CS144: Web Applications

Professor Cho

Thilan Tran

Spring 2021

Contents

CS144: Web Applications **3**

Web Standards **4**
HTTP . . . 4
HTTP/2 . . . 5
Content Encodings . . . 5
HTML . . . 7
CSS . . . 9
Cookies and Sessions . . . 10

Dynamic User Interaction **12**
Dynamic Web Server . . . 12
AJAX . . . 14

JavaScript **15**
Various Notes . . . 15
Document Object Model . . . 16
Advanced JavaScript . . . 19
Browser Event Handling . . . 22
Async Programming and Promises . . . 23
Promises . . . 24
Async and Await . . . 25
TypeScript . . . 26

MEAN Stack **29**
MongoDB . . . 29
Node.js . . . 30

Express 31

Single Page Applications **35**

Client Side Frameworks **38**

Angular 38

Reactive Programming **41**

CSS **43**

Scaling **46**

CS144: Web Applications

- covered topics:
  - core web standards eg. HTTP, Unicode, HTML, JSON, CSS
  - JavaScript programming
  - web programming paradigms eg. functional and asynchronous programming, MVC
  - website architecture, scalability, and security

Web Standards

- core Internet standards:
  - **domain name service (DNS)** maps domain names to IPs and details how to reach particular IPs in a hierarchical design
    - ICANN manages **top-level domains (TLDs)**
  - **transmission control protocol and Internet protocol (TCP/IP)** is the main Internet routing and transportation protocol
  - **hypertext transportation protocol (HTTP)** is the communication protocol between web servers and clients
  - encoding standards such as ASCII or Unicode text
    - multimedia types such as JPEG, MP3, H.264, etc.
  - **hypertext markup language (HTML)** is the markup standard
  - **cascading style sheets (CSS)** is the styling and formating standard
  - **JavaScript** is the defacto web programming language
- the early Web was mainly designed to retrieve static content eg. HTML pages and images from servers:
  - can be set up with an HTTP server eg. Apache and filesystem
  - need a URL path to file mapping
  - the **uniform resource locator (URL)** is a unique ID for any object on the web:
    - eg. `<protocol>://<hostname>/<path>?<query>#<fragment_id>`
  - the **fragment identifier** is the string behind the hash in the URL
    - points to the HTML element with the given ID
  - the **query** is a set of name-value pairs
- four general layers of a site:
  - storage / data layer stores and retrieves data
  - application layer store and retrieves data
  - HTTP layer interprets request and serves response
  - encryption layer encrypts transport

HTTP

- HTTPS/2 is the most recent version from 2015:
  - but HTTP/1.1 is still extremely popular from 1996
  - major browsers support HTTP/2 only over HTTPS
- two key properties of HTTP:
  1. request and response paradigm
     - all interactions start with a client's request, to which a server can reply

2. HTTP is a stateless protocol:
    - every request is handled independently of others
    - server is not required to remember history of past requests

- any HTTP message is either a request or a response with the following structure:
    - request / status line eg. `GET / HTTP/1.0` or `HTTP/1.1 200 OK` :
        * primary HTTP methods are `GET, POST, PUT, DELETE`
            · other methods include `HEAD, OPTIONS, TRACE`
        * by convention, GET methods should leave no significant side effects at the server
        * status codes fall into the categories
            · `2xx` success, `3xx` redirection, `4xx` client error, `5xx` server error
    - header lines
    - empty line
    - body

HTTP/2

- motivations:
    - many high-latency mobile devices with limited bandwidth
    - many objects are needed to display a single page:
        * eg. HTML, images, CSS, JavaScript, etc.
        * ie. 100s of HTTP requests may be needed to render a page
- HTTP/2 relaxes key assumptions for speed and efficiency:
    - server can now push messages
    - add stateful elements
    - binary encoding
- new features in HTTP/2:
    - multiplexed streams:
        * multiple outstanding requests through a single connection
        * split messages into small streams
        * priority specification
    - HPACK
        * stateful HTTP header compression
    - server push
        * predictively cache responses pushed by the server

Content Encodings

- only bits are transmitted over the Internet, so it is essential to include the `Content-Type` header in the HTTP response

  - eg. `Content-Type: text/html`
  - the **multi-purpose Internet extension (MIME)** type is a standard way to indicate the type of transmission:
    * formatted as `type/subtype`, case insensitive
    * eg. `text/html, text/plain, image/jpeg, video/mp4, application/pdf`
  - the character encoding is specified as the `charset` parameter
    * eg. `Content-Type: text/html; charset=utf-8`

- for text, how does a browser map a sequence of bits to characters?

- ASCII standard from 1963:

  - 7-bit, represented 128 characters
  - extended to many 8-bit standards eg. ISO-8859-1
  - basis of current standards for roman characters

- EBCDIC standard from 1963:

  - created by IBM for IBM mainframes
  - 8-bit, designed to be easy to represent in punch cards
  - still used by some IBM mainframes

- local character codes developed by each country:

  - the **double byte code character set (DBCS)** used two bytes for a character
  - frequently used in Asia eg. Chinese GB2312, Korean EUC-KR

- how does a computer know what encoding standard is used for a certain file?

  - early solution was a system-wide specification in a global **code page** ie. unique number for a particular character encoding

- **Unicode** was a single standard for all existing characters in the world:

  - motivated by need to standardize all the various character encodings
  - v1.0 was published in October 1991
    * almost yearly release of a new Unicode version
  - every character maps to a unique **code point**
    * eg. `A` corresponds to `U+0041`
  - originally a fixed-length 16-bit standard AKA UCS-2, now currently 21-bit standard
  - Unicode initially had an issue with little vs. big endian storage
    * use the Unicode byte order mark `U+FEFF` at the beginning of a Unicode string in order to give hints on the endian mode
  - an initial issue was many incompatibilities with legacy code

- need to make Unicode backward compatible with ASCII:

  - Unicode-aware programs would work with ASCII data, and legacy code would work with basic Unicode data
  - this is **UTF-8**
  - both UTF-8 and ASCII encoding should map all ASCII characters to the *same* 1-byte number:
    * eg. `A` with code point `U+0041` should be encoded as `41`
    * to accomplish this, UTF-8 needs to be a *variable* length encoding
  - all characters between `U+0000` and `U+007F` is encoded in a single byte
  - all characters between `U+0080` and `U+07FF` is encoded in 2 bytes
    * the initial prefixes of each byte are fixed as `110, 10` respectively
  - all characters between `U+0800` and `U+FFFF` is encoded in 3 bytes
    * the initial prefixes of each byte are fixed as `1110, 10, 10` respectively
  - all characters between `U+10000` and `U+10FFFF` is encoded in 4 bytes
    * the initial prefixes of each byte are fixed as `11110, 10, 10, 10` respectively
  - this allows all existing ASCII-encoded data to be UTF-8 encoded
  - UTF-16 is an extension of UCS-2 to cover 21 bit code points

HTML

- how does the browser extract the rich structure of a page from a pure text file?

- **hypertext markup language (HTML)** is the document standard of the web:

  - specifies both the content and the structure of a Web page
  - made up of text ie. content together with **tags** enclosed in `<...>` that represent the structure
    * eg. `Fat <strong>cats</strong> go down <em>alleys<em/>.`
  - history:
    * initial HTML1 version introduced in 1991
    * all the way up to HTML5 in 2014
      · standardized by WHATWG group, was competing with XML

- HTML document always starts with `<!DOCTYPE html>` :

  - earlier versions use different `DOCTYPE`
  - remnants from **standard generalized markup language (SGML)**
  - followed by `html, head, title, body` elements

Minimum HTML5 document:

```html
<!DOCTYPE html>
<html>
<head>
    <title>...</title>
</head>
<body>...</body>
</html>
```

- an HTML **element** is a single HTML entity enclosed in an opening and closing tag:
  - eg. `<p>paragraph</p>`
  - an open tag is always followed by a closing tag except **void elements**
    * eg. `br, hr, img, input`
  - tags in HTML5 should represent structure, not formatting
    * HTML5 also added additional semantic elements such as `header, nav, article, section`
  - instead, CSS should be used to specify formatting
- special characters:
  - multiple white spaces and line breaks are always displayed as a single white space
    * ` ` can be used for non-breaking space
  - `< > &amps;` for `< > &`
  - `<!-- ... -->` for comments
- HTML tags can have **attributes**:
  - eg. `<img src="...">`
  - the **ID attribute** should be unique in a document
    * acts as an unique identifier of an element, like a key
- embedding objects:
  - an `a` element with an `href` attribute can be used to embed links:
    * can use relative or absolute URL
    * eg. `<a href="{url}">...</a>`
    * we can embed another HTML page inside a page with an `iframe` element
  - images, videos, audio, and others can be embedded with the `img, video, audio, embed` elements, respectively
  - the **favorite icon (Favicon)** is displayed next to the title
    * default path is `/favicon.ico`
- more in HTML5:
  - clearly defined logic to handle errors for improperly defined documents
  - programmable JavaScript API
    * eg. canvas, web storage, offline web applications, drag-and-drop, document editing, etc.

