lte: 3
}})
  .sort({ "gpa": "desc" });
console.log(students);
```

- ☐ True
- ☐ False

Updating and deleting documents

The model's `save()` method can be used to update an existing document.

Figure 11.10.5: Changing a student's GPA with save().

```
// Find the student with this ID
Student.findById("56e711a9e1b0f9080f7a5621", function(err,
stu) {
  if (stu === null) {
    console.log("Student not found");
  }
  else {
    // Increase student's GPA and save
    stu.gpa += 0.1;
    stu.save(function (err, stu) {
      console.log("New GPA: " + stu.gpa);
    });
  }
});
```

Mongoose also provides methods to modify existing documents in a collection and to remove documents.

Table 11.10.2: Common Mongoose update methods.

Method	Description	Example
Model.updateOne(conditions, update, [options], [callback])	Updates the first document that matches the conditions.	<pre>// Change Sue's birthDate to 1995-12-02 Student.updateOne({ name: "Sue Black" }, { birthDate: new Date(1995, 11, 2) }, function(err, result) { console.log("Docs updated: " + result.modifiedCount); });</pre>
	Updates all	

<code>Model.updateMany(conditions, update, [options], [callback])</code>	documents that match the conditions.	<pre>// Add 0.5 to all GPAs <= 3.5 Student.updateMany({ gpa: { \$lte: 3.5 } }, { \$inc: { gpa: 0.5 } }, function(err, result) { console.log("Docs updated: " + result.modifiedCount); });</pre>
<code>Query.updateOne([conditions], [update], [options], [callback])</code>	Updates the first document that matches the optional conditions.	<pre>// Change this student's GPA to 3.9 Student.where({ _id: "56e711a9e1b0f9080f7a5621" }) .updateOne({ \$set: { gpa: 3.9 } }) .exec(function(err, result) { console.log("Docs updated: " + result.modifiedCount); });</pre>
<code>Query.updateMany([conditions], [update], [options], [callback])</code>	Updates all documents that match the optional conditions.	<pre>// Set all GPAs <= 3.5 to 2 Student.where({ gpa: { \$lte: 3.5 } }) .updateMany({ \$set: { gpa: 2 } }) .exec(function(err, result) { console.log("Docs updated: " + result.modifiedCount); });</pre>

Table 11.10.3: Common Mongoose delete methods.

Method	Description	Example

<code>Model.deleteOne(conditions, [callback])</code>	Deletes the first document that matches the conditions.	<pre>// Delete student w given id Student.deleteOne({ "56e711a9e1b0f9080f "}, function(err, re console.log(" " + result.deletedCount });</pre>
<code>Model.deleteMany(conditions, [callback])</code>	Deletes all documents that match the conditions.	<pre>// Delete all stude with GPA < 3 Student.deleteMany({ \$lt: 3 }}, function(err, re console.log(" " + result.deletedCount });</pre>
<code>Query.deleteOne([conditions], [callback])</code>	Deletes the first document that matches the optional conditions.	<pre>// Delete first stu with 2000-10-05 bir Student.find({ birt "2000-10-05" }) .deleteOne() .exec(function(e result) { console.log(" " + result.deletedCount });</pre>
<code>Query.deleteMany([conditions], [callback])</code>	Deletes all the documents that match the optional conditions.	<pre>// Delete all stude with GPA < 2 Student.find({ gpa: 2 }}) .deleteMany() .exec();</pre>

Like the Mongoose find methods, all update and delete methods may be executed asynchronously

with `await` instead of using callback functions or calling `exec()`.

Figure 11.10.6: Calling update and delete methods with `async`.

```
const updateResult = await Student.updateMany({ gpa: { $lte: 3.5 } }, {
  $inc: { gpa: 0.5 } });
console.log("Docs updated: " + updateResult.modifiedCount);

const deleteResult = await Student.deleteOne({ _id:
  "5f077d4616956f72105fb613" });
console.log("Docs deleted: " + deleteResult.deletedCount);
```

PARTICIPATION ACTIVITY

11.10.7: Updating and deleting documents with Mongoose.



Refer to the "superheroes" collection and `SuperHero` model.

```
[
  {
    _id: 111,
    universe: "DC",
    name: "Wonder Woman",
    stamina: 9,
    powers: ["strength", "combat"],
    origin: { comic: "All Star Comics #8", year: 1941 }
  },
  {
    _id: 222,
    universe: "DC",
    name: "Green Lantern",
    stamina: 5,
    powers: ["flying", "power ring"],
    origin: { comic: "All-American Comics #16", year: 1940 }
  }
]
```

- 1) What is the result of the following command?



```
SuperHero.updateMany({},  
  { $inc: { stamina: 1 } },  
  function(err) {} );
```

- ☐ Both heroes' stamina are set to 1.
- ☐ Wonder Woman's stamina increases by 1.
- ☐ Both heroes' stamina increases by 1.

- 2) What is the result of the following command?



```
SuperHero.updateOne({  
  "origin.year": { $gt : 1900 }  
},  
  { $set: { "origin.year":  
1900 } },  
  function(err, result) { ...  
});
```

- ☐ Both heroes' origin year are set to 1900.
- ☐ Wonder Woman's origin year is set to 1900.
- ☐ No heroes are updated.

- 3) Select the code segment equivalent to the following:



```
SuperHero.findById(222,
function(err, hero) {
  hero.powers.push("mind
control");
  hero.save(function(err,
hero) { ... });
});
```

- ☐ SuperHero.updateOne({
 _id: 222 },
 { \$set: {powers:
 ["mind control"]} });
- ☐ SuperHero.find({ _id:
 222 })
 .updateOne({ \$push:
 {powers: "mind control"}
 });
- ☐ SuperHero.find({ _id:
 222 })
 .updateOne({ \$push:
 {powers: "mind control"}
 })
 .exec();

- 4) Select the code segment equivalent to the following:



```
SuperHero.find({ _id: 111  
}).deleteOne().exec();
```

- ☐ SuperHero.deleteOne(111);
- ☐ SuperHero.deleteOne({
 _id: 111});
- ☐ await
 SuperHero.deleteOne({
 _id: 111 });

- 5) If three students have the same 2000-10-15 birthdate, what does the code output to the console?



```
const result = await
Student.deleteOne(
  { birthDate: "2000-10-05" });
console.log(result.deletedCount);
```

- ☐ 0
- ☐ 1
- ☐ 3

Project organization

Node.js code is often organized into separate modules to decrease the project's complexity and create re-usable code. Ex: Database connection code and Mongoose models are separated into separate modules with the models stored in a `models` directory.

Figure 11.10.7: Example Node.js project that adds new students entered in a form to a MongoDB database.

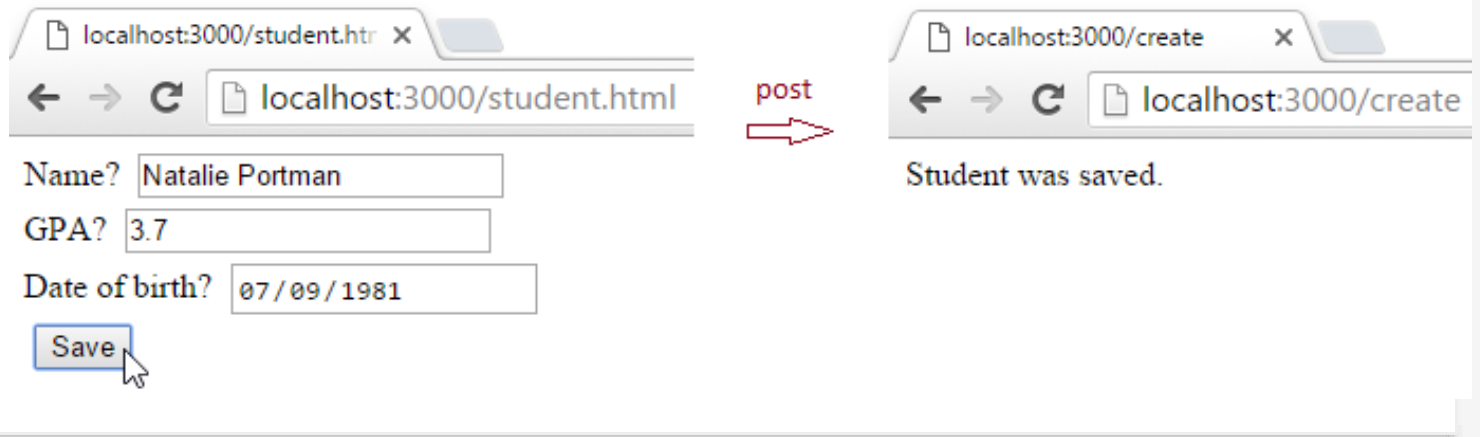
```
myproject
├── models
│   └── student.js
├── node_modules
│   ├── express
│   ├── mongoose
│   └── etc...
├── public
│   └── student.html
├── db.js
├── package.json
├── package-lock.json
└── server.js
```

```
// models/student.js
const db = require("../db");

const Student = db.model("Student", {
  name: String,
  gpa: { type: Number, min: 0, max: 4 },
  birthDate: { type: Date, default: Date.now },
  interests: [ String ]
});

module.exports = Student;
```

```
// db.js
const mongoose = require("mongoose");
mongoose.connect("mongodb://localhost/mydb");
module.exports = mongoose;
```

**PARTICIPATION
ACTIVITY**

11.10.8: Layout of Node.js project.

Refer to the figure above.

- 1) The `db` object in `student.js` is the `mongoose` object from `db.js`.
 - ☐ True
 - ☐ False
- 2) The `require` statement in `student.js` uses `"../db"` in the path because `db.js` is one directory above `student.js`.
 - ☐ True
 - ☐ False
- 3) Adding a new model called `Faculty` entails adding `Faculty` to `students.js`.
 - ☐ True
 - ☐ False
- 4) The `require("mongoose")` statement was mistakenly left out of `server.js`.
 - ☐ True
 - ☐ False

5) Changing the `method="post"` statement in `student.html` to `method="get"` requires making changes to `server.js`.

- ☐ True
- ☐ False

6) A 400 Bad Request is sent to the client if an error occurs saving the POSTed student to the database.

- ☐ True
- ☐ False

Exploring further:

- [Mongoose website](#)
- [Mongoose documentation](#)
- [MongoDB native driver for Node.js](#)

11.11 Creating RESTful web APIs (Node)

RESTful web APIs

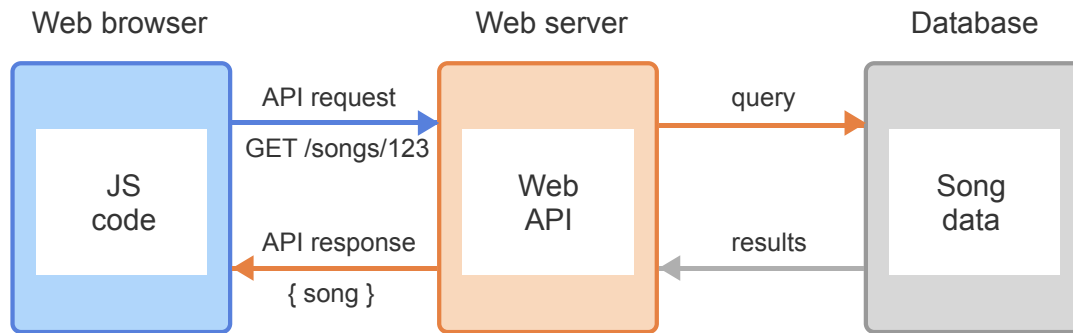
Modern web applications use web APIs to manipulate and transport data between the web browser and web server. A web API provides a standard interface to application data that may be used by other clients like mobile applications on phones and smartwatches. A **client** is a program that sends web API requests.

Web APIs typically are REST-based. **REST (representational state transfer)** is an architectural style that describes how various components interact in a distributed hypermedia system. A **RESTful web API** is a web API that implements CRUD (create, read/retrieve, update, delete) operations on resources using the mechanics of HTTP. A **resource** is a data entity whose representation is accessible via a URL. Ex: A representation for a "customer" resource may be accessible at

<http://paypal.com/customer>.

**PARTICIPATION
ACTIVITY**

11.11.1: Web application interacting with a web API to obtain song data.

**Animation content:**

A web browser using JavaScript code, a web server using a web API, and a database with song data are shown. The web browser sends an API request with GET /songs/123 to the web server. The web server requests the song from the database with a query. The database sends the song back to the web server and the web server sends an API response back to the web browser with the song.

Animation captions:

1. A web application uses JavaScript to send an API request. The request is for a song resource with ID 123.
2. The web API requests the song data from the database.
3. The database returns the song results to the web API.
4. The web API returns the song resource to the web browser in an API response.

**PARTICIPATION
ACTIVITY**

11.11.2: RESTful web APIs.

1) Where is data for resources primarily stored?

- ☐ Web browser
- ☐ Database
- ☐ Web server

2) In the animation above, the _____ is a client.

- ☐ Web browser
- ☐ Database
- ☐ Web server

HTTP request verbs

RESTful web APIs typically use four of the HTTP request verbs to implement CRUD operations:

- POST - Create a new resource
- GET - Retrieve a resource
- PUT - Update a resource
- DELETE - Delete a resource

Resources and related data are frequently transmitted with JSON. Ex: A GET request to a Music API may return a JSON-encoded list of songs.

Table 11.11.1: Example requests and responses for a Music API.

Resource	Request verb	Description	Status code
/songs	GET	Get a list of all songs	200 OK
/songs? genre=pop	GET	Get all songs in the pop genre	200 OK
/songs/234	GET	Get song with ID 234	200 OK
/songs/999	GET	Get non-existing song with ID 999	404 Not Found
/songs	POST	Create a new song	201 Created
/songs/234	PUT	Update song with ID 234	204 No Content
/songs/234	DELETE	Delete song with ID 234	204 No Content

**PARTICIPATION
ACTIVITY**

11.11.3: HTTP request verbs.

1) What does a web API return for a GET request with no query string or other path information?

- ☐ One resource
- ☐ No resources
- ☐ All resources

- 2) Which request verb represents an update to an existing resource.
- ☐ GET
 - ☐ POST
 - ☐ PUT
- 3) What response indicates the request was successfully fulfilled by the web server but no message body is returned?
- ☐ 200
 - ☐ 204
 - ☐ 404
- 4) What response indicates the requested resource to be deleted does not exist?
- ☐ 200
 - ☐ 204
 - ☐ 404
- 5) Which request verb is likely to return a 201 response?
- ☐ POST
 - ☐ DELETE
 - ☐ GET
- 6) Which request verb usually does not require the browser to send JSON data to the web server?
- ☐ POST
 - ☐ PUT
 - ☐ GET



Web services and SOAP

The term "web service" is often used synonymously with "web API", but web services are usually associated with XML-based SOAP (Simple Object Access Protocol) over HTTP. SOAP and related technologies provide some benefits over REST, but SOAP web services are generally more complex to build and use than REST-based web APIs.

Express router

To create a web API using Express, a developer uses the **express.Router** class to create modular route callbacks for the web API endpoints. An **endpoint** is a URL for a web API used to access a resource. Ex: `http://yahoo.com/api/movies`.

The example below creates an Express server that listens on port 3000. A route that was created with the **express.Router** returns a single song in JSON format when a GET request is sent to `http://localhost:3000/api/songs`.

PARTICIPATION ACTIVITY

11.11.4: Express router sends JSON-encoded song to web browser.