- XHTML is mostly the same as HTML, but with must stricter formatting rules:
  - tags and attributes *must* be lowercase
  - no more empty elements, all tags have matching end tags
  - always use quotes around attribute values
  - ie. makes HTML XML-compliant
  - failed to become popular because it was too strict without much benefit

CSS

- **cascading style sheets (CSS)** is used to specify document formatting and presentation:
  - each CSS **rule** is a selector together with a declaration block
  - each declaration is a name-value pair
- can use a `style` tag within the HTML document, or link to a separate CSS file
  - `div, span` are structure-less tags used to format particular parts of the document
- CSS selectors:
  - `[src]` selects for attribute `src`
  - `[target="_blank"]` selects for attribute `target` with value `_blank`
  - `div, p` applies to multiple tags
  - `div p` applies to `p` that is a descendant of `div`
  - `div > p` applies to `p` that is a direct child of `div`
  - `p.class1.class2` applies to `p` belonging to both specified classes
  - `div + p` applies to immediate adjacent sibling `p` of `div`
  - `div ~ p` applies to any sibling `p` of `div`
  - `:hover` is a **pseudo class** selector
    - class created by the browser
  - `::first-letter` is a **pseudo element** selector
    - elements created by the browser

Example CSS:

```css
h1 {
    font-family: "Arial";
    font-size: 40pt;
}

.code {
    font-family: monospace;
    white-space: pre;
    border: 1px solid black;
```

```
}

#warning1 {
    color: rgb(255, 0, 0);
}
```

Cookies and Sessions

- HTTP is a stateless protocol where every request can be handled independently of others:
  - how does a website remember a user and customize its behavior?
  - how does a website detect two requests are from the same user?
  - idea behind cookies is to embed a unique identifier in every request from a user
- **cookies** allow a server to ask a client to remember `key=value` pairs and send them back in all future requests:
  - the `Set-Cookie` HTTP response header tells the client to remember a cookie
  - eg. `Set-Cookie: username=john; expires=...;`
  - `expire` is the expiration time:
    - by default cookie becomes *transient* and is sent back only during the current browsing session
    - setting `expire` makes the cookie *persistent* until expiration
  - `path` and `domain` specify which requests to send the cookie in
  - the **same origin policy** is where the client sends the cookie only to the domain from which it was obtained
  - but it is still possibly to track a user's requests across multiple domains through **third-party cookies**:
    - a website will partner with partner sites in order to track the same user
    - embed a tiny, invisible image that is requested when a user visits a partner site
    - this requeset contains a `Referrer` header with cookies from the partner site
- cookie security:
  - cookies are generally unsafe since they can be stolen or tampered with
    - need to be careful about what we store in the cookie
  - the `secure;` attribute only sends the cookie back over HTTPS
    - protects against cookie theft
  - we can use signed cookies to prevent tampering

    * add a secret-key encrypted signature to the main cookie data
  - attaching an expiration date ensure the cookie expires eventually
    * even if cookie is stolen, it will eventually become invalid
- **JSON web tokens (JWTs)** are a web standard to encode and excange client-managed state with tampering protection:
  - format is `header.payload.signature`
  - header holds information on the token
  - payload holds the main body of the token
  - signature is an encrypted has value for tampering detection
  - the JWT header holds JSON data encoded into a `Base64` string:
    * typically has two fields, the `alg` hashing algoritm and `typ` token type (JWT)
    * eg. `{"alg": "HS259", "typ": "JWT"}`
  - the payload is also encoded JSON data:
    * may include **registered claims** ie. standardized fields such as `iss` issuer, `jti` JWT ID, `iat` issued at, `exp` expires at
    * can put anything else in the JSON, ie. as an **unregistered field**
  - the signature is a secret-key encrypted has of encoded `header.payload`:
    * if the JWT is ever tampered, a correct signature cannot be generated without knowing the correct password
  - JWT is sent to the browser
    * for future requests, the JWT will be sent back
- how does a server authenticate the identity of a user?
  - ask for a password
  - how can we let a user authenticate with their password once, without asking for authentication for every request?
  - use a cookie to keep track of username, or track **sessions** through a session ID
    * using a session allows for more flexibility, but forces the server to maintain state on the server side

Dynamic User Interaction

- many sites generate content dynamically based on user input
  - eg. keyword search, social media status update, etc.
  - how can the server obtain user input?
    - one way is to use the query string of the URL
    - another approach is to use HTML forms

Example HTML form:

```
<form action="http://www.google.com/search" method="GET">
    <input type="text" name="q">
    <input type="submit">
</form>
```

- need to specify in the form *where* to send the input, and what method to use:
  - default action is current directory
  - the `input` element has different possible types and a name specifying the query name
    - user provided input is sent as query name-value pairs of request
  - in a GET request, the query is added to the request path
    - in a POST request, the query is added to the request body
  - the `submit` button indicates that the user has completed input
  - different input types include text, password, checkbox, radio, select, submit, button, textarea, hidden, file
    - hidden field can be used to store state, ie. bypass HTTP stateless restrictions

Form for uploading a file:

```
<form action="..." method="POST" enctype="multipart/form-data">
    <input type="text" name="name">
    <input type="file" name="myfile">
    <input type="submit">
</form>
```

- `multiparte/form-data` is a way to include multiple objects in a single message
  - the `boundary` attribute of the content type header specifies the object boundaries

Dynamic Web Server

- how can we write code to generate dynamic content at the server?
  - two general approaches, programmatic vs. template
    - ie. write a program vs. write a Web page

Example programmatic approach with Java Servlet:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException
{
    PrintWriter out = response.getWriter();
    out.println("<html><head><title>Hello</title></head>");
    out.println("<body>Hello, " + request.getParameter("fname"));
    out.println("</body>");
    out.println("</html>");
    out.close();
}
```

Example template approach with Java ServerPages:

```
<html>
<head><title>Hello</title></head>
<body>Hello, <%= request.getParameter("first_name") %>
</body>
</html>
```

- even though the template approach seems cleaner, the page will quickly get messy as complex application logic is added:
  - can we separate the code from the page?
  - ie. enforce code *ownership* where page design is done by designers, while app coding is done by developers
    - we want to make each of these aspects as independent as possible
- in a **model-view-controller (MVC)**, we split the overall program into several parts:
  - data, application logic, and final result presentation
  - for the **data** layer:
    - data may be stored in a file or database engine, locally or remotely
    - where and how data is stored and managed may change over time
    - should be encapsulated in a layer independent from other layers
  - for the **presentation** layer:
    - same data may be presented in many different ways
    - presentation changes should not affect other layers
  - thus, split the code into three modular components:
    1. the **models** deal with data storage and access
    2. the **views** deal with result presentation

3. the **controllers** deal with application logic
   * each component may be owned by different people

MVC example in Java Servlet:

```
// Controller
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException
{
    user = getUser(...); // retrieve data
    ... // application logic here
    request.setAttribute("user_name", user.name); // dispatch data model to view
    request.getRequestDispatcher("/index.jsp").forward(request, response);
}

// Model
User getUser(String userid) {...}

// View
<html>
<head><title>Hello</title></head>
<body>Hello, <%= request.getAttribute("user_name") %></body>
</html>
```

AJAX

- in traditional website interactions:
  - all input is form-based
  - must press submit button and wait until the entire page *reloads*
  - leads to constant interruptions and significant delay
- **asynchronous JavaScript and XML (AJAX)** allows for:
  - immediate, in-place update of page content
  - allows for a user experience that is more similar to a desktop application
- new browser responsibilities:
  - AJAX is *event-driven*, where control flow is driven by **events**
    * need to allow **callback functions** that map events to actions
- AJAX building blocks:
  1. JavaScript is *the* programming language for the Web
     - allows running complex code inside a browser
  2. **document object model (DOM)**:
     - tree-based model of HTML document

  - JavaScript manipulates DOM to dynamically change page
  - JavaScript monitors events on the DOM and takes actions
3. asynchronous communication mechanism with the server
   - eg. `fetch, XMLHttpRequest`

JavaScript

- originally created as a simple script to manipulate Web pages:
  - NodeJS (JavaScript interpreter) runs almost everywhere
  - supported by most modern browsers
  - allows running arbitrary code inside the browser
  - current standard is much more complex than originally intended
- basic syntax is very similar to C-style languages

Various Notes

- `==, !=` check if operands have the same value after type conversion:
  - while `===, !==` check if operands have the same value and type
  - for equality on object, both check if operands reference the same object
- dynamically rather than statically typed
- `typeof` usually returns the current type of the variable, but not always
  - types are either primitive or object type
  - primitive types include `number, string, boolean`
    * as well as `bigint, symbol, null, undefined`
- note that all numbers are represented as a 64-bit floating point number:
  - no integer numbers in JS
  - bitwise operators convert a number to a 32-bit integer
    * truncate subdecimal digits if needed
  - `NaN, Infinity` are valid numbers
- `bigint` is a 64-bit integer that was added to ES2020 as a primitive type:
  - add `n` behind number type
  - note that this is *not* a number type
- booleans:
  - falsy values include `0, NaN, "", null, undefined`
  - everything else is truthy, including all arrays and objects
- strings are immutable
  - string manipulation creates a new string
- undefined and null types:
  - undefined indicates an uninitialized ie. default value *before* initialization

    - null indicates absence of an object
        * an object is expected but nothing can be returned
    - however, `typeof null` returns object
- objects:
    - object assignment is by reference
    - object comparison is by reference
    - arrays are objects
- `RegExp` is a special regular expression object in JS:
    - enclosed inside slashes, eg. `/a?b*c/`
    - can be used with string methods `search, match, replace, split`
        * as well as regex methods `exec, test`
- exception handling in JS:
    - use `try, catch, finally` blocks
    - we can `throw` any value or object
        * but typically only `Error` objects are thrown because it provides a stack trace `Error.stack`
- **JavaScript object notation (JSON)** is the standard syntax to represent literal objects in JS:
    - object property names and strings *require* double quotes
    - useful methods `JSON.stringify, JSON.parse`
    - JSON values cannot be functions or undefined
        * circular references cannot be stringified either

Document Object Model

- the **document object model (DOM)** is a standard to construct JavaScript objects from an HTML document:
    - allows JavaScript to interact with the webpage and manipulate elements
    - the HTML document is converted to a tree-like model
    - the `script` HTML tag is used to embed JavaScript code or link to a separate file
- three key node types:
    1. an **element node** represents an HTML element
        - every HTML tag creates an element node
    2. a **text node** is all text enclosed in an element
        - text node becomes a child of the element node
    3. an **attribute node** is the attribute of an element
        - is *associated* with its element node, but is *not* a child

Example HTML with DOM Conversion in Figure 1:

```html
<!DOCTYPE html>
<html>
<head><title>Page Title</title></head>
<body>
    <h1>Heading</h1>
    <a href="good/">Link</a>
</body>
</html>
```

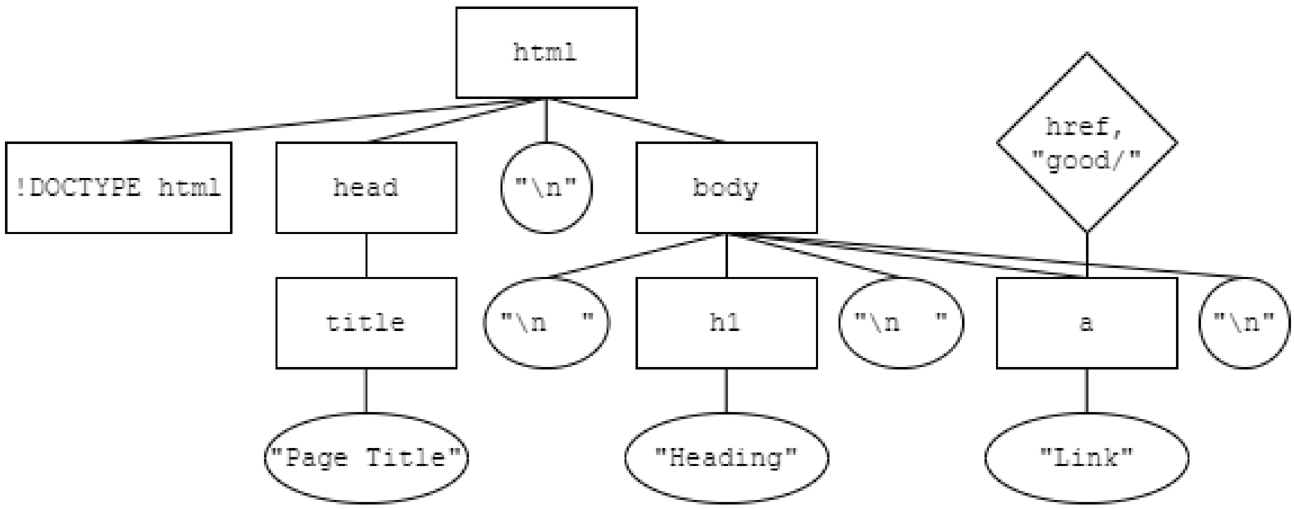

Figure 1: DOM Conversion Example

- note that white spaces are preserved from the header through the body:
  - allows the DOM representation to always be converted back into the exact HTML code
  - need to create additional text nodes for white spaces
- every DOM node becomes a JavaScript object:
  - with properties, methods, and associated events
    - by changing these property values, calling the methods, etc. we can update the HTML element dynamically
  - the `document` object is the root object that has the parsed DOM tree as its child
  - DOM object properties for the tree structure:
    - `childNodes` are the node's children
    - `parentNode` is the node's parent
    - `attributes` are the node's attributes
    - `nodeType` is one of `"Element", "Attribute", "Text"`
    - `nodeName` is the tag name eg. `HEAD`, attribute name eg. `href`, or `#text` for text nodes
    - `nodeValue` is the enclosed text for text nodes, the attribute value for attribute nodes, and `null` otherwise
- accessing DOM nodes:

1. can traverse the tree directly using the `childNodes` property starting from the `document`
2. get nodes directly with `getElementById, getElementsByClassName, querySelectorAll`

DOM manipulation examples:

```
document.body.style.background = "yellow"; // body is a shortcut property
document.getElementById('warning1').style.color = "red";
document.body.innerHTML = "<p>new text </p>"; // innerHTML is parsed into a DOM tree
                                              // and replaces the original children
let newP = document.createElement("p");
let newText = document.createTextNode("new text");
newP.appendChild(newText);
document.body.replaceChild(newP);
```

- manipulating the DOM:
  - methods include `createElement, createTextNode`
    - as well as `appendCHild, removeChild, replaceChild`
  - specific objects may have certain methods to take certain actions
    - eg. `reset, submit` on a form element
- event-driven programming:
  - to dynamically update a web page based on user action, JavaScript must handle certain invoked events
  1. wait for relevant events
  2. take appropriate actions given an event
  - common DOM **events** include `load, click, input, mouseover` :
    - an object has an associated **event handler** for each event
    - this function is invoked when the event is triggered
      - can customize its action by setting the event handler
    - when an event is fired AKA triggered on an object AKA event target, the associated callback function AKA event handler is called AKA invoked

Event handler example:

```
let colors = ["yellow", "blue", "red"];
let i = 0;

function changeColor(event) {
    document.body.style.color = colors[i++%3];
}

document.body.addEventListenenr("click", changeColor);
// alternatively, <body onclick="changeColor();"> (not recommended)
```

Advanced JavaScript

- variables scope:
  - variables declared outside any block has global scope
  - a variable declared with `let` has block-scope local variable
  - a variable used without an explicit `let` declaration has global scope
    - strongly discouraged
  - before `let`, `var` was used:
    - `var` has function scope as opposed to block scope
    - `var` declarations are *hoisted* (but *not* the assignment)
    - should always use `let` declarations

Scope example:

```
let a = "a"; // global
b = "a";     // global
function f() {
    c = "c";     // global
    let d = "d"; // local
}

var e = 10; // global
function g() {
    f = 15; // local!
    var f;
}
```

- in JS, functions are objects:
  - functions can be assigned to a variable, passed as a parameter, and can have properties
  - however, `typeof` operator returns `function`, not `object`
  - functions can also be nested:
    - local variables in nested functions follow *lexical* scope rather than dynamic scope
    - uses **closures** to hold references to variables declared lexically outside of the function
      - ie. a function bundled together with references to its enclosing state
    - closures were used extensively before ES6

Using objects as functions:

```
function square(x) { return x*x; }
function foo(x, fn) { return fn(x); }
```

```
foo(10, square); // returns 100
foo(10, function(x) { return x*2; }) // anonymous function, returns 20
foo.a = 20;
```

Nested functions:

```
function f() {
    let a = 1;
    let b = 2;
    function g() {
        console.log(b); // prints 2
        b = 3; // updates the previous b to 3
    }
    if (a > 0) {
        let b = 4;
        g();
        console.log(b); // prints 4
    }
    console.log(b);     // prints 3
}
f();
```

Closure example:

```
let age = 21;

function getFunc() {
    let age = 10;
    function printAge() { console.log(age++); }
    return printAge;
}

let myFunc = getFunc();
myFunc(); // prints 10
myFunc(); // prints 11
```

- the JS **arrow function** is a quick way to create an anonymous function in ES6:
  - in JS, we often have to pass a function as a parameter, so polluting the namespace can be simplified using anonymous functions
  - arrow function makes this even more concise
  - `() => expr` returns the value of the expression
  - in `() => { statements; }`, need to explicitly return a value
  - arrow functions retain the `this` binding of the enclosing lexical context
    - ie. *inherits* its `this` binding

- **object-oriented programming (OOP)** in JavaScript:
  - using objects that wrap together data and methods
  - in JS, we can add a method to an object:
    - inside an object's method, `this` points to the object itself
    - note that arrow functions should not be used for object / class method definitions due to its lexical binding properties
  - original syntax for creating objects is `new Object()` :
    - in ES2015, can use the typical `class` syntax with constructors, etc. as seen in other languages
    - ES2015 also added class inheritance support using `extends, super`
    - internally, class inheritance is implemented via a `prototype` object
- meaning of `this` in JavaScript:
  - becomes bound to different things depending on the context
  1. in a function called via an object or class method, `this` is bound to the called object or class
  2. in a function called via event triggering, `this` is bound to the DOM element in which the event handler was set
  3. everywhere else (in top-level block or other function calls), `this` is bound to the global object:
     - `window` in the browser or `global` in Node.js (now referable by the `globalThis` keyword in ES2020)
     - any variable assigned without declaration becomes a property of the global object

Arrow function lexical binding:

```
x = 10;
function_printx = function() { console.log(this.x); };
arrow_printx = () => { console.log(this.x); };

o = { x: 20 };
o.printx_f = function_printx;
o.printx_a = arrow_printx;

console.log(this.x); // prints 10
function_printx();   // prints 10
arrow_printx();      // prints 10
o.printx_f();        // prints 20
o.printx_a();        // prints 10
```

- array manipulation:
  - **mutators** modify the input array directly
    - `reverse, sort, push, pop, shift, unshift, splice`
  - **accessors** leave the input array intact:

    * `concat, slice, filter, map`
    * creates and returns a new output array

Browser Event Handling

- the `event` object is passed as the only argument to the event handler function:
  - `event.target` is the target to which the event was originally triggered (similar to `this`)
  - we can set our own event handler to catch any DOM event:
    * the *original* event handler is also invoked after our custom event handler
    * to prevent the original handler, we need to call `event.preventDefault()`
  - eg. `onclick="alert('Clicked!'); event.preventDefault()"` inside an `a` link
    * note that if an event handler is set using statements inside an HTML tag, they are wrapped into a function with the single input parameter `event`
  - `event.type` is the event type
- most DOM events *bubble up* through the DOM tree:
  - target's ancestors get the event all the way through the `document` and sometimes `window` object
    * exceptions include `focus, scroll`
  - to stop event propagation, call `event.stopPropagation()` inside event handler
- inside the browser, JS code is executed in a *single* thread:
  - thus no two event handlers will ever run at the same time
  - document contents are never update simultaneously
    * no concerns about locks or deadlock
  - but the web browser stops responding to user input while the handlers are running
    * need to break into parts or use web workers
- JavaScript execution timeline in the browser:
  1. `document` object is created and `document.readyState` is set to `loading`
  2. browser downloads and parses the page:
     - scripts are downloaded and executed synchronously in the order they appear in the page
     - unless the script is `async` , in which they are downloaded asynchronously in the background and is executed as they are available
  3. once the page is completely parsed, `document.readyState` is set to `interactive`

    4. then the browser fires the `DOMContentLoaded` event and calls the `document.onload` callback
    5. `document.readyState` is set to `complete`
    6. browser waits for events and calls appropriate event handlers
    - use `onload, onunload` handlers for initialization and cleanup code
- the `window` object is the global object within a browser:
    - `document` is really `window.document`
    - `window.location` is the URL of the current page
    - `window.history` gives the browsing history
    - `window.alert, confirm, prompt`

Async Programming and Promises

- in synchronous APIs, operations block on every step:
    - program becomes stuck at every step
    - how can the program handle many requests concurrently despite long, blocking waits?
    - traditionally, multithreading is used for multiple request processing:
        * invoke multiple handlers in parallel
        * *no change* in coding style
            · structure of each synchronous request handler remains the same
        * used by most traditional servers, eg. Apache, Tomcat
        * however, multithreading incurs a significant resource overhead:
            · high memory use
            · thread invocation overhead
            · concurrency handling logic eg. semaphores and locks
- existing JS engines are still *single threaded*:
    - cannot use multi-threading
    - instead, an asynchronous API is used for multiprocessing under the single threaded environment:
        * do not wait, and return immediately
        * invoke a callback function when ready
    - eg. `db.find({userid: id}, callback)` :
        * `db.find` returns immediately
        * the `callback` is invoked when the database object is ready
        * retrieved object is passed as a parameter to `callback`
        * only the `callback` can perform actions with the object
            · ie. the next line in the code does not have the required object
    - actions may be spread across multiple callback functions, leading to "callback hell"

Tradition synchronous programming example:

```
function sendPicture(id) {
    user = db.find({userid: id});
    picture = fs.readFile(user.picFile);
    socket.write(picture);
    console.log('done');
}
```

Illustrating callback hell:

```
function sendPicture(id) {
    db.find({userid: id}, (err, user) => {
        fs.readFile(user.picFile, (err, picture) => {
            socket.write(picture, () => console.log('done'));
        });
    });
}
```

Promises

- **promises** were introduced in ECMAScript 2015:
  - an asynchronous function *immediately* returns a "promise"
  - once a promise is obtained, a callback can be attached using `then`
    - a resolve callback runs on a success called with the return value of the operation
    - a reject callback runs on failure called with the error value
  - `then` returns a new promise:
    - can set a callback to the returned promise, creating a **promise chain** of asynchronous callbacks
    - makes the code look and work more like a synchronous program
    - each asynchronous function is non-blocking and return immediately
- details of promises:
  - a promise is **settled** by being **resolved** or **rejected**
  - promise operation depends on what value is returned by a callback:
    - if a regular value is returned by a callback, the chained promise is resolved
      - if a promise is returned by a callback, the chained promise returns either a value or error of that settled promise depending if it is resolved or rejected
    - if an error is thrown at some point in the callback, the chained promise is rejected
- what if a call fails, but the rejection callback is not set?

  - setting one rejection callback at the end is enough
    * no need to set a rejection callback in every `then`
  - `catch(rejectCB)` is shorthand for `then(null, rejectCB)`
- promise guarantees:
  - callbacks added with `then` even *after* the success or failure of the asynchronous operatio *will* be called:
    * ie. setting the `then` occurs after the asynchronous function completes
    * this was a possible problem we had to consider with callbacks
  - callbacks will never be called before the completion of the current run of the JS event loop
  - these guarantees motivated the name "promise"
- creating a promise:
  - `Promise.resolve(val)` creates a promise that always resolves to `val`
  - `Promise.reject(err)` creates a promise that always rejects to `err`
  - `new Promise((res, rej))` creates a dynamic promise

Creating a dynamic promise:

```
let p = new Promise((resolve, reject) => {
    ...
    if (cond) resolve(val);
    else reject(err);
});
```

Async and Await

- `async/await` keywords are syntactic sugar to make asynchronous look almost like synchronous code:
  - `await` can be used on any function that returns a promise inside an `async` function
  - adding `async` to a function changes the function to one that returns a promise immediately, without blocking:
    * ie. "promisifies" the function
    * if the original function returns a regular value, the returnd promise resolves to the value
    * if the original function throws an error, the returned promise is rejected to the error
    * if the original function returns a promise, the previous chaining logic applies
  - the `await` keyword can be used in front of a function that returns a promise:
    * "blocks" the code in that location until it settles

    * the next action is performed after the promise is settled
    * can only be used inside an `async` function
  - if we want to use `await` in the outer most block, we can use an IIFE

Using `async/await` :

```
async function sendPicture(id) {
    try {
        user = await db.find({userid: id});
        picture = await fs.readFile(user.picFile);
        await socket.write(picture);
        return picture.size;
    } catch (e) {
        throw new Error("Cannot send picture");
    }
}
```

Parallel `await` gotchas:

```
function doubleAfter2(x) {
    return new Promise((res, rej) => setTimeout(res, 2000, x*2));
}

async function addAsync1(x) {
    return await doubleAfter2(x)
        + await doubleAfter2(x)
        + await doubleAfter2(x);
}

async function addAsync2(x) {
    const a = doubleAfter2(x);
    const b = doubleAfter2(x);
    const c = doubleAfter2(x);
    return await a + await b + await c;
}

addAsync1(10).then(v => console.log(v)); // prints 60 after 6 seconds
addAsync2(10).then(v => console.log(v)); // prints 60 after *2* seconds
```

TypeScript

- **TypeScript** is a superset of JavaScript:
  - additional features include types, interfaces, decorators, etc.

  - all additional TS features are strictly optional and not required
  - thus any JS code is also a TS code
  - TS must be complied to JS using a `tsc` , the TypeScript compiler
- type annotations can be added to functions and variables:
  - allows for static type checking
  - makes large-scale code easier to manage
  - compile-time error vs. run-time error
    * rigidity vs. flexibility
- type annotations:
  - eg. `number, string, boolean`
    * three basic type values cannot be assigned to a different type variable
  - arrays and tuples eg. `number[], [number, string]`
  - unions eg. `number | string`
  - objects eg. `{x: number, y: string}`
    * object of type `A` can be assigned to a variable of type `B` if their structure is compatible ie. the properties of `A` should be a superset of `B` 's
  - also `any, void, never`
  - `undefined` and `null` can be assigned to any types

TS "hello, world" example with types:

```
function hello(greeting: string): string {
    return 'Hello, ' + greeting + '!';
}

const world: string = 'world';
hello(world);
hello([0, 1, 2]); // generates an error during compilation
```

TS object type compatibility:

```
interface Point2D {
    x: number;
    y: number;
};

function plot(p: Point2D): void { ... }
let point3D = { x: 1, y: 2, z: 3 };
plot(point3D); // no error
```

- type conversion:
  - primitives eg. `Number('1'), String(2), Boolean('true')`

  - objects eg. `<HTMLInputElement>document.querySelector('input[type="text"]')` :
    * `as` and `<>` are equivalent
    * `HTMLInputElement` is a subclass of `HTMLElement`
- functions:
  - in JS, missing parameters are OK and are bound to `undefined`
  - in TS, all function parameters must be passed
    * an optional parameter is indicated by the suffix `?`
- classes:
  - explicit member property declaration
    * in JS, had to use a constructor to declare fields
  - adds access modifiers `public, private, protected`
  - interfaces like the Java interface
  - generic classes and functions
    * promises can be generic types as well

TS generics:

```
class Dot<t> {
    public x: T;
    constructor(x: T) { this.x = x; }
};
let s = new Dot<number>(1);

function log<T>(arg: T): void {
    console.log(arg);
}
log<number>(1);
```

- decorators:
  - syntax `@decorator_name`
  - can be added to certain class or method declarations
    * can modify various aspect of declared entities
  - eg. `@sealed` seals objects so that property values may change but the structure is fixed

MEAN Stack

- traditional web development:
  - stack includes:
    - Apache / Nginx for HTTP
    - PHP or Servlet for the server runtime
    - MySQL for data storage
  - almost all code runs on the server
  - browser is a passive HTML / CSS rendering engine
  - after AJAX, most code runs in the browser as JS:
    - server transformed into a back-end service that provides data persistence and transaction support
    - leads to better user experience and less load on the server
  - challenges with AJAX development:
    - increasing complexity in the JS code
    - impedance mismatch of JS on the client and PHP or Java on the server
      - JSON for data transport vs. relational data for data storage
- modern full stack web development:
  - Angular / React / Vue for the client runtime
  - Node.js and its packages for the server runtime
  - MongoDB data engine
  - MEAN AKA MongoDB, Express.js, Angular, and Node.js

MongoDB

- database for JSON objects:
  - a NoSQL database with no predefined schema
  - no normalization or joins
  - other libraries eg. `Mongoose` can be used for ensuring structure in the data
- data in MongoDB is stored as a collection of documents:
  - a **document** is a JSON object
  - a **collection** is a group of similar documents
- document vs. relational databases:
  - a relational model flattens the data:
    - stores as a set of independent tables
    - removes redundancy
    - table is designed around the intrinsic nature of the data
    - efficient join algorithms

  - a document model preserves the view of a particular application:
    * hierarchically nested objects
    * potential redundancy
    * no need to decompose data for storage and join back for retrievals
    * retrieving data with a different view is difficult
- CRUD operations:
  - `insertOne, insertMany`
  - `findOne, find`
  - `updateOne, updateMany`
  - `deleteOne, deleteMany`

Example MongoDB operations:

```
db.books.insertOne({title: "MongoDB", likes: 100});

db.books.find({$and: [{likes: {$gte: 10}}, {likes: {$lt: 20}}]});

db.books.updateOne({title: "MongoDB"}, {$set: {title: "MongoDB II"}});

db.books.deleteMany({likes: {$lt: 100}});
```

- administrative commands:
  - `show dbs` shows list of databases
  - `use <dbname>` uses a specific database
  - `db.dropDatabase()` deletes the current database
  - `show collections` shows list of collections
  - `db.createCollection(<cname>)` creates a collection
  - `db.cname.drop()` drops a collection
  - `db.cname.createIndex({title:1, likes:-1})` creates an index on combined attributes
    * ascending or descending order

Node.js

- **Node.js** is a JavaScript runtime environment based on the Chrome V8 JavaScript engine:
  - allows JS to run on any computer
  - intended to run directly on ON, instead of inside a browser:
    * removes browser-specific JS API like the HTML DOM
    * adds support for OS APIs such as file system and network
- notably, Node.js is *single* threaded:
  - no overhead from multi-threading

  - requires *asynchronous* programming:
    - to avoid blocking calls
    - nonblocking API
    - very different from traditional procedural programming
    - uses many callback functions

Example web server with Node:

```
let http = require("http");
let httpServer = http.createServer((req, res) => {
    // event-driven, defining callback whenever server receives a request
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.write('Hello world!\n');
    res.end('PATH: ' + req.url); // print URL
});
httpServer.listen(3000); // listen on port 3000, nonblocking call!
console.log("HTTP server started");
```

- Node modules is based on CommonJS instead of the ECMAScript 2015 standard:
  - syntax eg. `require(module_name)` and `module.exports`
- the **Node package manager (NPM)** helps intsall and manage third-party Node modules:
  - install eg. `npm install express`
  - the `package.json` file helps manage package dependencies:
    - when installing packages, the package and its dependencies is added to `package.json`
    - with `package.json` in the current directory, `npm install` installs all dependencies into `node_modules/`
    - also contains `scripts` and `dependencies`
  - `package-lock.json` contains all the packages, including dependencies, in detail
    - exact versions, etc.
  - version numbers follow the format `major.minor.patch` :
    - the prefix `~` indicates any patch version
    - the prefix `^` indicates an equal or higher version with the same major version

Express

- **Express.js** is a Node package for developing a web server with three key functionalities:

1. URL-routing mechanism
2. **middleware** integration
   - ie. a set of controllers / handlers
3. view template engine integration

Simple Express demo:

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

app.get('/', (req, res, next) => {
    res.send('Hello world');
});
app.get('/john', (req, res, next) => {
    res.send('Hello, John');
});
app.listen(3000);
```

- ExpresssURL routing:
  - `app.method(path, handler)` invokes `handler` for a request sent to an exact match on `path` via `method`
  - `app.all(path, handler)` handles *all* methods
  - parameters can be used embedded in the URL path itself:
    * eg. `app.get('/dogs/:breed', ...)`
    * `:breed` makes the matching substring available as a parameter at `req.params.breed`
  - a regular expression may also be used in the path
- request handlers take three parameters, `request, response, next` :
  1. the `request` object contains information on the HTTP request
     - eg. `app, body, query`
  2. the `response` object controls the response to be sent to the client
     - eg. `res.send(...)`
  3. `next` passes onto the request handling chain
- multiple handlers may be attached at the same path in a **request handling chain**:
  - when multiple handlers process a request, they are processed top down in the sequence they are attached
  - calling `next()` exits from the current handler and moves on to the next in the chain
- generating a response:
  - set status code eg. `res.status(200)`
  - set header field eg. `res.append(field, value)`
  - redirect eg. `res.redirect([status,] URL)`

- generating the body:
  1. raw string eg. `res.send(body)`
  2. static file eg. `res.sendFile(absPath)`
  3. JSON eg. `res.json(obj)`
  4. from a template eg. `res.render(templateFile, templateData)` :
     * generates an HTML page from `templateFile` using `templateData`
     * multiple template engines exist, eg. Pug, EJS, Mustache
- EJS is a popular template engine used with Express:
  - similar to JSP, uses scriplet tags
  - `<% ... %>` is used for control-flow with no output
  - `<%= ... %>` prints out the result of the expression after HTML escaping
  - `<%- ... %>` prints out the raw result of the expression without HTML escaping

EJS example:

```
<!DOCTYPE html>
<html>
<head><title><%= title %></title></head>
<body>
    <ul>
        <% for (let post of posts) { %>
            <li><%= post.title %></li>
        <% } %>
    </ul>
</body>
```

- advanced URL routing:
  - `app.use([path,] middleware)` for prefix routing:
    * `path` is interpreted as a prefix instead of an exact match
    * `path` prefix is then removed in `req.path` passed to middleware
  - `express.Router()` creates a “mini Express app”:
    * create one `Router` per prefix, and mount them on the corresponding prefix with `use`
    * inside each rotuer, use `router.method` to handle subpaths
    * allows for modular, hierarchical development
- standard middleware:
  - `express.static(absPathRootDir)` serves static files from a root directory
  - `body-parser` package is a collection of HTTP body parsers
    * eg. `bodyParser.json()`
- what if an error occurs during request handling?
  - call `next(err)` to get into an error-handling mode:

    * stops request handling chain and invokes an error handler
    * `next()` with no parameter moves onto the next request handler, while `next(err)` moves onto the error handler
  - similar to throwing and catching exceptions
  - the **error handler** is a callback function `cb(err, req, res, next)` :
    * additional `err` parameter
    * the default error handler in Express simply prints out the error
    * to use an error handler, simply mount it with `app.use` at the end of the middleware chain
    * multiple error handlers can be attached and called in sequence with `next(err)`
- the Express application generator can be used to generate skeleton code
  - `express -e` , use EJS template engine
- MVC in Express:
  - templates corresponds to views
  - handlers, routers, middleware correspond to controller
  - nothing yet corresponds to the model

Single Page Applications

- a **single-page application (SPA)** is a web application where everything happens on a single page:
  - no page reload and waiting
  - even when the browser needs to obtain data from the server
  - creates a desktop-app like experience
  - mechanisms required:
    * need to detect certain user events
    * allow user interaction to continue while awaiting data from the server
    * need an asynchronous HTTP request and response API
- `fetch(url)` is commonly used:
  - asynchronous API to issue an HTTP request
  - is a new "promisified" version of the old `XMLHttPRequest`
  - sends a request and returns a promise that will be resolved to the response from the server
  - to obtain the response body, we can use `response.text()` or `response.json()`
    * returns another promise that resolves to the body in the request format
  - the request can be customized by adding options to `fetch`
    * eg. methods, headers, body
  - note that the returned promise is rejected *only* in the case of a network error:
    * `4xx, 5xx` status codes resolve normally
    * `response.ok` is set to `false` for non- `2xx` status code
    * alternatively, access status code directly through `response.status`
- due to the **same-origin policy**, `fetch` can send a request only to the *same* host of the page:
  - ie. cannot send a request to a third-party site
  - a browser policy
  - possible workarounds:
    * run a proxy on the same host that forwards the request
    * **cross-origin resource sharing (CORS)**
      · browser gets an explicit approval from the third-party server to receive any requests by checking for the `Access-Control-Allow-Origin` header
    * JSONP
  - modern browsers take care of CORS automatically:
    * server should be configured to respond with the access control

header
    * by default, third-party cookies are not sent to the cross-origin server for added security
        · the option `{credential: include}` for `fetch` sends cookies, server needs to respond with the `Accces-Control-Allow-Credentials` header
- **extensible markup language (XML)** is a data representation standard with a semantic tag:
    - any tag name can be used to represent the data
    - unlike HTML tags that are used entirely for document structure, not semantics
    - XML is still popularity, but JSON is growing
    - to parse an XML string into XML DOM, we can use the `DOMParser`
        * the DOM tree can be accessed through JS DOM functions eg. `getElementsByTagName`
    - to serialize an XML DOM into an XML string, we can use the `XMLSerializer`

Example XML to HTML snippet:

```
function xml2html(res_text) {
    let parser = new DOMParser();
    let xml = parser.parseFromString(res_text, "text/xml");
    let s = xml.getElementsByTagName('suggestion');

    let htmlCode = "<ul>";
    for (let i = 0; i < s.length; i++) {
        let text = s[i].getAttribute("data");
        htmlCode += "<li><b>" + text + "</b></li>";
    }
    htmlCode += "</ul>";
    return htmlCode;
}
```

- the back button may cause a usability issue:
    - user may expect the previous app state *within* the SPA
    - browser may instead unload the app and go to the previous page
    - within a SPA, how can we solve this **deeplink** issue ie. go back to a particular state of the app if all states are the same page?
        * one approach is to use a **URL fragment identifier**
    - changes in URL fragment identifiers do not reload a page:
        * allow for navigation within the same page eg. `http://test/path#fragment`
        * we can associate each state of the app with a unique URL fragment
        * browser will change the URL, but the page will not change