```
const express = require("express");

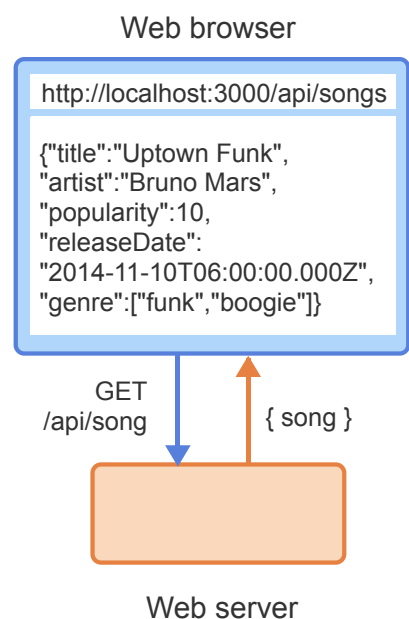
const app = express();
const router = express.Router();

// GET request returns JSON-encoded song
router.get("/songs", function(req, res) {

  const song = {
    title: "Uptown Funk",
    artist: "Bruno Mars",
    popularity: 10,
    releaseDate: new Date(2014, 10, 10),
    genre: ["funk", "boogie"]
  };

  res.json(song);
});

// All requests to API begin with /api
```



```
app.use("/api", router);  
app.listen(3000);
```

Animation content:

A block of Node.js is shown:

```
const express = require("express");  
const app = express();  
const router = express.Router();  
// GET request returns JSON-encoded song  
router.get("/songs", function(req, res) {  
  const song = {  
    title: "Uptown Funk",  
    artist: "Bruno Mars",  
    popularity: 10,  
    releaseDate: new Date(2014, 10, 10),  
    genre: ["funk", "boogie"]  
  };  
  res.json(song);  
});  
// All requests to API begin with /api  
app.use("/api", router);  
app.listen(3000);
```

A web browser is shown with the URL `http://localhost:3000/api/songs`. The web browser sends a GET request with `/api/songs` to the web server. The web server returns `{songs}` and

```
{  
  "title": "Uptown Funk",  
  "artist": "Bruno Mars",  
  "popularity": 10,  
  "releaseDate":  
    "2014-11-10T06:00:00.000Z",  
  "genre": ["funk", "boogie"]  
}
```

displays on the web browser.

Animation captions:

1. Express router creates a `/songs` route.
2. Any URL that begins with `/api` will be routed to the Express router, so `/api/songs` request is routed to `/songs` route.

3. Browser requests song data with API request: `http://localhost:3000/api/songs`
4. The `/songs` route uses `res.json()` to return a JSON-encoded song in the API response.

**PARTICIPATION
ACTIVITY**

11.11.5: Express router.



Refer to the expanded example below, which supports GET and POST requests.

```
const express = require("express");

const app = express();
const router = express.Router();

// Middleware that parses HTTP requests with JSON body
app.use(express.json());

// GET request returns JSON-encoded songs
router.get("/songs", function(req, res) {
  const songs = [
    {
      title: "We Found Love",
      artist: "Rihanna",
      popularity: 10,
      releaseDate: new Date(2011, 9, 22),
      genre: ["electro house"]
    },
    {
      title: "Happy",
      artist: "Pharrell Williams",
      popularity: 10,
      releaseDate: new Date(2013, 11, 21),
      genre: ["soul", "new soul"]
    }
  ];

  res.json(songs);
});

// POST request of JSON-encoded song displays song in the console
router.post("/songs", function(req, res) {
  console.log(req.body);
  res.sendStatus(201);
});

// All requests to API begin with /api
app.use("/api", router);

app.listen(3000);
```

1) What status code does a GET request to `http://localhost:3000/songs` return?

- ☐ 200
- ☐ 201
- ☐ 404

2) What does a GET request to `http://localhost:3000/api/songs` return in the response body?

- ☐ A single song
- ☐ Array of two songs
- ☐ Two individual song objects

3) ____ is middleware that enables HTTP requests containing JSON to be automatically converted into a JavaScript object.

- ☐ `express.json()`
- ☐ `router.post()`
- ☐ `res.sendStatus()`

- 4) What MIME type is missing in the HTTP request below so the Express app outputs the JSON-encoded song to the console?



```
POST /api/songs HTTP/1.1
Host: localhost:3000
Content-Type: _____
User-Agent: Mozilla/5.0
Chrome/48.0.2564

{
  "title": "Stayin' Alive",
  "artist": "Bee Gees",
  "popularity": 9,
  "releaseDate": "1977-02-01",
  "genre": ["disco"]
}
```

- ☐ text/plain
- ☐ application/json
- ☐ text/javascript

Developer tools for interacting with web APIs

Developers use a variety of tools to test and interact with web APIs. [cURL](#) is a popular command-line tool for sending HTTP requests. The image below shows curl sending GET and POST requests to the Music API.

```
$ curl http://localhost:3000/api/songs
{"title":"Uptown Funk","artist":"Bruno Mars","popularity":10,
"releaseDate":"2014-11-10T06:00:00.000Z","genre":["funk","boogie"]}

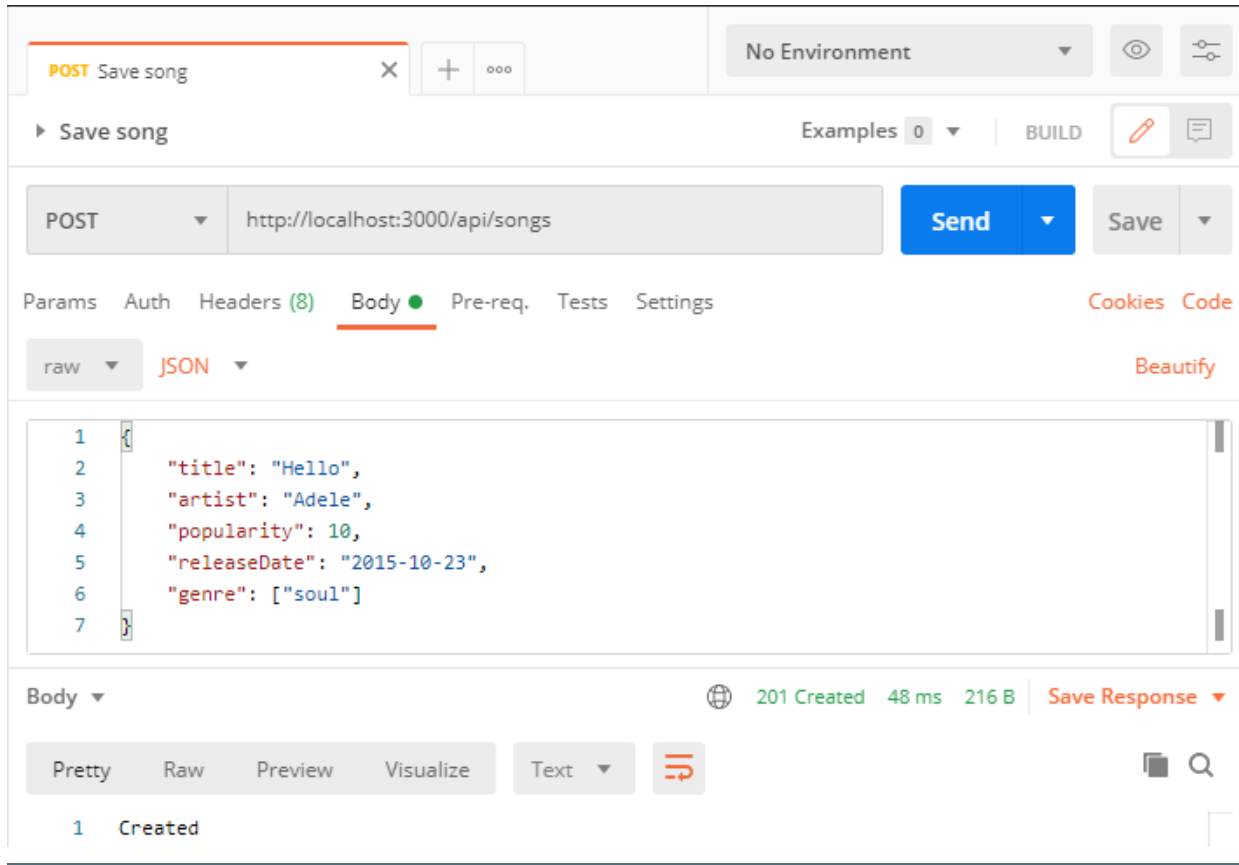
$ curl -H "Content-Type: application/json" -X POST -d
'{"title":"Hello","artist":"Adele"}' http://localhost:3000/api/songs
Created
```

[Postman](#) is a popular application that allows developers to formulate HTTP requests. API queries can be saved into collections for easy retrieval. The screenshot below shows a GET request to the example Music API that returned back a JSON-encoded song.

The screenshot shows the Postman interface for a GET request to `http://localhost:3000/api/songs`. The request is named "Find song" and is in the "No Environment" state. The response is a 200 OK status with a 67 ms response time and 341 B of data. The response body is displayed in the "Body" tab, showing a JSON object with the following structure:

```
1 {
2   "title": "Uptown Funk",
3   "artist": "Bruno Mars",
4   "popularity": 10,
5   "releaseDate": "2014-11-10T06:00:00.000Z",
6   "genre": [
7     "funk",
8     "boogie"
9   ]
10 }
```

The screenshot below shows a POST request in Postman. The Body tab shows the JSON-encoded song to be sent in the POST request. The "raw" and "JSON" options are selected so the HTTP request will automatically include a **Content-Type: application/json** header. After pressing Send, Postman shows the "Created" API response.



Using a database

The previous Music API example does not interact with a database. The figure below uses the **mongoose** module to store and retrieve song information from a MongoDB database. The get route uses the Mongoose **Model.find()** method to retrieve all songs in the database. The post route uses the Mongoose **Document.save()** method to save the posted song to the database.

Figure 11.11.1: Music API using Mongoose to access/save data in MongoDB.

server.js

```
const express =
require("express");
const Song =
require("../models/song");

const app = express();
```

models/song.js

```
const db = require("../db");

// Create a model from the schema
const Song = db.model("Song", {
  title: { type: String, required: true },
  artist: String,
  popularity: { type: Number, min: 1, max: 10 },
  releaseDate: { type: Date, default: Date.now },
  genre: [ String ]
});

module.exports = Song;
```

db.js

```
const mongoose = require("mongoose");
mongoose.connect("mongodb://localhost/musicdb");
module.exports = mongoose;
```

```
// Middleware that parses
HTTP requests with JSON
body
app.use(express.json());

const router =
express.Router();

// Get list of all songs in
the database
router.get("/songs",
function(req, res) {
  Song.find(function(err,
songs) {
    if (err) {
res.status(400).send(err);
    }
    else {
      res.json(songs);
    }
  });
});

// Add a new song to the
database
router.post("/songs",
function(req, res) {

  // Save the song
  const song = new
Song(req.body);
  song.save(function(err,
song) {
    if (err) {
res.status(400).send(err);
    }
    else {
res.status(201).json(song);
    }
  });
});

app.use("/api", router);
app.listen(3000);
```

To manage the complexity of web APIs, developers organize routes and API functionality into

separate Node.js modules. The Node.js project in the below figure defines the song routes in `api/songs.js`. In `server.js`, the song routes are defined in relation to the relative path `/api/songs`, so the `/` route path in `songs.js` means the full path is `/api/songs/` or `/api/songs` without the `/` at the end.

Figure 11.11.2: Organizing routes and APIs in separate modules:
Example Music API project.

```
musicproject
├── api
│   └── songs.js
├── models
│   └── song.js
├── node_modules
│   ├── express
│   ├── mongoose
│   └── etc...
├── db.js
├── package.json
└── server.js
```

api/songs.js

models/song.js

```
const Song =
require("../models/song");
const router =
require("express").Router();

// Get list of all songs in
the database
router.get("/",
function(req, res) {
  Song.find(function(err,
songs) {
    if (err) {

res.status(400).send(err);
    }
    else {
      res.json(songs);
    }
  });
});

// Add a new song to the
database
router.post("/",
function(req, res) {
  const song = new
Song(req.body);
  song.save(function(err,
song) {
    if (err) {

res.status(400).send(err);
    }
    else {
      res.status(201).json(song);
    }
  });
});

module.exports = router;
```

```
const db = require("../db");

// Create a model from the schema
const Song = db.model("Song", {
  title:      { type: String, required: true
},
  artist:     String,
  popularity: { type: Number, min: 1, max: 10
},
  releaseDate: { type: Date, default: Date.now
},
  genre:      [ String ]
});

module.exports = Song;
```

db.js

```
const mongoose = require("mongoose");
mongoose.connect("mongodb://localhost/musicdb");
module.exports = mongoose;
```

server.js

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

// Middleware that parses HTTP requests with
JSON body
app.use(express.json());

app.use("/api/songs", require("../api/songs"));

app.listen(3000);
```

**PARTICIPATION
ACTIVITY**

11.11.6: Web API project.



Use the Music API project in the figure above.

- 1) Which file defines the Mongoose model?
 - ☐ models/song.js
 - ☐ api/songs.js
 - ☐ db.js
- 2) What is the name of the MongoDB database storing song data?
 - ☐ songs
 - ☐ musicdb
 - ☐ db.js
- 3) After POSTing the same JSON data twice to the Music API, what is stored in the MongoDB database?
 - ☐ Nothing
 - ☐ One version of the song
 - ☐ Two versions of the same song with different song IDs
- 4) In which file should the API code to handle the DELETE verb for the path `/api/songs` be added?
 - ☐ models/song.js
 - ☐ server.js
 - ☐ api/songs.js

5) What file should be modified to change the route path `/api/songs` to `/api/music`?

- ☐ `models/song.js`
- ☐ `server.js`
- ☐ `api/songs.js`

Search by genre and ID

The Node.js code below improves the Music API to support searching by genre using the query string parameter "genre".

Figure 11.11.3: GET request can limit results by genre using the query string.

```
router.get("/", function(req, res) {
  let query = {};

  // Check if genre was supplied in query string
  if (req.query.genre) {
    query = { genre: req.query.genre };
  }

  Song.find(query, function(err, songs) {
    if (err) {
      res.status(400).send(err);
    }
    else {
      res.json(songs);
    }
  });
});
```

Request:

`http://localhost:3000/api/songs?genre=funk`

Response:

```
[{"_id":"56f00ce0853096ac12c88ba5","title":"Uptown Funk",
"artist":"Bruno Mars","popularity":10,"__v":0,"genre":
["funk","boogie"],
"releaseDate":"2014-11-10T06:00:00.000Z"},
{"_id":"56f00ce0853096ac12c88ba8","title":"Brick House",
"artist":"Commodores","popularity":8,"__v":0,"genre":
["disco","funk"],
"releaseDate":"1977-08-26T05:00:00.000Z"},
{"_id":"56f00ce0853096ac12c88ba9","title":"Super Freak",
"artist":"Rick James","popularity":8,"__v":0,"genre":["funk"],
"releaseDate":"1981-07-10T05:00:00.000Z"}]
```

Most web APIs support finding a resource by ID. The route in the figure below only executes if a URL contains characters after `/api/songs/`. If the characters do not match a valid ObjectId, the

`err` object will contain an error message that is transmitted to the client.