    - the client must monitor the fragment identifier change event to instead perform a state change event through `window.onhashchange`
        * `window.location.hash` holds the URL fragment identifier
- the **session history API** added in HTML5 is another approach for sessions:
    - `history.pushState(obj, title, url)` and `history.replaceState(obj, title, url)`
    - allows saving an object
    - when users navigate history through the back button, the pop state event is triggered
        * `window.onpopstate` is the event to update the app using the popped object
- the **web storage API** allows for persistent data storage on the client-side:
    - would allow SPA to work using saved data even with no network
    - downsides of a cookie?
        * can expire, and must be sent to the server
    - `localStorage` is an associative array for persistent client-side data storage
        * lasts over multiple browser sessions
    - `sessionStorage` persists only within the current browser tab:
        * data disappears once the browser tab is closed
        * if two tabs from the same server is opened, they get separate storage
    - standard allows storing any object for local and session storage, but most browsers only support string
    - `IndexDB` is a more advanced local storage API with support for:
        * JSON object storage
        * transactions
        * non-blocking asynchronous API

Client Side Frameworks

- as the app becomes more complex, the code becomes less simple to modularize and much more difficult to maintain:
  - what would the code for Gmail look like?
  1. code complexity
     - thousands of lines of JS, HTML, CSS
  2. lack of modularity:
     - many global variables and name conflicts
     - code maintenance difficulty
  3. code reusability
- framework idea:
  - any complex UI apps consists of simpler **components**:
    * each component should be mostly generic and independent of others
    * structure and develop the app to exploit this independence and reusability
  - can we split a complex program into independent *modules*?
  - can we develop and provide a *library* of commonly-used independent modules?
  - can we program at a *higher-level* than DOM elements?
- component-based development:
  1. split the app into a hierarchy of simpler components
  2. develop each component independently with unit testing
  3. combine simple components into more complex ones
  - advocated first by React, now adopted by all popular frameworks:
    * reduces development and maintenance complexity
    * local changes are limited to a particular component
- case conventions:
  - camel case is typically used in JavaScript
  - kebab case is typically used in HTML and filenames

Angular

- **Angular** is a web frontend development framework developed by Google:
  - supports development of complex SPAs
  - provides easy-to-use end-to-end development tool-chain
  - encourages modular development through components and services
  - one of three most popular frameworks together with React.js and Vue.js
- Angular CLI:

  - `ng new <app-name>` generates initial skeleton code
    * main code is in `src/app`
  - `ng build --prod` builds the final production HTML, CSS, and JS files in `dist/`
  - `ng serve --host 0.0.0.0` starts up a temporary server that detects changes to source files:
    * helps avoid manual recompilation and deployment
    * only for development
    * note that Angular runs in the browser and *not* the server
  - `ng generate component <component-name>` generates skeleton code for a component
- in Angular, the app is split into modular components:
  - each component is developed independently with unit tests
  - a **component** is a specific part of an app responsible for a certain UI interaction:
    * eg. label list, search box, email list, etc.
    * the `@Component` decorator takes `selector, template, styles` as options
      · alternatively `templateUrl, styleUrls`
  - the HTML **template** for a component determines what a component displays on the page
  - **component directives** are custom Angular HTML tag extensions
    * eg. `<app-search-box></app-search-box>`
- **data binding** allows interaction between template and its class:
  - we want the component template to interact with its class dynamically
    * several different data binding mechanisms
  1. in **interpolation**, we use the syntax `{{ expr }}` to replace with the result of `expr` :
     - eg. `{{ title }}` displays the `title` property of the component in its template with
     - the expression should have no side effect
  2. in **property binding**, we use the syntax `[property]="expr"` to set the value of a property of an HTML element:
     - the `@Input` decorator defines which properties bindings are exposed to parent components
     - eg. `[value]="defaultQuery"` sets the default value of an input to the result of `defaultQuery`
     - importantly, whenever the value of `expr` changes, the `property` value is *dynamically* updated
  3. in **event binding**, we use the syntax `(event)="statement;"` to call a class method on a certain event:
     - executes `statement` when `event` is triggered:

    * all standard DOM events eg. from `input` will bubble up to parents
    * but a component can throw its own *custom* events as well
  - the statement may have side effect
  - to get access to the default DOM event, we can use `$event`
  - note that if we attach a regular click handler through `onclick=...` , this click handler will not have access to the component methods
    * ie. this click handler is not compiled into Angular-specific code
- custom Angular events:
  - eg. `<app-search-box (advice)="onAdvice($event);">`
  - to "throw" a custom event:
    * we need an `EventEmitter` object and assign it to a property
    * add the `@Output` decorator to make the object available for event binding
    * then, calling `emit(obj)` on the property will throw an event with the property name
      · `obj` is passed as the `$event` object
  - eg. `@output() advice = new EventEmitter<string[]>();` in search box component
    * `onAdvice` will have access to an emitted string array when the child calls `emit(advice)`
  - note that custom events *do not* bubble up and only its direct parent can catch custom events
    * vs. standard DOM events
- we can use attribute directives in components to allow interaction between parent and child components:
  - creating extended components that look and behave like a standard HTML element:
    * property and event binding act as an API for the component
      · name is the component directive, inputs are property bindings, outputs are event bindings
    * can eg. create custom properties that parent components can control
    * can throw custom events that parent components can intercept and additionally add handlers for
  - eg. `<app-search-box [query]="title" (input)="onInput($event);">`
- **structural directives** allow different HTML elements depending on a class property value:
  - include `*ngIf, *ngFor, *ngSwitch`
  - eg. `<img [src]="imgUrl" *ngIf="imgUrl"` creates an element and its descendants only if the `imgUrl` is truthy
  - eg. `<li *ngFor="let item of items">` creates one DOM element per

each element in `items`
- `item` is a template input variable

`ngSwitch` example:

```
<ng-container [ngSwitch]="media.type">
    <img [src]="media.url" *ngSwitchCase="'image'"/>
    <video [src]="media.url" *ngSwitchCase="'video'"></video>
    <embed [src]="media.url" *ngSwitchDefault>
</ng-container>
```

- an Angular **service** provides services that can be used by many components

Reactive Programming

- **reactive programming** is a programming paradigm that revolves around observables and events:
  - a functional, declarative, asynchronous paradigm
    - **RxJS** is a JS library for reactive programming
  - useful for operating on lists that are expensive to iterate through:
    - there may be latency associated wit accessing the next element
    - eg. a list being sent over a network
  - in a reactive program, we write a set of operators performed on observables:
    - consists of *reactions* to input events
    - ie. reactive programs "react to" input events
  - an **observable** is a generalization of an iterable object that accounts for asynchronous list accesses:
    - `onNext` will be called on every item `e`
    - `onCompleted` will be called
    - AKA a publisher or an object that produces a sequence of events
    - note that everything is observable eg. arrays, iterables, events, variables
      - eg. sequence of events, a variable is simply a single element array
  - an **observer** is an object interested in the events from an observable
    - AKA a subscriber
  - an **operator** transforms input observables into output observables:
    - eg. `filter, map, reduce`
    - complex operators can be created by piping together simple operators
  - observables are most assumed to operate through a "push" operation

        * instead of constantly monitoring for the next element to be available, it will be notified
- when are observables useful:
    - observables can be used for any type of programming
    - particularly useful when dealing with *streams* of events:
        * UI apps
        * asynchronous programs
        * servers
    - reactive programs are *declarative*:
        * different from procedural or imperative programming
        * declarative programs provide enormous optimization opportunity
    - reactive programming uses **pure functions**:
        * the same input always gives the same output ie. function can be understood on its own
        * no side effects ie. function does not change outside states
- reactive operators:
    - `filter` filters those events that meet a condition
    - `map` maps every input event to an output event
    - `flatmap` creates multiple output events from one input event
        * ie. flattening the output
    - `reduce` performs cumulative operations to produce one final output at the end
    - `scan` is similar to reduce, but produces one output per every input based on the cumulative progress
    - `buffer(time)` buffers input events for a specified period and produces buffered inputs as output
        * `bufferTime(timeSpan, creationInterval), bufferCount(m, n)` perform similarly
    - `debounce(time)` produces an output after a specified period of inactivity
- multi-way operators take multiple input streams as input:
    - `merge` merges events from all input streams into a single output stream
    - `zip` takes one event from each input stream and generates an output from the pair
    - `A.buffer(B)` buffers events from `A` until `B` emits a new event
        * also `A.bufferToggle(opening, closing)`
    - `join(time)` produces one output per every input event pair within a time window
- ex. Convert a single-click stream into double or triple clicks if there is less than a 250ms pause between clicks:
    - `clickstream.buffer(clickstream.debounce(250)).map(e => e.length)`