Figure 11.11.4: GET request to find specific song by ID.

```
router.get("/:id", function(req, res) {  
  // Use the ID in the URL path to find the song  
  Song.findById(req.params.id, function(err, song) {  
    if (err) {  
      res.status(400).send(err);  
    }  
    else {  
      res.json(song);  
    }  
  });  
});
```

Request:

`http://localhost:3000/api/songs/56ec1e48960dd58447478ce0`

Response:

```
{"_id": "56ec1e48960dd58447478ce0", "title": "Humble And Kind",  
"artist": "Tim McGraw", "popularity": 8, "genre": ["country  
pop"], "releaseDate":  
"2016-02-01T06:00:00Z", "__v": 0}
```

PARTICIPATION ACTIVITY

11.11.7: Web API GET parameters.

Refer to the two figures above.

1) The GET request to `/api/songs` returns all songs in the MongoDB database.

- ☐ True
- ☐ False

2) The GET request to `/api/songs?genre=rap` returns an empty array if no songs exist in the MongoDB database with a "rap" genre.

- ☐ True
- ☐ False

3) The GET request to `/api/songs/someid` where `someid` is not a valid ObjectId results in a 400 response.

- ☐ True
- ☐ False

4) The GET request to `/api/songs/someid` where `someid` is a valid ObjectId returns a 404 response if the ID is not in the database.

- ☐ True
- ☐ False

Updating and deleting songs

Web APIs use the PUT verb to update an existing resource. The figure below uses the Mongoose function `updateOne()` to update a song with data transmitted in the body of the PUT request. The Music API returns a 204 status for a successful update, a 400 status if an error occurs, and a 404 status if the song's ID is not found.

Figure 11.11.5: Update song's title and popularity with PUT verb in Music API.

```
router.put("/:id", function(req, res) {  
  // Song to update sent in body of request  
  const song = req.body;  
  
  // Replace existing song fields with updated song  
  Song.updateOne({ _id: req.params.id }, song, function(err, result) {  
    if (err) {  
      res.status(400).send(err);  
    }  
    else if (result.matchedCount === 0) {  
      res.sendStatus(404);  
    }  
    else {  
      res.sendStatus(204);  
    }  
  });  
});
```

HTTP request	HTTP response
<pre>PUT /api/song/56ec1e48960dd58447478cdf HTTP/1.1 Host: localhost:3000 Content-Type: application/json User-Agent: Mozilla/5.0 Chrome/48.0.2564 { "title": "Hello There!", "popularity": 8 }</pre>	<pre>HTTP/1.1 204 No Content Date: Mon, 15 Mar 2021 14:26:20 GMT ETag: W/"66- cZws74LP8XEnwbIEyLgWFw" X-Powered-By: Express</pre>

PUT and PATCH

Some RESTful web APIs support two HTTP methods for updating resources:

- *PUT* for performing a complete update
- *PATCH* for performing a partial update

For simplicity, this material uses *PUT* to perform a partial update and does not include *PATCH*.

Web APIs use the DELETE verb to delete an existing resource. The figure below uses the Mongoose method `deleteOne()` to delete a song whose ID appears in the URL of the DELETE request. The Music API returns a 204 status for a successful delete, a 400 status if an error occurs, and a 404 status if the song's ID is not found.

Figure 11.11.6: Delete a song by ID with DELETE verb in Music API.

```
router.delete("/:id", function(req, res) {  
  Song.deleteOne({ _id: req.params.id }, function(err, result) {  
    if (err) {  
      res.status(400).send(err);  
    }  
    else if (result.matchedCount === 0) {  
      res.sendStatus(404);  
    }  
    else {  
      res.sendStatus(204);  
    }  
  });  
});
```

HTTP request	HTTP response
DELETE /api/song/56ec1e48960dd58447478cdf HTTP/1.1 Host: localhost:3000 User-Agent: Mozilla/5.0 Chrome/48.0.2564	HTTP/1.1 204 No Content Date: Mon, 15 Mar 2021 14:31:35 GMT ETag: W/"a-oQDOV50e1MN2H/N8GYi+8w" X-Powered-By: Express

**PARTICIPATION
ACTIVITY**

11.11.8: Update and delete.



Refer to the two figures above.

1) Where is the song data to update sent in a PUT request?



- ☐ Query string
- ☐ Request body
- ☐ Response body

2) In the figure that updates Adele's song, what song data is changed?



- ☐ _id
- ☐ artist
- ☐ title and popularity

3) A PUT request with the following message body specifies a title that is identical to the song's existing title. What status code does the PUT route return?



```
{  
  "title": "Hello"  
}
```

- ☐ 204
- ☐ 400
- ☐ 404

4) What status code do the PUT and DELETE routes return if the supplied song ID cannot be located in the database?

- ☐ 204
- ☐ 400
- ☐ 404

CORS

A web API that needs to be accessed from different origins needs to enable CORS. The **`cors` module** provides middleware to enable CORS in an Express app. Install `cors` with `npm`: `npm install cors`.

```
const express = require("express");
const cors = require("cors");
const app = express();

// Enable all CORS requests
app.use(cors());
```

Exploring further:

- [Postman](#)
- [cURL](#)
- [Understanding SOAP and REST Basics And Differences](#)
- [PUT vs. PATCH](#)
- [cors middleware](#)

11.12 Using RESTful web APIs with Fetch

Music API

A web application uses a web API running on the web server to manipulate data in the server's database. This section uses a RESTful Music API that was developed elsewhere in this material. The Music API uses a song resource with the following fields:

- `_id` - Object ID uniquely identifying each song
- `title` - Required string
- `artist` - String
- `popularity` - Number between 1 and 10
- `releaseDate` - Date
- `genre` - Array of strings

The Music API supports the operations in the below table.

Table 11.12.1: Music API operations.

Resource	Request verb	Description	Status code
/api/songs	GET	Gets a list of all songs in the database	200 OK
/api/songs/:id	GET	Gets the song with the given id	200 OK
/api/songs	POST	Adds the JSON-encoded song in the request body to the database	201 Created
/api/songs	PUT	Updates the existing song with the JSON-encoded song in the request body	204 No Content
/api/songs/:id	DELETE	Removes the song with the given id from the database	204 No Content



- 1) A POST request to `/api/songs` returns all songs in the database.
☐ True
☐ False
- 2) A GET request to `/api/songs/888` returns the song with ID 888.
☐ True
☐ False
- 3) Only the POST and PUT requests require sending JSON to the web API.
☐ True
☐ False

Getting songs

Web applications call a web API from the browser by using the `XMLHttpRequest` object, the Fetch API, or some other library that can send HTTP requests. The example code below uses Fetch to interact with the Music API. The Music API is housed on an Express web server, and song data is saved in a MongoDB database.

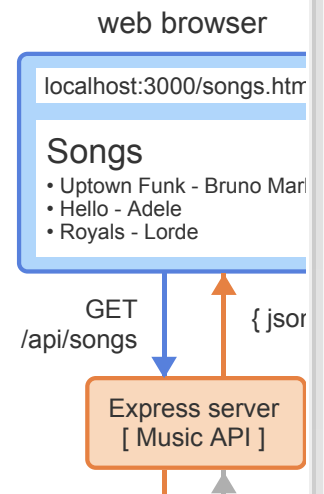
PARTICIPATION ACTIVITY

11.12.2: Using Fetch to request all songs from the Music API.

```
<body>
  <h1>Songs</h1>
  <ul>
    <li>Uptown Funk - Bruno Mars</li>
    <li>Hello - Adele</li>
    <li>Royals - Lorde</li>
  </ul>
</body>
```

```
addEventListener("DOMContentLoaded", async function() {
  const response = await fetch("/api/songs");
  const songs = await response.json();

  let html = "";
```



```
for (let song of songs) {
  html += `<li>${song.title} - ${song.artist}</li>`;
}

document.querySelector("ul").innerHTML = html;
});
```

Get all songs ▼ | { json

```
[
  { "_id": "56f00ba5",
    "title": "Uptown Funk",
    ... },
  { "_id": "56f00ba8",
    "title": "Hello",
    ... },
  { "_id": "56f00cb2",
    "title": "Royals",
    ... }
]
```

MongoDB

Animation content:

An HTML block is shown:

```
<body>
  <h1>Songs</h1>
  <ul>
    <li>Uptown Funk - Bruno Mars</li>
    <li>Hello - Adele</li>
    <li>Royals - Lorde</li>
  </ul>
</body>
```

A block of JavaScript is shown:

```
addEventListener("DOMContentLoaded", async function() {
  const response = await fetch("/api/songs");
  const songs = await response.json();
  let html = "";
  for (let song of songs) {
    html += `<li>${song.title} - ${song.artist}</li>`;
  }
  document.querySelector("ul").innerHTML = html;
});
```

A web browser with the URL localhost:3000/songs.html sends a GET request to the Express server for /api/songs. The Express server gets all songs from the MongoDB database and returns the songs back to the web browser encoded in JSON. The web browser then displays the following:

Songs

- Uptown Funk - Bruno Marks

- Hello - Adele
- Royals - Lorde

Animation captions:

1. Browser requests songs.html from the Express web server and renders the HTML with an empty list.
2. fetch() sends a GET request using the URL /api/songs.
3. Web server's Music API retrieves all songs from MongoDB and returns the songs encoded in JSON.
4. response.json() parses the JSON-encoded songs and returns an array of song objects.
5. The for-of loop creates elements containing each song title and artist. The elements are added to , creating a bulleted list of songs.

PARTICIPATION ACTIVITY

11.12.3: Getting songs with Fetch.

- 1) What is missing to output the first song's artist to the console?

```
const response = await
fetch("/api/songs");
const songs = await
response.json();
console.log(_____);
```

- ☐ songs.artist
- ☐ songs[0].artist
- ☐ songs.artist[0]

- 2) What is missing to output the song's popularity to the console?



```
const response = await
fetch("/api/songs/123");
const song = await
response.json();
console.log(_____);
```

- ☐ song[0].popularity
- ☐ popularity.song
- ☐ song.popularity

- 3) What is missing to ensure that the Music API returns a valid song?



```
const response = await
fetch("/api/songs/888");
if (_____) {
    const song = await
response.json();
    console.log(song.title);
}
else {
    console.log("Can't get
song");
}
```

- ☐ response.ok
- ☐ response
- ☐ song

Adding new songs

The `fetch()` method can send a POST request to a web API by specifying a second argument containing the request method, the Content-Type header, and the data to send in the request body. The figure below displays a form allowing the user to enter song data. Clicking the Add button results in `fetch()` POSTing the JSON-encoded song data from the web form to the Music API.

Figure 11.12.1: POSTing a new song.

```
<!-- add_song.html -->
```

```

<form>
  <p>
    <label
for="title">Title:</label>
    <input type="text"
id="title">
  </p>
  <p>
    <label
for="artist">Artist:</label>
    <input type="text"
id="artist">
  </p>
  <p>
    <label
for="released">Released:
</label>
    <input type="date"
id="released">
  </p>
  <p>
    <label
for="popularity">Popularity:
</label>
    <input type="number"
min="1" max="10"
id="popularity">
  </p>
  <p>
    <label
for="genre">Genre:</label>
    <input type="text"
id="genre">
  </p>
  <p>
    <input type="button"
id="addBtn" value="Add">
  </p>
  <p id="error"></p>
</form>

```

```

addEventListener("DOMContentLoaded", function() {

document.querySelector("#addBtn").addEventListener(
addSong);
});

async function addSong() {
  // Create a song object from the form fields
  const song = {
    title: document.querySelector("#title").value,
    artist: document.querySelector("#artist").value,
    releaseDate:
document.querySelector("#released").value,
    popularity:
document.querySelector("#popularity").value,
    genre: document.querySelector("#genre").value,
    document.querySelector("#genre").value.split(" ")
  };

  // POST a JSON-encoded song to Music API
  const response = await fetch("/api/songs", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(song)
  });

  if (response.ok) {
    const results = await response.json();
    alert("Added song with ID " + results._id);

    // Reset the form after adding the song
    document.querySelector("form").reset();
  }
  else {
    document.querySelector("#error").innerHTML =
add song.";
  }
}

```

Add Song

×

+

←

→

↻

localhost:3000/add_song.html

Add Song

Title:

Never Let Me Down Again

Artist:

Depeche Mode

Released:

08 / 25 / 1987

Popularity:

6

Genre:

synthpop,ebm

Add

**PARTICIPATION
ACTIVITY**

11.12.4: POST requests and the Music API.

- 1) The `JSON.stringify()` method converts a JavaScript object to a JSON string.
☐ True
☐ False
- 2) The HTTP request must have the header **Content-Type:** **application/json** or the web server will not properly accept the JSON-encoded data that is POSTed.
☐ True
☐ False

3) In the figure above, the web browser displays a dialog box when the browser receives a 400 status code from the web server.

- ☐ True
- ☐ False

4) If the Music API returns a 400 status code, the webpage displays an error message.

- ☐ True
- ☐ False

Updating songs

The example below uses `fetch()` to load a song into a web form when the page first loads. The user can then modify the song data. When the user clicks the Update button, the `updateSong()` function calls `fetch()` to send a PUT request to the Music API to update the song.

Figure 11.12.2: Sending PUT request to update an existing song.

```
<!-- update_song.html -->

<form>
  <p>
    <label
for="songId">ID:</label>
    <input type="text"
id="songId">
  </p>
  <p>
    <label
for="title">Title:</label>
    <input type="text"
id="title">
  </p>
  <p>
    <label
for="artist">Artist:</label>
    <input type="text"
id="artist">
  </p>
  <input type="button" value="Update" />
</form>
```

```
addEventListener("DOMContentLoaded", async function() {
  document.querySelector("#updateBtn").addEventListener(
    click, updateSong);

  // Load a song into the web form
  const songId = "5fe1097caf3b173148985746";
  const response = await fetch("/api/songs/" + songId);
  if (response.ok) {
    let song = await response.json();
    document.querySelector("#songId").value = song.id;
    document.querySelector("#title").value = song.title;
    document.querySelector("#artist").value = song.artist;
    document.querySelector("#released").value =
      song.releaseDate.substring(0, 10);
    document.querySelector("#popularity").value =
      song.popularity;
    document.querySelector("#genre").value = song.genre;
  }
});
```

```

    <label
for="released">Released:
</label>
    <input type="date"
id="released">
    </p>
    <p>
    <label
for="popularity">Popularity:
</label>
    <input type="number"
min="1" max="10"
id="popularity">
    </p>
    <p>
    <label
for="genre">Genre:</label>
    <input type="text"
id="genre">
    </p>
    <p>
    <input type="button"
id="updateBtn"
value="Update">
    </p>

    <p id="error"></p>
</form>

```

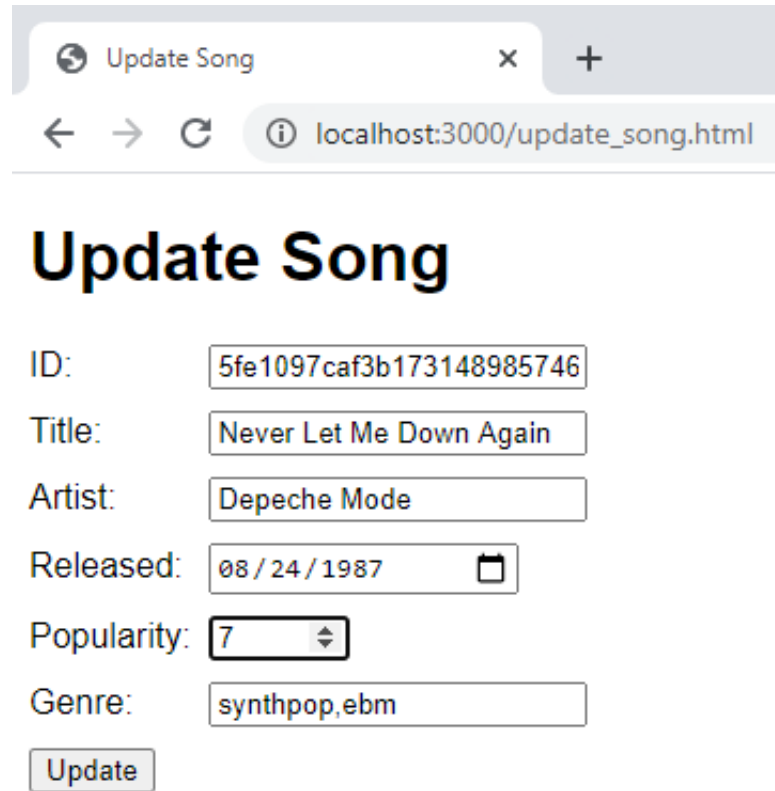
```

async function updateSong() {
    // Create a song object from the form fields
    const song = {
        _id: document.querySelector("#songId").value,
        title: document.querySelector("#title").value,
        artist: document.querySelector("#artist").value,
        releaseDate: document.querySelector("#releaseDate").value,
        popularity: document.querySelector("#popularity").value,
        genre: document.querySelector("#genre").value,
        // ... other fields ...
    };

    // Send PUT request with JSON-encoded song to API
    const response = await fetch("/api/songs", {
        method: "PUT",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(song)
    });

    if (response.ok) {
        alert("Updated song.");
    } else {
        document.querySelector("#error").innerHTML =
            "Error updating song.";
    }
}

```

Update Song


← → ↻ ⓘ localhost:3000/update_song.html


Update Song

ID:

Title:

Artist:

Released: 

Popularity: 

Genre:

**PARTICIPATION
ACTIVITY**

11.12.5: PUT request and the Music API.

Refer to the figure above.

- 1) What happens when the page first loads if the database does not contain a song with ID 5fe0f679af3b173148985743?
- ☐ Error message is displayed
 - ☐ Form displays a default song
 - ☐ Form displays no song data

2) In the examples above, what fields are different in the calls to `fetch()` when creating a POST request vs. a PUT request?

- ☐ method
- ☐ headers
- ☐ body

3) What happens if the user changes the title to "Little 15" and clicks Update?

- ☐ Nothing happens
- ☐ A dialog box displays
- ☐ The song's title is left unchanged

4) What happens if the user removes a character from the ID text box and clicks Update?

- ☐ Nothing happens
- ☐ The song's ID is updated
- ☐ An error message displays

Deleting songs

The `getAllSongs()` function below uses `fetch()` to load all song titles into a drop-down list. When the user clicks Delete, `deleteSong()` calls `fetch()` to send a DELETE request to delete the selected song.

Figure 11.12.3: Sending DELETE request to delete a song.

```
addEventListener("DOMContentLoaded", async function() {  
    document.querySelector("#deleteBtn").addEventListener("click",  
        deleteSong);  
    getAllSongs();  
});
```

```
<!--
delete_song.html -
-->

<form>
  <select
id="songDropDown">
    </select>

    <input
type="button"
value="Delete"
id="deleteBtn">
    <div
id="error"></div>
</form>
```

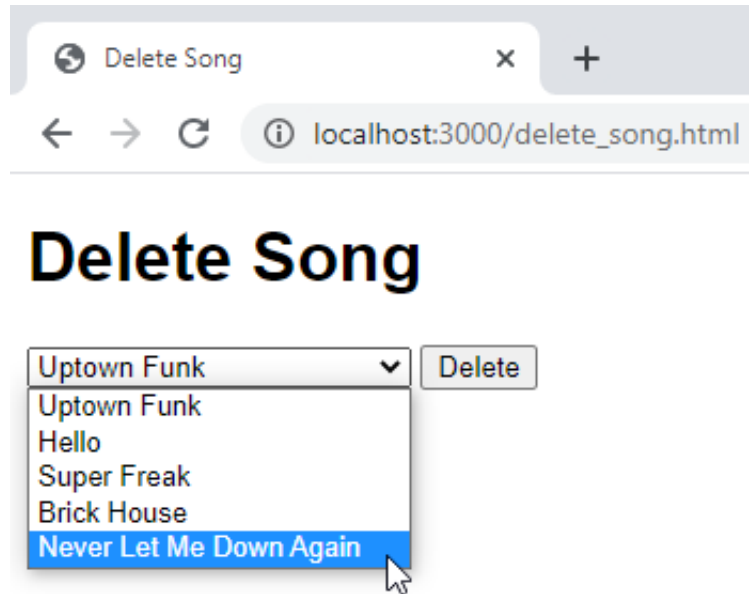
```
// Load all songs into the drop-down list
async function getAllSongs() {
  const response = await fetch("/api/songs");
  if (response.ok) {
    const songs = await response.json();
    let html = "";
    for (let song of songs) {
      html += `<option value="${song._id}">${song.title}
</option>`;
    }

    document.querySelector("#songDropDown").innerHTML =
html;
  }
}

async function deleteSong() {
  // Get the song ID of the selected song
  const songId = document.querySelector(
    "#songDropDown option:checked").value;

  const response = await fetch("/api/songs/" + songId, {
    method: "DELETE"
  });

  if (response.ok) {
    // Successfully deleted song
    getAllSongs();
  }
  else {
    document.querySelector("#error").innerHTML = "Cannot
delete song.";
  }
}
```

**PARTICIPATION
ACTIVITY**

11.12.6: DELETE request and the Music API.

Refer to the figure above.

1) According to the `fetch()` call in `deleteSong()`, what is sent in the request body?

- ☐ JSON
- ☐ song ID
- ☐ Nothing

- 2) What effect would the code below have if placed immediately before the for-of loop in `getAllSongs()`?



```
songs.sort(function(a, b) {  
  const title1 =  
a.title.toLowerCase();  
  const title2 =  
b.title.toLowerCase();  
  if (title1 < title2) {  
    return -1;  
  }  
  if (title1 > title2) {  
    return 1;  
  }  
  return 0;  
});
```

- ☐ No effect
- ☐ Sorts titles in the drop-down list in ascending (A to Z) order
- ☐ Sorts titles in the drop-down list in descending (Z to A) order
- 3) What happens after the user clicks the Delete button?
- ☐ The selected song is deleted from the database but remains in the drop-down list.
- ☐ The selected song is removed from the drop-down list but remains in the database.
- ☐ The selected song is removed from the drop-down list and the database.



Exploring further:

- [Fetch API \(MDN\)](#)
- [JavaScript Fetch API Tutorial with JS Fetch Post and Header Examples](#)

11.13 Using RESTful web APIs with jQuery

Music API

A web application uses a web API running on the web server to manipulate data in the server's database. This section uses a RESTful Music API that is developed elsewhere in this material. The Music API uses a song resource with the following fields:

- `_id` - Object ID uniquely identifying each song
- `title` - Required string
- `artist` - String
- `popularity` - Number between 1 and 10
- `releaseDate` - Date
- `genre` - Array of strings

The Music API supports the operations in the below table.

Table 11.13.1: Music API operations.

Resource	Request verb	Description	Status code
/api/songs	GET	Gets a list of all songs in the database	200 OK
/api/songs/:id	GET	Gets the song with the given id	200 OK
/api/songs	POST	Adds the JSON-encoded song in the request body to the database	201 Created
/api/songs	PUT	Updates the existing song with the JSON-encoded song in the request body	204 No Content
/api/songs/:id	DELETE	Removes the song with the given id from the database	204 No Content

PARTICIPATION
ACTIVITY

11.13.1: Music API.

- 1) A POST request to `/api/songs` returns all songs in the database.
- ☐ True
- ☐ False
- 2) A GET request to `/api/songs/888` returns the song with ID 888.
- ☐ True
- ☐ False
- 3) Only the POST and PUT requests require sending JSON to the web API.
- ☐ True
- ☐ False

Getting songs

Web applications call a web API from the browser by using the `XMLHttpRequest` object to make Ajax requests. jQuery provides Ajax methods like `$.get()` to make calling a web API easier than using the `XMLHttpRequest` object directly.

The example code below uses jQuery to interact with the Music API. The Music API is housed on a web server running Express, and song data is saved in a MongoDB database.

PARTICIPATION
ACTIVITY11.13.2: Using `$.get()` to request all songs from the Music API.

```
<!DOCTYPE html>
<html lang="en">
<title>Songs</title>
<script
  src="https://code.jquery.com/jquery-3.4.1.min.js"
  integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo="
  crossorigin="anonymous"></script>
</script>
```

Web browser

localhost:3000/songs.htm

Songs

- Uptown Funk - Bruno Mar
- Hello - Adele
- Royals - Lorde

```

$(function() {
  let $list = $("ul");
  $.get("/api/songs", function(songs) {
    songs.forEach(function(song) {
      $list.append("<li>" + song.title + " - " +
        song.artist + "</li>");
    });
  });
});

</script>
<body>
  <h1>Songs</h1>
  <ul>
    <li>Uptown Funk - Bruno Mars</li>
    <li>Hello - Adele</li>
    <li>Royals - Lorde</li>
  </ul>
</body>
</html>

```



Animation content:

The following code is displayed.

```

<!DOCTYPE html>
<html lang="en">
<title>Songs</title>
<script
  src="https://code.jquery.com/jquery-3.4.1.min.js"
  integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFIBw8HfCJo="
  crossorigin="anonymous"></script>
<script>
$(function() {
  let $list = $("ul");
  $.get("/api/songs", function(songs) {
    songs.forEach(function(song) {
      $list.append("<li>" + song.title + " - " +
        song.artist + "</li>");
    });
  });
});
</script>

```



```
<body>
  <h1>Songs</h1>
  <ul>

  </ul>
</body>
</html>
```

Step 1: The browser's request to the Express web server reads "localhost:3000/songs.html". The HTML is then rendered, and "Songs" is printed on the screen.

Step 2: The line of code reading "\$.get('/api/songs', function(songs) {" is highlighted.

Step 3: The list of all songs in MongoDB reads

```
"[
  { "_id":"56f00ba5",
    "title":"Uptown Funk",
    ... },
  { "_id":"56f00ba8",
    "title":"Hello",
    ... },
  { "_id":"56f00cb2",
    "title":"Royals",
    ... }
]".
```

Step 4: The lines of code reading "songs.forEach(function(song) { \$list.append("" + song.title + " - " + song.artist + "");" are highlighted.

The code is updated to read the following.

```
<!DOCTYPE html>
<html lang="en">
<title>Songs</title>
<script
  src="https://code.jquery.com/jquery-3.4.1.min.js"
```

```
integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFIBw8HfCJo="
crossorigin="anonymous"></script>
<script>
$(function() {
  let $list = $("ul");
  $.get("/api/songs", function(songs) {
    songs.forEach(function(song) {
      $list.append("<li>" + song.title + " - " +
        song.artist + "</li>");
    });
  });
});
</script>
<body>
  <h1>Songs</h1>
  <ul>
<li>Uptown Funk - Bruno Mars</li>
<li>Hello - Adele</li>
<li>Royals - Lorde</li>
  </ul>
</body>
</html>
```

The output now reads

"Songs

- Uptown Funk - Bruno Mars
- Hello - Adele
- Royals - Lorde".

Animation captions:

1. Browser requests songs.html from the Express web server and renders the HTML.
2. jQuery code executes, and \$.get() makes an Ajax request to /api/songs.
3. Web server's Music API gets a list of all songs from MongoDB and returns the songs encoded in JSON.
4. \$.get() callback function is called with the song data. The forEach() method appends elements to the for each song title and artist.

**PARTICIPATION
ACTIVITY**

11.13.3: Getting songs with jQuery.

- 1) What is missing to output the first song's artist to the console?



```
$.get("/api/songs",  
function(songs) {  
    console.log(____);  
});
```

- ☐ songs.artist
- ☐ songs[0].artist
- ☐ songs.artist[0]

- 2) What is missing to output the song's popularity to the console?



```
$.get("/api/songs/123",  
function(song) {  
    console.log(____);  
});
```

- ☐ song[0].popularity
- ☐ popularity.song
- ☐ song.popularity

3) Why would the `fail()` callback execute?



```
$.get("/api/songs/123",  
function(song) {  
    console.log(song);  
})  
.fail(function() {  
    console.log("Fail");  
});
```

- ☐ Database has only one song with ID 123
- ☐ Database does not have any songs
- ☐ Database has no room for more songs

Adding new songs

The jQuery method `$.post()` makes a POST request to a web API. However, `$.post()` is not ideal for sending JSON-encoded data. Instead, the jQuery method `$.ajax()` POSTs JSON-encoded data to a web API.

The figure below displays a form allowing the user to enter song data. Clicking the Add button results in the `$.ajax()` method POSTing the JSON-encoded song data from the web form to the Music API.

Figure 11.13.1: jQuery `$.ajax()` method POSTing a new song.

```
<!-- add_song.html -->  
  
<form>  
  <p>  
    <label for="title">Title:</label>  
    <input type="text" id="title">  
  </p>  
  <p>  
    <label for="artist">Artist:</label>  
    <input type="text" id="artist">  
  </p>  
  <p>  
    <label for="released">Released:</label>  
    <input type="date" id="released">  
  </p>  
</form>
```

```
</p>
<p>
  <label for="popularity">Popularity:</label>
  <input type="number" min="1" max="10"
    id="popularity">
</p>
<p>
  <label for="genre">Genre:</label>
  <input type="text" id="genre">
</p>
<input type="button" id="addBtn" value="Add">

<p id="error"></p>
</form>
```

```
$("#addBtn").click(function() {
  let genre = [];
  if ($("#genre").val()) {
    // Create an array from comma-separated values
    genre = ($("#genre").val().split(",");
  }

  // Create a song object from the form fields
  const song = {
    title: $("#title").val(),
    artist: $("#artist").val(),
    releaseDate: $("#released").val(),
    popularity: $("#popularity").val(),
    genre: genre
  };

  // POST JSON-encoded song to Music API
  $.ajax({
    type: "POST",
    url: "/api/songs",
    data: JSON.stringify(song),
    contentType: "application/json"
  }).done(function(data) {
    // Reset the form after adding the song
    $("form").trigger("reset");
  }).fail(function(jqXHR) {
    $("#error").html("Song could not be added.");
  });
});
```

Add Song

×

+

← → ↻ ⓘ localhost:3000/add_song.html

Add Song

Title:

Never Let Me Down Again

Artist:

Depeche Mode

Released:

08 / 25 / 1987

Popularity:

6

Genre:

synthpop,ebm

Add

**PARTICIPATION
ACTIVITY**

11.13.4: POST requests to the Music API.