CSS

Example CSS selectors:

```
p {}        /* p element */
p.notes {}  /* p element of notes class */
p .notes {} /* element of notes class that is a descendant of p */
#text42 {}  /* id text42 */
img[src$=".svg"] {} /* src attribute ending with .svg  */
```

- CSS inheritance:
  - CSS can be specified in three places:
    1. browser default
    2. user preference
    3. web page
  - if not set in any of these places, an element will *inherit* its parent's CSS properties
- the CSS **cascading rule** dictates which CSS rule wins in case of a conflict:
  1. more specific rules win
     - id > class > tag
  2. source order:
     - if equal specifity, a later rule wins
     - web page > user preference > browser default
- CSS **variables** AKA custom properties allows using a logical name to specify a value:
  - wherever a CSS custom property is defined, only the descendants can see the property
    * thus variables are typically defined in `body` or `root`
  - must start with `--`
  - can be referenced with `var(..)`
    * can also give fallback values
  - eg. `--dark-bg-color: brown` , `var(--dar-bg-color, black)`
- CSS box model and positioning:
  - width, height, padding, border, margin
  - `position` can be:
    * `relative` ie. relative to its normal position
    * `absolute` ie. relative to its nearest positioned ancestor
    * `fixed` ie. relative to the viewport
    * `static` is the default, ie. element is unpositioned
- block vs. inline elements:
  - inline elements do not create a separate block, and instead flow with surrounding text

  - width, height, and vertical margin properties are ignored for inline elements

Example CSS for a fixed header and side menu:

```
#header {
    width: 100%;
    height: 90px;
    position: fixed;
    left: 0;
    top: 0;
}

#menu {
    width: 100px;
    height: calc(100% - 90px);
    position: fixed;
    left: 0;
    top: 90px;
}
```

- CSS grid:
  - container properties:
    - `display: grid`
    - `grid-template-rows` specifies the size for each row
      - similarly for `grid-template-columns`
  - item properties
    - `grid-column-start, grid-column-end` , etc. specify how many rows or columns tos pan
- in **responsive web design**, we design for a wide range of devices:
  - eg. phone, tablet desktop, etc.
  - page design should dynamically adapt to the screen size
  - in *fixed* layout, elements have fixed width
    - resizing the window does not change their sizes
  - in *fluid* layout, elements instead use a percentage of page width
    - elements dynamically resize to fit window width
  - general rules:
    - do not force users to scroll horizontally
    - do not use fixed-width elements
    - use CSS media queries to apply different styling based on screen size:
      - eg. `@media condition { ...rules... }` like `@media (max-width: 800px) {}`
      - media types include `screen, print, speech, all`
      - media features include `orientation, min-width, resolution` etc.

        - boolean operators for the condition include `,` `and` `not`
    - the `viewport` meta tag defines a user's visible area of the web page:
        * `width` is the viewport width
        * `initial-scale` is the initial zoom level
        * generally want to override the default viewport setting for smaller devices
- CSS flexbox:
    - a new addition to CSS to enable flexible layout of elements with `display: flex` on the container
        * all children of a flex container become flex items
    - changing size:
        * `flex-basis` is the default size of an element
        * `flex-grow` specifies how to divide extra remaining space
        * `flex-shrink` specifies how to take away space when there is space shortage
    - rearranging inputs:
        * `flex-wrap` wraps the flexbox
        * `flex-direction` determines the wrapping direction
- animations:
    - can use JavaScript, eg. with `setInterval/setTimeout` and `style` property of elements
    - CSS animation:
        * `transition` property creates a transition effect eg. `transition: height 1s;`
        * `@keyframes` rule specifies the keyframes of an animation
        * `transform` can specify more complex shape transformations

Keyframes example:

```
@keyframes background-change {
    0%: { background: red; }
    50%: { background: yellow; }
    100%: { background: green; }
}

#someId {
    animation: background-change 3s;
}
```

Scaling

- how do we plan for capacity when deploying a website on servers?
  - number of machines and number of requests a machine can handle depends on different applications
  1. set your minimum acceptable service requirement
  2. characterize the workload
     - eg. measure requests per second, resource utilization per second
  3. premature optimization is "the root of all evil"
     - measure the workload first
- how many *static* web pages can a standard machine handle per second?
  - typical machine speeds:
    * disk and DB IO transfers between 100-3000 MB/sec, and seeks on average 5-10 ms
    * memory transfers 10-50 GB/sec
    * network transfers 1-10 Gbps
  - just have to serve data from disk or memory over the network
    * main bottleneck is mostly disk (reduced with caching) and network IO
  - DNS lookup is often very slow, so reverse DNS lookup should be disabled
  - tens of thousands per second per core, billion requests per day
- how many *dynamic* web page can a machine serve per second?
  - depends on the complexity of the application
    * any of IOs, context switches, CPU may bottleneck
  - rule of thumb is 10 requests per second per core
- basic UNIX monitoring tools:
  - `top, ps, pstree` for CPU and processes
  - `iostat` for disk IO
  - `netstat` for network IO
  - `free -m, ps axl, vmstat, memstat` for memory
- can we use **caching** to improve performance and scalability?
  - what layer to cache at?
    * database, application, HTTP, encryption layer?
  - *below* the database layer, ie. at the filesystem level, cache disk blocks
    * add RAM, or increase database bufferpool size
  - *above* the database layer, cache database objects like tuples in RAM:
    * minimize number of requests hitting the DB
    * common tools include Memcached and Redis
      · generally support distributed caching
- caching above the application layer:

  - store and cache generated HTML page as a static file:
    * avoids generating HTML for a short lifetime, eg. if page was already requested in the last 10 seconds, simply serve the last cached copy
    * microcaching AKA caching for a very short period
  - especially useful for slow-updating dynamic sites or if short delay is tolerable
    * eg. blogs, web forums, etc.
  - what if a *small* part of a page has to change every time?
    * eg. Reddit
    * use a **edge-side include (ESI)** to separate out uncachable parts from the cachable part
      · eg. `<esi:include src="part1.html"/>`
    * the ESI server fetches all parts and synthesizes the final pages
      · regenerate and cache each part at a different granularity ie. expiration date
- caching above the HTTP layer:
  - use a **content distribution network (CDN)**
    * cache pages, images, videos, etc. close to users at the *edge* of the network
  - users access cached objects located close to them:
    * lower delay
    * lower load on network
  - CDNs are a must for sites like YouTube
- caching above the encryption layer:
  - browser cache
  - let browser cache decrypted pages locally
- how can we scale a web site as it grows?
  - scale up and buy a larger, more expensive server
  - scale out and add more machines
- scaling DB layer?
  1. global read-only access:
     - requests do not change the underlying database
     - replication thus has no synchronization issue for read-only accesses
  2. local read and write:
     - touching a very specific part of the data eg. web mail, banking
     - if we are doing replication, we can only scale the reads, but not the writes
     - instead, partition / split up the database ie. **sharding**
       * need to route requests to the correct machines
  3. global read and write:
     - writes are globally visible eg. online auction, social network

    - replication and partition still work, but to less degree
        * eventually write requests saturate the DB
  - CPU is rarely a bottleneck for scaling out:
    * instead, DB and storage becomes the main bottleneck
    * identify early on how we will cache / replicate / partition the DBMS as number of users grow
- cluster computing:
  - eg. Kubernetes: - automatic deployment and management of containerized applications - progressive rollout of application changes - automatic scaling and load balancing of apps based on CPU usage - automatic restart of failed, unresponsive nodes