- 1) The `JSON.stringify()` method converts a JavaScript object to a JSON string.
☐ True
☐ False
- 2) The HTTP request must have the header **Content-Type:** **application/json** or the web server will not properly accept the JSON-encoded data that is POSTed.
☐ True
☐ False

- 3) The `$.ajax()` method returns a `jqXHR` (jQuery XMLHttpRequest) object.
- ☐ True
- ☐ False
- 4) The `jqXHR.done()` method's callback function is called before the browser receives the entire Ajax response.
- ☐ True
- ☐ False
- 5) The `jqXHR.error()` method's callback function is called when the browser receives a 200 response from the web server.
- ☐ True
- ☐ False
- 6) Saving a song with popularity set to 1000 causes the `jqXHR.error()` method's callback function to display an error message.
- ☐ True
- ☐ False
- 7) The code below resets the values of all elements in the form.
- ```
$("form").trigger("reset");
```
- ☐ True
- ☐ False

## Updating and deleting songs

The `$.ajax()` method supports GET, POST, PUT, or DELETE requests. The example below uses

`$.get()` to load a song into a web form, and then uses `$.ajax()` to send a PUT request to update the song.

Figure 11.13.2: jQuery `$.ajax()` method sending PUT request to update an existing song.

```
<!-- update_song.html -->

<form>
 <p>
 <label for="songId">ID:</label>
 <input type="text" id="songId">
 </p>
 <p>
 <label for="title">Title:</label>
 <input type="text" id="title">
 </p>
 <p>
 <label for="artist">Artist:</label>
 <input type="text" id="artist">
 </p>
 <p>
 <label for="released">Released:</label>
 <input type="date" id="released">
 </p>
 <p>
 <label for="popularity">Popularity:</label>
 <input type="number" min="1" max="10"
 id="popularity">
 </p>
 <p>
 <label for="genre">Genre:</label>
 <input type="text" id="genre">
 </p>
 <input type="button" id="updateBtn" value="Update">

 <p id="error"></p>
</form>
```

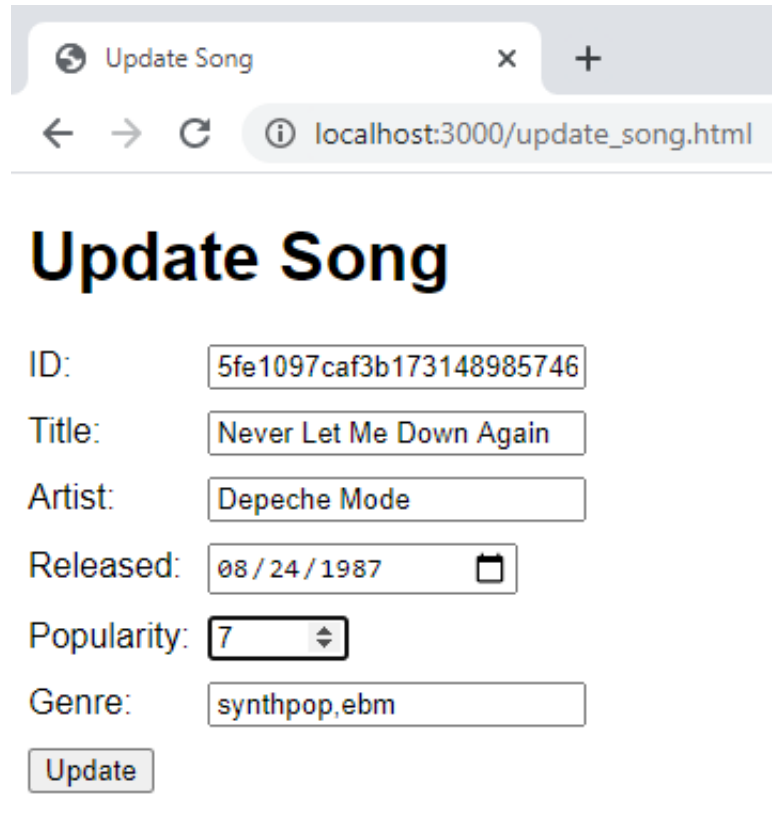


```
// Load a song into the web form
const songId = "5fe1097caf3b173148985746";
$.get("/api/songs/" + songId, function(song) {
 $("#songId").val(song._id);
 $("#title").val(song.title);
 $("#artist").val(song.artist);
 $("#released").val(song.releaseDate.substring(0, 10));
 $("#popularity").val(song.popularity);
 $("#genre").val(song.genre);
});

$("#updateBtn").click(function() {
 let genre = [];
 if ($("#genre").val()) {
 // Create an array from the comma-separated values
 genre = ($("#genre").val().split(","));
 }

 // Create a song object from the form fields
 const song = {
 _id: $("#songId").val(),
 title: $("#title").val(),
 artist: $("#artist").val(),
 releaseDate: $("#released").val(),
 popularity: $("#popularity").val(),
 genre: genre
 };

 // Send PUT request with updated song data
 $.ajax({
 type: "PUT",
 url: "/api/songs",
 data: JSON.stringify(song),
 contentType: "application/json"
 }).done(function(data) {
 // Successfully updated a song
 $("#form").trigger("reset");
 }).fail(function(jqXHR) {
 $("#error").html("The song could not be updated.");
 });
});
```



Update Song

ID:

Title:

Artist:

Released:

Popularity:

Genre:

The example below uses `$.get()` to load all song titles into a drop-down list. When the user clicks Delete, `$.ajax()` sends a DELETE request to delete the selected song.

Figure 11.13.3: jQuery `$.ajax()` method sending DELETE request to remove a song.

```
<!-- delete_song.html -->

<form>
 <select>
 </select>

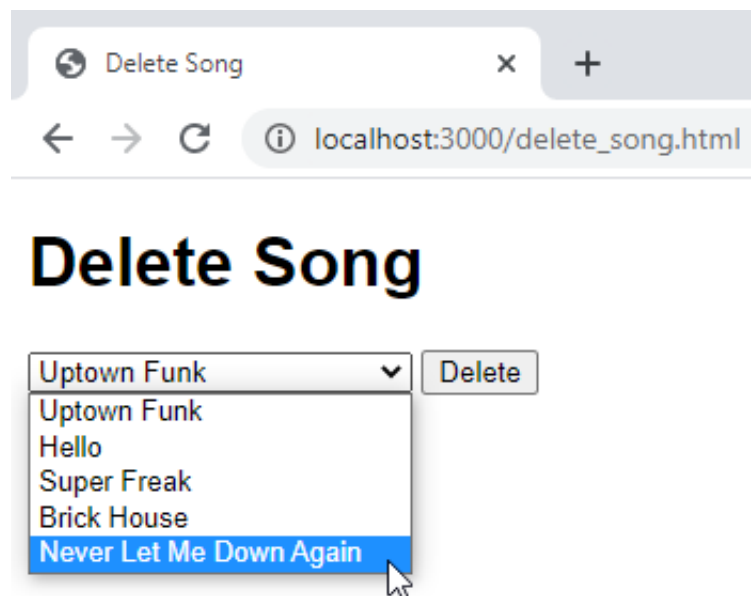
 <input type="button" value="Delete"
 id="deleteBtn">
 <p id="error"></p>
</form>
```

```
getAllSongs();

// Load all songs into the drop-down list
function getAllSongs() {
 $.get("/api/songs", function(songs) {
 let $select = $("select");
 $select.html("");
 songs.forEach(function(song) {
 $select.append("<option value='" + song._id + "'>" +
 song.title + "</option>");
 });
 });
}

$("#deleteBtn").click(function() {
 // Get the song ID of the selected song
 const songId = $("select :selected").val();

 $.ajax({
 type: "DELETE",
 url: "/api/songs/" + songId
 }).done(function(data) {
 // Successfully deleted song
 getAllSongs();
 }).fail(function(jqXHR) {
 $("#error").html("The song could not be deleted.");
 });
});
```



- 1) In the examples above, what fields are different in the calls to `$.ajax()` when creating a POST request vs. a PUT request?
- ☐ type
- ☐ url
- ☐ data
- 2) In the examples above, which request verb does not send data in the "data" field when calling `$.ajax()`?
- ☐ POST
- ☐ PUT
- ☐ DELETE
- 3) What effect would the code below have if placed immediately before the `songs.forEach(...)` loop in the figure above?



```
songs.sort(function(a, b) {
 const title1 =
a.title.toLowerCase();
 const title2 =
b.title.toLowerCase();
 if (title1 < title2) {
 return -1;
 }
 if (title1 > title2) {
 return 1;
 }
 return 0;
});
```

- ☐ No effect.
- ☐ The titles in the drop-down list would be sorted in ascending (A to Z) order.
- ☐ The titles in the drop-down list would be sorted in descending (Z to A) order.

4) In the figure above, why is the `getAllSongs()` function called in the `done()` callback function?

- ☐ The song is not deleted until `getAllSongs()` is called.
- ☐ The `done()` callback function must call at least one other function.
- ☐ To remove the deleted song from the drop-down list.

## Keeping the user interface updated

Suppose two users named Bob and Sue view the webpage that lists all songs. Bob deletes a song, and the song is removed from Bob's drop-down list of songs. However, the deleted song remains in Sue's drop-down list until Sue reloads the webpage or deletes a song and gets an updated list of available songs. Keeping a website's user interface consistent or up-to-date when multiple users are accessing the same data is a significant challenge for developers. Two general solutions exist:

1. **Polling** - The web browser sends periodic requests to the web server asking if the data has changed.
2. **Pushing** - The web server pushes updates to all web browsers as soon as the data is changed on the web server.

Polling is typically implemented with Ajax, but can be problematic: The browser can become out of synch with the web server in the time between poll requests. Pushing keeps the browser and server better synchronized. Pushing is implemented with WebSockets or server-sent events (SSE).

### PARTICIPATION ACTIVITY

11.13.6: Keeping the UI updated.

1) Suppose Bob and Sue access the webpage that displays the drop-down list of songs. If Bob deletes a song, and then Sue selects the same song to delete, what will Sue see after pressing Delete?

- ☐ A "404 Not Found" message.
- ☐ An error message that reads, "The song could not be deleted."
- ☐ No error message.

2) If a developer modifies the above example to call `getAllSongs()` every 5 seconds to keep the list of available songs current, the developer has implemented a \_\_\_\_ solution.

- ☐ polling
- ☐ pushing

3) \_\_\_\_ requires the web server to keep a network connection open to every browser that needs to receive a message.

- ☐ Polling
- ☐ Pushing

Exploring further:

- [jQuery Ajax methods](#)
- [WebSockets \(MDN\)](#)
- [Server-sent events \(MDN\)](#)

# 11.14 Third-party web APIs (Node)

## Introduction

Many organizations have created public web APIs that provide access to the organization's data or the user's data that is stored by the organization. Ex: The Google Maps API provides applications information about geographic locations, and the Instagram API allows applications access to photos shared on Instagram. [Public APIs](#) on GitHub.com lists thousands of free, public web APIs.

A **third-party web API** is a public web API that is used by a web application to perform some operation. "Third-party" refers to a person or organization that is neither the web application using the API nor the user using the web application, which are the "first" and "second" parties. Websites rely on third-party web APIs to integrate with social media, obtain maps and weather data, or access collections of data.

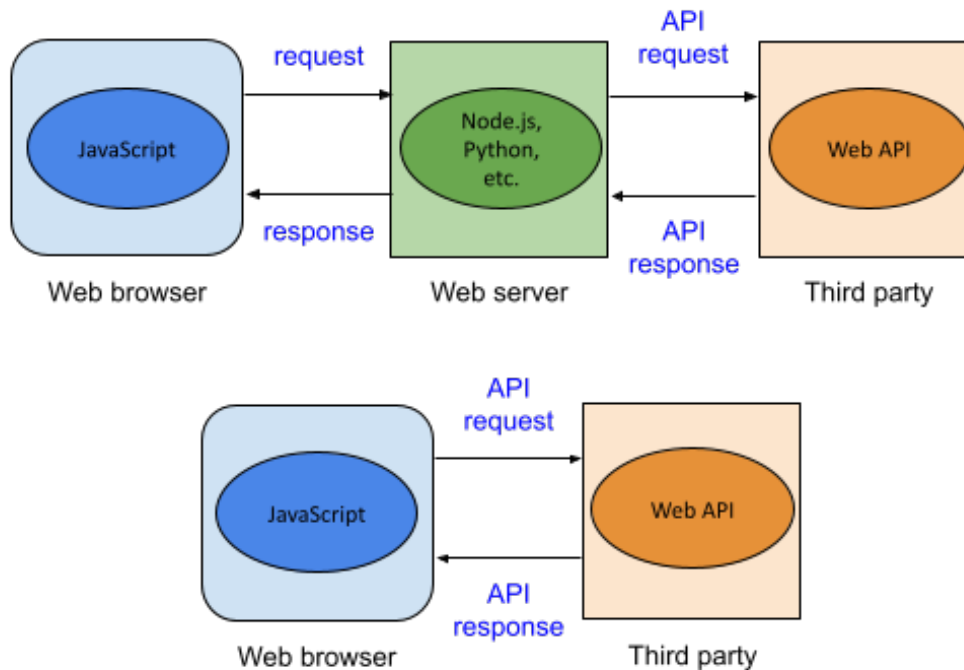
To use a third-party web API, a developer usually registers with the third party to obtain an API key. Third parties require an API key for several reasons:

- The API key identifies who or what application is using the web API.
- The API key helps the third party limit the number of requests made to the API in a fixed time period or may be used to charge a developer a fee for additional requests.
- To obtain an API key, developers must agree to restrictions the third party places on data obtained from the web API.

This material discusses third-party web APIs that are RESTful. A **RESTful web API** is a web API that is called with a URL that specifies API parameters and returns JSON or XML containing the API data. Ex: The URL `http://linkedin.com/api/article?id=123` specifies the article ID 123, so the article would be returned formatted in JSON. Some APIs are SOAP-based, which are generally more complex to use than RESTful web APIs. And some APIs, like the Google Maps API, require developers to use a specific JavaScript library.

Third-party web APIs may be called from the web server or the web browser. This material shows how to call web APIs from an Express web server.

Figure 11.14.1: Calling third-party web API from the web browser or web server.



**PARTICIPATION  
ACTIVITY**

11.14.1: Third-party web APIs.

1) Information from a third-party web API will reach the browser faster if the browser calls the web API directly instead of the web server calling the web API.

- ☐ True
- ☐ False

2) For a third-party web API requiring an API key, the API key must be transmitted for every API request.

- ☐ True
- ☐ False



3) When the browser makes an API request to a third-party web API, the web API key can be kept secret from prying eyes.

- ☐ True  
☐ False

4) Many web APIs charge a fee to the developer after a limited number of requests have been made in a 24-hour period.

- ☐ True  
☐ False

5) Developers always use jQuery to call third-party web APIs from the web server.

- ☐ True  
☐ False

## Weather API

OpenWeatherMap provides a free [Weather API](https://openweathermap.org/api) providing current weather data, forecasts, and historical data. Developers must register at [openweathermap.org](https://openweathermap.org) for an API key that must be transmitted in all API requests.

The OpenWeatherMap website provides documentation explaining how to use the Weather API using GET requests with various query string parameters. The API endpoint `http://api.openweathermap.org/data/2.5/weather` returns the current weather based on the following query string parameters:

- zip - Five digit US ZIP code
- units - Standard, metric, or imperial units to use for measurements like temperature and wind speed
- appid - Developer's API key

Other parameters are documented in the OpenWeatherMap website. The Weather API returns weather data in JSON format by default.

## Figure 11.14.2: GET request to obtain the current weather for ZIP 90210.

©zyBooks 04/15/24 16:48 2071381

Marco Aguilar

CIS192\_193\_Spring\_2024

[http://api.openweathermap.org/data/2.5/weather?](http://api.openweathermap.org/data/2.5/weather?zip=90210&units=imperial&appid=APIKEY)

**zip=90210&units=imperial&appid=APIKEY**

```
{
 "coord":{
 "lon":-118.4,
 "lat":34.07
 },
 "weather":[
 {
 "id":800,
 "main":"Clear",
 "description":"clear sky",
 "icon":"01d"
 }
],
 "base":"cmc stations",
 "main":{
 "temp":75.61,
 "pressure":1017,
 "humidity":14,
 "temp_min":60.8,
 "temp_max":82.4
 },
 "wind":{
 "speed":3.36
 },
 "clouds":{
 "all":1
 },
 "id":5328041,
 "name":"Beverly Hills",
 "cod":200
}
```

City's geo location

Overall description

Degrees Fahrenheit

Percent humidity

Minimum and maximum temps at the moment

Miles per hour

Percent cloudy

City

## Try 11.14.1: Try OpenWeatherMap's API in your web browser.

1. Go to [openweathermap.org](https://openweathermap.org).
2. Sign up for an account.
3. Try the link: <http://api.openweathermap.org/data/2.5/weather?zip=90210&units=imperial&appid=APIKEY> to make an API request for the weather with ZIP 90210. An error message indicating an invalid API key was used should be returned.
4. Replace APIKEY in the URL's query string with your API key, and reload the webpage. The JSON-encoded weather information for 90210 should be displayed.
5. Change the ZIP code in the URL's query string to your ZIP code, and reload the URL.

### PARTICIPATION ACTIVITY

#### 11.14.2: The Weather API.

- 1) What request method does the Weather API expect in an API request for the current weather data?
  - ☐ POST
  - ☐ PUT
  - ☐ GET
- 2) What "units" parameter value would make the Weather API return the temperature in Celsius?
  - ☐ imperial
  - ☐ metric
  - ☐ standard

3) Does the Weather API support finding the current weather by city name?

- ☐ Yes
- ☐ No

## Calling the Weather API from the web server

The Weather API may be called from an Express web server using the `node-fetch` module. The **node-fetch module** imitates `window.fetch`, which implements the Fetch API to make HTTP requests. The `node-fetch` module is installed using: `npm install node-fetch`.

The animation below uses the `node-fetch` module to request the weather for ZIP 90210. The **URLSearchParams** class is used to create query string parameters for the web API's URL. The **fetch()** method sends an HTTP request to the specified URL and returns a Promise object. The Promise object's `then()` method is passed arrow functions that convert the JSON response into JavaScript objects and then pass the JavaScript objects to Pug for rendering a webpage.

For the animation code to work properly in an Express server, the "APIKEY" string must be replaced with an actual API key.

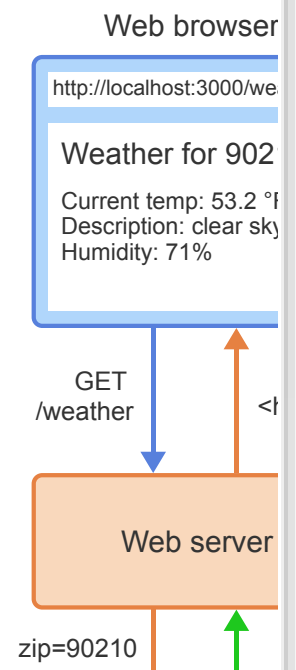
### PARTICIPATION ACTIVITY

11.14.3: Calling the Weather API from an Express server with the `node-fetch` module.

```
const fetch = require("node-fetch");

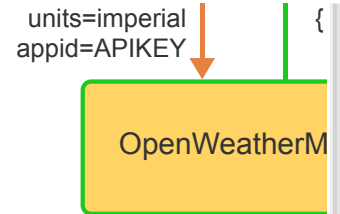
app.get("/weather", function(req, res) {
 const zip = 90210;
 const params = new URLSearchParams({
 zip: zip,
 units: "imperial",
 appid: "APIKEY"});

 fetch("http://api.openweathermap.org/data/2.5/weather?" + params)
 .then(response => response.json())
 .then(data => {
 const locals = {
 data: data,
 zip: zip
 };
 res.render("weather", locals);
 })
 .catch(error => console.log(error))
});
```



```
title Weather
body
 h2 Weather for #{zip}
 p Current temp:
 | #{data.main.temp} °F
 p Description:
 | #{data.weather[0].description}
 p Humidity:
 | #{data.main.humidity}%
```

weather.pug



## Animation content:

A JavaScript block is shown:

```
const fetch = require("node-fetch");
app.get("/weather", function(req, res) {
 const zip = 90210;
 const params = new URLSearchParams({
 zip: zip,
 units: "imperial",
 appid: "APIKEY"});
 fetch("http://api.openweathermap.org/data/2.5/weather?" + params)
 .then(response => response.json())
 .then(data => {
 const locals = {
 data: data,
 zip: zip
 };
 res.render("weather", locals);
 })
 .catch(error => console.log(error))
 });
```

A block of Pug called weather.pug is also shown:

```
title Weather
body
 h2 Weather for #{zip}
 p Current temp:
 | #{data.main.temp} °F
 p Description:
 | #{data.weather[0].description}
```

p Humidity:

| #{data.main.humidity}%

A web browser with the URL `http://localhost:3000/weather` sends a GET request to the web server for `/weather`. The web server sends a request to the OpenWeatherMap API with the following parameters:

`zip=90210`

`units=imperial`

`appid=APIKEY`

The API sends back a JSON encoded file with the weather for the zip 90210 to the web server.

The web server sends the web browser an html file that displays the following on the screen:

Weather for 90210

Current temp: 53.2 °F

Description: clear sky

Humidity: 71%

### Animation captions:

1. Web browser requests `/weather` from Express web server, which triggers the server's GET route.
2. The `URLSearchParams` object creates a query string with the given parameters and values.
3. Web server uses the `node-fetch` module to generate a GET request for the current weather:  
`http://api.openweathermap.org/data/2.5/weather?`  
`zip=90210&units=imperial&appid=APIKEY`
4. OpenWeatherMap responds with JSON containing current weather for ZIP code 90210.
5. The `response.json()` method converts the JSON response into JavaScript objects.
6. Pug renders the webpage with weather data, and the page is sent to the web browser.

#### PARTICIPATION ACTIVITY

11.14.4: Calling the Weather API from the web server.



Refer to the animation above.

- 1) When is the first `then()` arrow function executed?
- ☐ Before the API request is sent.
  - ☐ Immediately after the API request is sent.
  - ☐ After the API response is received.
- 2) When is the second `then()` arrow function executed?
- ☐ Before the API request is sent.
  - ☐ While the first `then()` arrow function executes.
  - ☐ Immediately after the first `then()` arrow function completes.
- 3) What variable would the Pug code above display in order to show the current wind speed?
- ☐ `data.main.temp`
  - ☐ `data.wind.speed`
  - ☐ `data.main.speed`

Exploring further:

- [Public APIs](#) on GitHub.com

## 11.15 Token-based user authentication (Node)

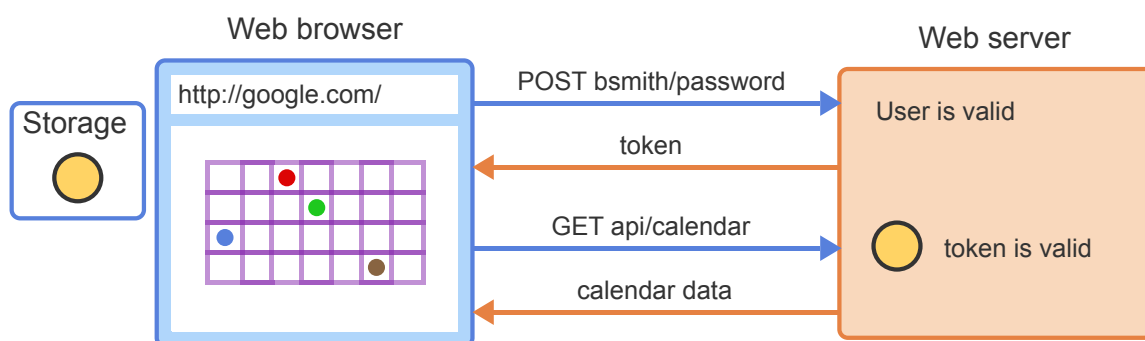
### Overview

**User authentication** is the process of verifying that a user is who the user claims to be. Many websites authenticate users based on a user provided username and password. Some websites supplement passwords with security questions, security images, and other techniques. Using usernames and passwords to authenticate users is certainly not foolproof, but the technique is currently the most common form of user authentication.

To implement user authentication, most websites use token-based authentication. **Token-based authentication** is a technique where the client uses a signed token to "prove" to a server that the client has successfully authenticated. A **signed token** is a string of characters produced with a secret key that uniquely identifies the entity that created the token.

#### PARTICIPATION ACTIVITY

#### 11.15.1: Token-based authentication process.



#### Animation content:

A web browser is shown with a username and password field. When the user types in the username bsmith and password, a POST request with bsmith/password is sent to the web server for authentication. Once validated, the web server sends back a token. The web app request bsmith's calendar with GET api/calendar and the token is sent back to the web server for validation. Once the token is found valid, bsmith's calendar information is sent back to the web browser and displayed on the screen.

#### Animation captions:

1. User authenticates by entering a username and password, which are sent to the web server.



2. Web server verifies the username and password are valid and generates a token.
3. The token is returned to the web browser and stored for future requests.
4. Web app requests bsmith's calendar, so the token is sent in the web API request.
5. Web server validates bsmith's token and returns back bsmith's calendar data.

**PARTICIPATION  
ACTIVITY**

## 11.15.2: Steps in token-based authentication.



Match each request or response with the action the request or response generates.

If unable to drag and drop, refresh the page.

Server generates a token.

Client sends the token with an API request.

Client sends username and password for authentication.

Server responds with data when given a valid token.

	Server verifies the username and password.
	Client stores the token for future requests.
	Server validates the token.
	Client receives API response.

Reset

## JSON Web Tokens (JWT)

JSON Web Tokens are a popular way of implementing tokens in token-based authentication. A **JSON Web Token (JWT)**, pronounced "jot", is a string produced by the server that encodes JSON

data. The string is signed with a secret key known only by the server to ensure the data in a JWT is unaltered by the client. JWTs are composed of three parts separated by periods:

**Header.Payload.Signature.**

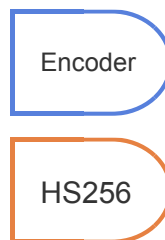
JSON data is base64 encoded in a JWT. **Base64 encoding** is a technique that converts data into 64 printable characters.

**PARTICIPATION  
ACTIVITY**

11.15.3: Creating a JWT.



Header:     {"typ": "JWT", "alg": "HS256"}  
Payload:    {"username": "bob"}  
Secret Key:  supersecret



Header:     eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.  
Payload:    eyJ1c2VybmFtZSI6ImJvYiJ9.  
Signature:  Mm0fNOZMBFOrFu99NlnHdz3jkg5IE\_BQCNOz4sh1epQ

## Animation content:

A header, payload, and secret key contain the following before being encoded:

Header: {"typ": "JWT", "alg": "HS256"}

Payload: {"username": "bob"}

Secret Key: supersecret

The header and payload are encoded with Base64 and stored in the token as the following:

Header: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.

Payload: eyJ1c2VybmFtZSI6ImJvYiJ9

A signature is stored in the token and reads:

Signature: Mm0fNOZMBFOrFu99NlnHdz3jkg5IE\_BQCNOz4sh1epQ

## Animation captions:

1. The Header contains the type of token (JWT) and the algorithm used to sign the token (HMAC SHA256).
2. The Header is Base64 encoded.
3. The Payload containing the information to transmit is Base64 encoded.
4. The Signature is created using the HMAC SHA256 algorithm on the base64-encoded Header + base64-encoded Payload + Secret Key.

When the client sends a JWT back to the server, the server validates the JWT by generating a new signature based on the JWT's base64 encoded header and payload and the server's secret key. If the payload was not modified, the new signature should be identical to the JWT's signature.

### PARTICIPATION ACTIVITY

#### 11.15.4: JSON Web Tokens.



- 1) JWTs almost always have the same header.  
☐ True  
☐ False
- 2) JWTs almost always have the same payload.  
☐ True  
☐ False
- 3) A JWT's payload can only include the user's username.  
☐ True  
☐ False
- 4) If the client changes one character of the JWT, the JWT will fail validation on the server.  
☐ True  
☐ False



5) If a client knows the server's secret key, the client can create JWTs with any payload that the server will accept as valid JWTs.

- ☐ True
- ☐ False

## jwt-simple module

The **jwt-simple module** is used to encode and decode JWTs in Node.js web applications. The jwt-simple module is installed using: `npm install jwt-simple`. jwt-simple provides an **jwt.encode()** method to create a JWT and a **jwt.decode()** method to decode a JWT.

Figure 11.15.1: Encoding and decoding a JWT with the jwt-simple module.

```
const jwt = require("jwt-simple");
const secret = "supersecret";
const payload = { username: "bsmith" };

// Create a JWT
const token = jwt.encode(payload, secret);
console.log("Token: " + token);

// Decode a JWT
const decoded = jwt.decode(token, secret);
console.log("Decoded payload: " + decoded.username);
```

```
Token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImJvYiJ9.Mm0fNOZMBFOrFu99NlnHdz3jkgp5IE_BQC
Decoded payload: bsmith
```

The figure below uses jwt-simple in the context of an Express server with two routes:

1. **/api/auth** - Accepts a username and password in a POST request. After verifying the username is "bsmith" and password is "pass", the route sends back a JWT carrying a payload containing bsmith's username.
2. **/api/status** - Accepts a GET request and expects the X-Auth header to contain the JWT obtained from the **/api/auth** route. After validating the JWT, the route sends back bsmith's

status.

Developers can use a tool like Postman to test the Express server with the example HTTP requests shown at the bottom of the figure below.

Figure 11.15.2: Using jwt-simple in an Express server.

```
const express = require("express");
const bodyParser = require("body-parser");
const jwt = require("jwt-simple");

const app = express();
const router = express.Router();

// Parse application/x-www-form-urlencoded
router.use(bodyParser.urlencoded(
 { extended: false }));

// Secret used to encode/decode JWTs
const secret = "supersecret";

router.post("/auth", function(req, res) {

 // Verify bsmith/pass was POSTed
 if (req.body.username === "bsmith" && req.body.password === "pass") {
 // Send back a token that contains the user's username
 const token = jwt.encode({username: "bsmith"}, secret);
 res.json({ token: token });
 }
 else {
 // Unauthorized access
 res.status(401).json({ error: "Bad username/password" });
 }
});

router.get("/status", function(req, res) {

 // Check if the X-Auth header is set
 if (!req.headers["x-auth"]) {
 return res.status(401).json({error: "Missing X-Auth header"});
 }

 // X-Auth should contain the token value
 const token = req.headers["x-auth"];
 try {
 const decoded = jwt.decode(token, secret);

 // Send back a status
 if (decoded.username === "bsmith") {
 res.json({status: "Enjoying a beautiful day!"});
 }
 }
});
```

```
 }
 else {
 res.json({status: "Working hard!"});
 }
 }
 catch (ex) {
 res.status(401).json({ error: "Invalid JWT" });
 }
});

app.use("/api", router);

app.listen(3000);
```

HTTP request	HTTP response
POST /api/auth HTTP/1.1 Host: localhost:3000 User-Agent: Mozilla/5.0 Chrome/48.0.2564 Content-Type: application/x-www-form-urlencoded  username=bsmith&password=pass	HTTP/1.1 200 OK Content-Type: application/json; charset=utf-8 Content-Length: 121 Date: Tue, 03 May 2016 15:22:17 GMT ETag: W/"79-d6Dq2n+D6aIQoSfkTMks8w" X-Powered-By: Express  { "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1HmPCsMMWPZvVflRwxkJV3o1Gk0xsUfaCI7wmleC0"}

HTTP request	HTTP response
GET /api/status HTTP/1.1 Host: localhost:3000 User-Agent: Mozilla/5.0 Chrome/48.0.2564 Content-Type: application/x-www-form-urlencoded X-Auth: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJlc2VybmFtZSI6ImJzbWl0aCJ9.yuHqmPCsMMWPZvVflRwxkJV3o1Gk0xsUfaCI7wmleC0	HTTP/1.1 200 OK Content-Type: application/json; charset=utf-8 Content-Length: 121 Date: Tue, 03 May 2016 15:22:17 GMT ETag: W/"26-MFB4mdGAWP1" X-Powered-By: Express  { "status": "Working hard today!" }



Refer to the code in the figure above.

1) What status code is returned when a POST to `/api/auth` contains the username "bmsith" and the password "opensesame"?

- ☐ 200
- ☐ 401
- ☐ 404

2) What status code is returned when a GET to `/api/status` does not contain an X-Auth header?

- ☐ 200
- ☐ 401
- ☐ 404

3) Can the Express server above return a token that would later cause the "Working hard!" status to be returned by `/api/status`?

- ☐ Yes
- ☐ No

## Storing the secret key.

*The examples in this section show the secret key used to sign the JWT in the source code. Good practice is to store the secret key in a configuration file and use a long randomized string for the secret key. Ex: qs3h6z0JUN9wgTy1j2Cl54gB6yzG is a good secret key.*

## Using a database

Authentication services generally store and retrieve data from a database. The figure below shows

a Node.js project that stores data in a MongoDB database using a Mongoose model called `User`. The `User` model stores a username, password, and status for a user. The Express server has three routes:

1. `/api/user` - Accepts a username, password, and status in a POST request. The route adds a new user to the database.
2. `/api/auth` - Accepts a username and password in a POST request. The route verifies the POSTed username/password matches a username/password from the database and returns a JWT with the username in the payload.
3. `/api/status` - Accepts a GET request and expects the X-Auth header to contain a valid JWT. The route validates the JWT and sends back the username and status of all users in the database.

Figure 11.15.3: Node.js project uses token-based authentication with user data in a MongoDB database.

```
// api/users.js
const jwt = require("jwt-simple");
const User =
 require("../models/user");
const router =
 require("express").Router();

// For encoding/decoding JWT
const secret =
 "supersecret";

// Add a new user to the
// database
router.post("/user",
 function(req, res) {

 if (!req.body.username ||
 !req.body.password) {
 res.status(400).json({
 error: "Missing username
 and/or password"});
 return;
 }

 const newUser = new
 User({
 username:
 req.body.username
```



```

myproject
├── api
│ └── users.js
├── models
│ └── user.js
├── node_modules
│ ├── express
│ ├── mongoose
│ ├── jwt-simple
│ └── etc...
├── db.js
├── package.json
└── server.js

```

```

// models/user.js
const db = require("../db");

// Create a model from the schema
const User = db.model("User", {
 username: { type: String, required: true },
 password: { type: String, required: true },
 status: String
});

module.exports = User;

```

```

// db.js
const mongoose = require("mongoose");
mongoose.connect("mongodb://localhost/mydb");
module.exports = mongoose;

```

```

req.body.username,
 password:
req.body.password,
 status:
req.body.status
 });

 newUser.save(function(err) {
 if (err) {

res.status(400).send(err);
 }
 else {

res.sendStatus(201); //
Created
 }
 });
});

// Sends a token when given
valid username/password
router.post("/auth",
function(req, res) {

 if (!req.body.username ||
!req.body.password) {
 res.status(401).json({
error: "Missing username
and/or password"});
 return;
 }

 // Get user from the
database
 User.findOne({ username:
req.body.username },
function(err, user) {
 if (err) {

res.status(400).send(err);
 }
 else if (!user) {
 // Username not in
the database

res.status(401).json({
error: "Bad username"});
 }
 else {
 // Check if
password from database
matches given password

```

```
// server.js
const express = require("express");
const bodyParser = require("body-parser");
const User = require("./models/user");

const app = express();
const router = express.Router();
router.use(bodyParser.urlencoded(
 { extended: false }));
router.use("/api", require("./api/users"));
app.use(router);

app.listen(3000);
```

```
 if (user.password
 != req.body.password) {
 res.status(401).json({
 error: "Bad password"});
 }
 else {
 // Send back a
 token that contains the
 user's username
 const token =
 jwt.encode({ username:
 user.username }, secret);
 res.json({
 token: token });
 }
 });
});

// Gets the status of all
users when given a valid
token
router.get("/status",
function(req, res) {

 // See if the X-Auth
 header is set
 if (!req.headers["x-
 auth"]) {
 return
 res.status(401).json({error:
 "Missing X-Auth header"});
 }

 // X-Auth should contain
 the token
 const token =
 req.headers["x-auth"];
 try {
 const decoded =
 jwt.decode(token, secret);

 // Send back all
 username and status fields
 User.find({},
 "username status",
 function(err, users) {
 res.json(users);
 });
 }
 catch (ex) {
 res.status(401).json({
 error: "Invalid JWT" });
 }
```

```
 }
 });

module.exports = router;
```

**PARTICIPATION  
ACTIVITY**

## 11.15.6: Token-based authentication with a MongoDB database.

Refer to the code in the figure above.

1) What does the HTTP request below do?

```
POST /api/user HTTP/1.1
Host: localhost:3000
Content-Type: application/x-www-form-urlencoded

username=jwhite&password=qwerty&status=Working+hard!
```

- ☐ Authenticates user jwhite
- ☐ Gets all user statuses
- ☐ Creates a new user

2) If user "jwhite" with password "qwerty" is in the database, what response does the Express server return to the request below?

```
POST /api/auth HTTP/1.1
Host: localhost:3000
Content-Type: application/x-www-form-urlencoded

username=jwhite&password=opensesame
```

- ☐ 200 status code with a JWT
- ☐ 401 status code with an error message
- ☐ 201 status code

3) What response does the Express server return to the request below?



```
GET /api/status HTTP/1.1
Host: localhost:3000
X-Auth: jwhite
```

- ☐ 200 status code with all user statuses
- ☐ 401 status code with an error message
- ☐ 201 status code

## Saving passwords in the database

*The examples above show passwords stored as plain text in a database. Passwords should NEVER be stored as plain text in a database. Instead, password hashes (discussed elsewhere) should be stored in the database.*

## Storing JWT in localStorage

The web application running in the web browser must interact with the server's API to authenticate the user, save the JWT, and transmit the JWT in subsequent API requests. The `window.localStorage` object allows web applications to store data in the web browser and is ideal for storing a JWT. The example below uses the Fetch API to obtain and transmit a JWT.

### PARTICIPATION ACTIVITY

11.15.7: Authenticating with Fetch and storing the JWT in localStorage.



```
async function login(username, password) {
 const response = await fetch("/api/auth", {
 method: "POST",
 body: new URLSearchParams({
 username: username,
 password: password
 })
 });
}
```

### web browser

http://localhost:3000/app

Status:

- bsmith - Enjoying a beautiful day!
- jwhite - Working hard!
- ablack - Doing homework

local  
Storage  
token:3a

```

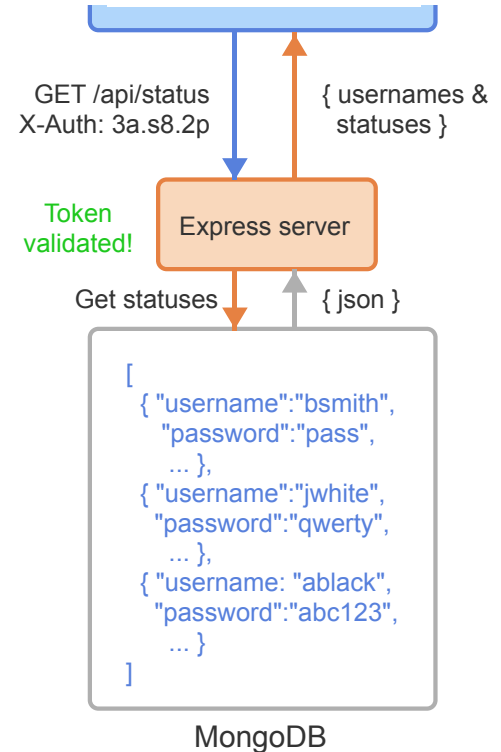
 if (response.ok) {
 const tokenResponse = await response.json();
 localStorage.setItem("token",
 tokenResponse.token);
 }
 }

 async function displayStatus() {
 const token = localStorage.getItem("token");

 const response = await fetch("/api/status", {
 headers: { "X-Auth": token }
 });

 if (response.ok) {
 const users = await response.json();
 let html = "";
 for (let user of users) {
 html += "" + user.username + " - " +
 user.status + "";
 }
 const status = document.querySelector("ul");
 status.innerHTML = html;
 }
 }
}

```



## Animation content:

A block of JavaScript is shown:

```

async function login(username, password) {
 const response = await fetch("/api/auth", {
 method: "POST",
 body: new URLSearchParams({
 username: username,
 password: password })
 });
 if (response.ok) {
 const tokenResponse = await response.json();
 localStorage.setItem("token",
 tokenResponse.token);
 }
}

async function displayStatus() {
 const token = localStorage.getItem("token");
 const response = await fetch("/api/status", {
 headers: { "X-Auth": token }
 });
}

```

```
});
if (response.ok) {
 const users = await response.json();
 let html = "";
 for (let user of users) {
 html += "" + user.username + " - " +
 user.status + "";
 }
 const status = document.querySelector("ul");
 status.innerHTML = html;
}
}
```

A web browser is shown with a username and password field. The user types in the bsmith username and password. The following POST request is sent to the Express server:

```
POST /api/auth
username=bsmith
password=pass
```

The web server verifies the login information with a MongoDB database. The Express server sends a JWT token back to the web browser with the following information:

```
{ "token":
 "3a.s8.2p" }
```

The web browser stores the token in local storage. To request status the web browser sends the following GET request to the web server:

```
GET /api/status
X-Auth: 3a.s8.2p
```

The web server validates the token and retrieves the data to be displayed on the web browser, which is the following information:

Status:

- bsmith - Enjoying a beautiful day!
- jwhite - Working hard!
- ablack - Doing homework

## Animation captions:

1. The user enters a username and password and clicks Login button.
2. To authenticate the user, the browser calls login() with the username and password. The fetch() method sends a POST request with username and password to /api/auth.
3. The server verifies bsmith/pass and returns a JWT to the web browser.
4. response.json() extracts the token from the JSON response. The setItem() method saves

the token to localStorage.

5. To display user statuses, `displayStatus()` retrieves the token from localStorage with `getItem()`. Then `fetch()` sends a GET request to `/api/status` with the token in the X-Auth header.
6. Token is validated by the server, so server sends JSON with all user statuses to the browser.
7. `response.json()` parses the user data. The usernames and statuses display in an unordered list.

#### PARTICIPATION ACTIVITY

#### 11.15.8: Storing JWT in localStorage.



Refer to the code in the animation above.

- 1) localStorage does not store a JWT until after the user authenticates.



- ☐ True  
☐ False

- 2) The JWT is not stored in localStorage if the user provides the wrong username or password when authenticating.



- ☐ True  
☐ False

- 3) If the user provides the wrong username or password when authenticating, `displayStatus()` can still display the user statuses.



- ☐ True  
☐ False

- 4) If the code below executes, `displayStatus()` can no longer obtain the user statuses.



```
localStorage.removeItem("token");
```

- ☐ True
- ☐ False

**PARTICIPATION  
ACTIVITY**

11.15.9: User authentication with Fetch.



Press the Display Status button before typing a username/password to see an "X-Auth header missing" error message. The Display Status button calls the JavaScript function `displayStatus()`. If the user is not logged in, `displayStatus()` sends a GET request to `https://wp.zybooks.com/status.php?op=auth` with an empty token, which returns back an error message.

Enter the username "bsmith" and password "pass" and press the Login button. Then press the Display Status button again. The status of three users will display. Only bsmith/pass is an acceptable username/password.

If the user enters a bad username or password, the webpage does not indicate so. Add the necessary code in `login()` to display an appropriate error message when the server returns a 401 response.

Note that JavaScript does not use `localStorage` due to browser restrictions. Also, the URLs in the Fetch calls differ from previous examples.

HTML

CSS

JavaScript



```
1 <form id="login" autocomplete="off">
2 <p>
3 <label for="username">Username:</label>
4 <input type="text" id="username">
5 </p>
6 <p>
7 <label for="password">Password:</label>
8 <input type="text" id="password">
9 </p>
10 <button id="loginBtn" type="button">Login</button>
11 <p id="errorMsg"></p>
12 </form>
13
14 <button id="statusBtn">Display Status</button>
15
16
```

[Render webpage](#)[Reset code](#)

### Your webpage

Username:

Password:

[► View solution](#)

## Third-party authentication

Many web applications and mobile apps allow users to login to the web/mobile app using the user's account on Google, Facebook, Microsoft, etc., using an authentication protocol called **OpenID**. OpenID allows a user to login to the application without creating another username/password. The application does not have to manage the user's password or provide methods for the user to reset their password.

**OAuth** is an authorization protocol that allows a web/mobile app to access services from a service provider like Google, Facebook, Microsoft, etc., on behalf of the authenticated user. Ex: Words with Friends is a popular mobile game that uses a user's Facebook friends to help the user find friends to play with.

Exploring further:

- [Authentication](#) (TechTarget)
- [jwt.io](#) - Encode/decode JWTs online
- [node-jwt-simple](#)
- [OpenID](#) and [OAuth](#)

## 11.16 Password hashing (Node)

### Cryptographic hash functions

Websites must exercise care when storing personal information about users. Numerous high-profile data breaches confirm that protecting personal data is very difficult to do. To protect users, organizations should store as little sensitive information as possible.

Many individuals are interested in discovering the passwords used by a website's users. Therefore, user passwords should never be stored as plain text in a database. Instead, developers should use a cryptographic hash function to convert passwords into hashes and store only the hashes in the

database. A **cryptographic hash function** is a mathematical algorithm that converts text of any length into a fixed-length sequence of characters called the **hash digest** or "hash". Various cryptographic hash functions exist including MD5, SHA-1, bcrypt, and PBKDF2.

Figure 11.16.1: Hashing various passwords using the MD5 cryptographic hash function.

Input	Hash Function	Hash Digest
password	MD5	5f4dcc3b5aa765d61d8327deb882cf99
pass123	MD5	32250170a0dca92d53ec9624f336ca24
pass123!	MD5	c9d9b8cab32214716ee1b44b3aae2502
Pass123!	MD5	10487c8581423e8b2fbeed2b21c2cc53

Try 11.16.1: Try generating your own MD5 hashes.

1. Go to an online [MD5 Hash Generator](#).
2. Type "password" into the text box and verify the MD5 hash is the same as the "password" hash digest in the figure above.
3. Try changing the password by a single character, and notice that the hash completely changes.

**PARTICIPATION  
ACTIVITY**

11.16.1: Cryptographic hash functions.

- 1) The length of the hash digest is the same regardless of how many characters are in the password.

- ☐ True
- ☐ False

2) All cryptographic hash functions produce the same size hash digest.

- ☐ True  
☐ False

3) Given a hash digest, a clever hacker might be able to determine the original input that created the hash digest.

- ☐ True  
☐ False

4) Two different inputs may exist that will convert into the same hash digest.

- ☐ True  
☐ False

## Hashed passwords and password cracking

An authentication system should only store password hashes, not plain text passwords. To authenticate a user, the authentication system hashes the password submitted by the user and compares the hash with the hash stored in the database. If the two hashes are identical, then the user provided the correct password.

### PARTICIPATION ACTIVITY

11.16.2: Verifying passwords by comparing password hashes.

New user: pgreen  
              opensesame

cryptographic  
hash function

Database	
username	passhash
bsmith	F993GH93F1
jwhite	83B8F2EE48
ablack	49A8EF48D3
pgreen	BC029G8A38

Login attempt: pgreen  
OpenSesame



75EE39FA90 != BC029G8A38

Login attempt: pgreen  
opensesame



BC029G8A38 = BC029G8A38

## Animation content:

A database is shown with column names username and passhash. The following data is in the format username, passhash:

bsmith, F993GH93F1

jwhite, 83B8F2EE48

ablack, 49A8EF48D3

A new user is added with the username pgreen and password opensesame. The password is sent through a cryptic hash function and is stored in the table as the following:

pgreen, BC029G8A38

When the user tries to login the tried password is sent through the cryptic hash function and checked against the passhash stored in the table. If pgreen types in OpenSesame, the hash value is 75EE39FA90 which does not equal BC029G8A38, the passhash that is stored in the table, so pgreen is denied access.

## Animation captions:

1. New user's username and password hash is added to the database.
2. User pgreen attempts to login with the wrong password. Hash of "OpenSesame" does not match hash of "opensesame".
3. User pgreen attempts to login with the correct password. Hash of "opensesame" matches hash of "opensesame".

If a data breach occurs and an attacker obtains an organization's database of password hashes, the attacker would be unable to directly convert the hashes back into the users' original passwords. However, a determined attacker may use a variety of methods to crack the password hashes.

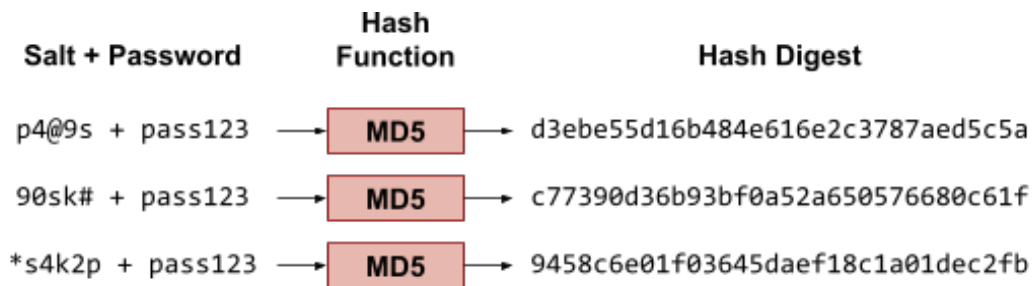
**Password cracking** is the process of recovering passwords from data, like the database of a compromised website.

A **dictionary attack** is a popular password-cracking strategy where the attacker feeds a number of possible passwords, such as words from a dictionary, into a hash function and compares the

stolen hashes to the generated hashes. Passwords are revealed for any matching hashes. Dictionary attacks are computationally expensive, so attackers often create precomputed dictionary tables in advance.

Developers use salt to circumvent attacks with precomputed tables. A **salt** is a random string that is combined with a password so two identical passwords produce different hashes. Cracking salted passwords is significantly more challenging because an attacker has to create a table for each salt value, which is computationally expensive.

Figure 11.16.2: Using salt to protect against precomputed table attacks.



Salt does not necessarily keep attackers from cracking passwords, but strong passwords are less susceptible to dictionary attacks. A **strong password** is a password that is difficult to guess. A strong password has the following characteristics:

- Does not contain words found in a dictionary or on the web
- Composed of uppercase and lowercase letters, digits, and perhaps punctuation
- At least 10 characters in length
- Has not been used as a password before on the same website or on any other website
- Does not conform to popular password patterns like an initial capital letter, 2-4 digits at the end, or adding ! at the end

Figure 11.16.3: Secure password check indicates 'qwerty123!' is a poor password.

## How secure is your password?

**Tip:** Try to make your passwords at least 15 characters long

Show password: ☒

**qwerty123!**

**Very Weak**

10 characters containing: ☒ Lower case ☒ Upper case ☒ Numbers ☒ Symbols

Time to crack your password:

**0.01 seconds**

**Review:** Oh dear, using that password is like leaving your front door wide open. Your password is very weak because it contains a common password, a sequence of characters and a dictionary word.

Your passwords are never stored. Even if they were, we have no idea who you are!

Source: [my1login.com](https://my1login.com)

### PARTICIPATION ACTIVITY

#### 11.16.3: Password hashes.

1) If an authentication system hashes a user's submitted password and finds the hash is the same as the user's password hash in the database, then the user supplied the \_\_\_\_\_ password.

- ☐ correct
- ☐ wrong
- ☐ hashed

2) Someone who steals a database of password hashes may attempt a \_\_\_\_\_ attack to discover some of the passwords.

- ☐ hash
- ☐ dictionary
- ☐ salt

3) Is the salt stored with the passwords in the database?

- ☐ Yes
- ☐ No

4) Which password is a strong password?

- ☐ PurpleRain
- ☐ qwertyuiop
- ☐ fiBPs03!5a

5) Can a website that stores password hashes in their database recover lost passwords for their users?

- ☐ Yes
- ☐ No

## 2012 LinkedIn data breach

*Russian cybercriminals stole 6.5 million password hashes from LinkedIn on June 5, 2012. The criminals cracked more than 60% of the unique passwords from the hashes and published the passwords in plain text on the web. LinkedIn had used the SHA1 hash function to generate the hashes but did not use salt. The data breach highlights the risk to users who do not use strong passwords.*

Source: [2012 LinkedIn hack \(Wikipedia\)](#).



## bcryptjs

Cryptography researchers have found weaknesses in older hash functions like MD5 and SHA-1 that make those hash functions vulnerable to attacks. *Good practice is to use a strong hash function like bcrypt, scrypt, or PBKDF2, which purposely execute slowly and make password cracking significantly more difficult.*

The **bcryptjs module** is a popular module for implementing authentication in Node.js applications. bcryptjs is installed using: `npm install bcryptjs`. The module has two commonly used methods:

1. The **bcrypt.hashSync()** method returns a hash value that is 60 characters long for a given password and "cost factor". The hash value is created by prepending a random salt to the hash digest.
2. The **bcrypt.compareSync()** method takes a password and bcrypt hash value as arguments and returns true if the password is identical to the password used to create the hash value, false otherwise.

An example bcrypt hash of

\$2a\$10\$33M.4Zn7R7k3jHOISHxCe.yUql4vl9mv4/oeiLuhHDQZcfySVg6wC is composed of four parts:

1. "2a" indicates the bcrypt algorithm created the hash.
2. "10" indicates the cost factor is 10.
3. "33M.4Zn7R7k3jHOISHxCe." is the 16-byte salt value encoded to 22 characters.
4. "yUql4vl9mv4/oeiLuhHDQZcfySVg6wC" is the 24-byte hash digest encoded to 31 characters.

Figure 11.16.4: Generating a hash and comparing hashes with the bcryptjs module.

```
const bcrypt = require("bcryptjs");

// Generate 3 hashes for the same password
for (let c = 0; c < 3; c++) {
 let hash = bcrypt.hashSync("opensesame", 10);
 console.log("Hash: " + hash);
}

// Compare hash produced by identical passwords
const passwordHash = bcrypt.hashSync("opensesame", 10);
if (bcrypt.compareSync("opensesame", passwordHash)) {
 console.log("Same!");
}
else {
 console.log("Not the same");
}
```

```
Hash:
$2a$10$33M.4Zn7R7k3jHOISHxCe.yUqI4v19mv4/oeiLuhHDQZcfySVg6wC
Hash:
$2a$10$ByrTcHizkol25k1WOnlV2egFI3DdRzo9.xrjal/IyiaKd8X5diFpi
Hash:
$2a$10$cd.MX7Ubbm3n9Lfc8ilrgeYw3MRpJ512fUtPq4kHEQyhI8wpJ8LPq
Same!
```

**PARTICIPATION  
ACTIVITY**

11.16.4: Password hashing with bcryptjs.

1) Does `bcrypt.hashSync()` always generate a unique hash for the same password and cost factor?

- ☐ Yes
- ☐ No

2) The larger the cost factor passed to `bcrypt.hashSync()`, the \_\_\_\_ time `bcrypt.hashSync()` takes to produce a hash value.

- ☐ less
- ☐ more

3) What does the code below output to the console?

```
const hash =
bcrypt.hashSync("qwerty", 10);
if
(bcrypt.compareSync("Qwerty",
hash)) {
 console.log("Same!");
}
else {
 console.log("Not the
same");
}
```

- ☐ Same!
- ☐ Not the same

## Using a database

The Node.js project in the figure below creates a web API that stores usernames, password hashes, and statuses in a MongoDB database.

- The `/api/user` route accepts a username, password, and status in a POST request. The `bcrypt.hashSync()` method generates a password hash, which is saved with the username and status to the database.
- The `/api/auth` route accepts a username and password in a POST request. The `bcrypt.compareSync()` method compares the password hash from the provided password with the password hash in the database. If the password is correct, the route returns a JWT.
- The `/api/status` route accepts a GET request and expects the X-Auth header to contain a valid JWT. The route returns a JSON-encoded list of usernames and statuses for all users in the database.

Figure 11.16.5: Node.js project uses token-based authentication and password hashing with bcryptjs.

```
// api/users.js
const jwt = require("jwt-simple")
const User =
require("../models/user");
const router =
require("express").Router();
const bcrypt = require("bcryptjs")

// For encoding/decoding JWT
const secret = "supersecret";

// Add a new user to the database
router.post("/user", function(req, res) {

 if (!req.body.username ||
 !req.body.password) {
 res.status(400).json({ error:
 "Missing username and/or
password" });
 return;
 }

 // Create a hash for the submitted
password
 const hash =
bcrypt.hashSync(req.body.password,
10);

 const newUser = new User({
 username: req.body.username,
 passwordHash: hash,
 status: req.body.status
 });

 newUser.save(function(err) {
 if (err) {
 res.status(400).send(err);
 }
 else {
 res.sendStatus(201); //
Created
 }
 });
});

// Sends a token when given valid
```

```

myproject
├── api
│ └── users.js
├── models
│ └── user.js
├── node_modules
│ ├── bcryptjs
│ ├── express
│ ├── mongoose
│ ├── jwt-simple
│ └── etc...
├── db.js
├── package.json
└── server.js

```

```

// models/user.js
const db = require("../db");

// Create a model from the schema
const User = db.model("User", {
 username: { type: String, required:
true },
 passwordHash: { type: String, required:
true },
 status: String
});

module.exports = User;

```

```

// db.js
const mongoose = require("mongoose");
mongoose.connect("mongodb://localhost/mydb");
module.exports = mongoose;

```

```

username/password
router.post("/auth", function(req,
res) {

 if (!req.body.username ||
!req.body.password) {
 res.status(401).json({ error:
"Missing username and/or
password" });
 return;
 }

 // Get user from the database
 User.findOne({ username:
req.body.username }, function(err,
user) {
 if (err) {
 res.status(400).send(err);
 }
 else if (!user) {
 // Username not in the
database
 res.status(401).json({ error:
"Bad username" });
 }
 else {
 // Does given password
match the database password hash
 if
(bcrypt.compareSync(req.body.passwordHash,
user.passwordHash)) {
 // Send back a token
contains the user's username
 const token = jwt.encode({
username: user.username }, secret);
 res.json({ token: token });
 }
 else {
 res.status(401).json({
error: "Bad password" });
 }
 }
 });
});

// Gets the status of all users
given a valid token
router.get("/status", function(req,
res) {

 // Check if the X-Auth header
set
 if (!req.headers["x-auth"]) {

```

```
// server.js
const express = require("express");
const bodyParser = require("body-parser");
const User = require("../models/user");

const app = express();
const router = express.Router();
router.use(bodyParser.urlencoded(
 { extended: false }));
router.use("/api", require("../api/users"));
app.use(router);

app.listen(3000);
```

```
 return
 res.status(401).json({error: "Missing X-Auth header"});
 }

 // X-Auth should contain the token
 const token = req.headers["x-auth"];
 try {
 const decoded =
 jwt.decode(token, secret);

 // Send back all username and status fields
 User.find({}, "username status");
 function(err, users) {
 res.json(users);
 }
 }
 catch (ex) {
 res.status(401).json({ error: "Invalid JWT" });
 }
});

module.exports = router;
```