CS 142 Final Examination

Winter Quarter 2022

You have 3 hours (180 minutes) for this examination; the number of points for each question indicates roughly how many minutes you should spend on that question. Make sure you print your name and sign the Honor Code below. During the examination you may consult two double-sided pages of notes; all other sources of information, including laptops, cell phones, etc. are prohibited.

I acknowledge and accept the Stanford University Honor Code. I have neither given nor received aid in answering the questions on this examination.

____
(Signature)

____
(Print your name, legibly!)

____@stanford.edu
(SUID - Stanford email account for grading database key)

Problem	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
Points	12	12	12	12	10	16	10	12	16	10
Problem	#11	#12	#13	#14	#15					Total
Points	10	15	12	11	10					180

Problem #1 (12 points)

The “same origin property” allows browsers to isolate cookies from different websites. Websites at different locations (i.e. hostname and port number) on the Internet can be assured that their cookies aren't accessible to the other locations. In addition to protecting cookie access at different locations, the “same origin property” also includes the scheme in the origin definition. Thus, a single website supporting different access protocols like HTTP and HTTPS might require multiple cookies, with one per location/protocol pair.

A. A website could choose to hand out the same cookie for both HTTP and HTTPS and not have to worry about different cookie values. Explain the advantage for the backend engineer of having the same cookie for both protocols.

   If a website used the same cookie for both HTTP and HTTPS protocols, this would be convenient for backend engineers because they would only have to worry about one cookie’s values that would point to the session state. Using something like express-session would mean that each of the protocols would have its own session state object for the same user. Having the same cookie would give one session state shared across the protocols
   .

B. Explain why having the same cookie value won't be a good idea and the website should use different values.

   Having the same cookie value for both HTTP and HTTPS is dangerous because an adversary could carry out a “Session Hijacking” attack. An adversary could steal your cookie from the website supported by HTTP by implementing a “man-in-the-middle attack”, where they simply read your cookie from an unencrypted connection. They can then use it to impersonate you even in areas of the website supported by HTTPS.

Problem #2 (12 points)

Our photoShare React.js program uses HTTP GET requests to fetch model data from the Node.js web server. For example, HTTP GET to the URL `/photosOfUser/:id` where `:id` is the MongoDB id of a user object will return model data of all the photos of a user.

Express.js allows our web server to fill in the HTTP response to the HTTP GET request so the body of the response has a JSON-encoded array of the photos. Express.js also allows us to set properties in the HTTP response header. For example:

```
response.set("Cache-control", "max-age=300");
```

will set the HTTP Cache-control property in the response header.

A. Explain the advantages and disadvantages (from the perspective of the web app developer) of setting this `Cache-control` value when fetching model data from `/photosOfUser/:id`?

   A web app developer might notice that their data will load more quickly on the front-end, since the response data is periodically cached (rather than having to search through the database each time a GET request is made to that endpoint). However, they will now notice that there will be a delay in fetching/rendering real-time updates to the model data, because the response data to the GET /photosOfUser/:id endpoint will be cached for 5 minutes. So, any data from this specific endpoint which is changed within that 5 minute cacheing window will not be accessible until the cache window ends, and a fresh GET request is made.

B. What changes in app behavior would the end user of the photo app see from this change?

   Suppose a user visits a UserPhoto's page. For the 5 minutes following, any newly-added (or deleted) data (such as photos, comments, etc.) which is rendered directly from an HTTP request to GET /photosOfUser/:id will not show up until the cache-control max-age directive expires and a fresh fetching of the data is made. However, the cached data will load more quickly.

Problem #3 (12 points)

A. When using the MVC (model, view, controller) decomposition for view construction, all three MVC components must be present for the view to be rendered. When the rendering is done in the browser as we did for our React.js photoApp, we end up fetching the components from the web server. What can you say about the order that the components are fetched?

   In React.js, each *component* contains both the view and the controller. In our photoApp we load the webpack JavaScript bundle that contains all the React.js components with a single script tag when the page is first loaded. Once the components are mounted we fetch the model data of the view. Given this, the view and controller are fetched at the same time (in the JS bundle), and the model data is fetched later using API calls to the REST endpoint.

B. If we consider the most optimal solution for fetching the model data of MVC components, it would be to launch a single request that specified all the model data needed for the currently rendered MVC components. Explain why GraphQL is superior to REST APIs in achieving this optimal approach.

   REST API exports model data as collections of resources with an HTTP GET request used to fetch a single resource. A web application with model data from multiple resources would thus require multiple GET requests, one for each different resource. GraphQL's ability to submit a query that specifies properties for multiple resources would allow fetching the same amount of data in a single HTTP request, achieving the optional solution whereas REST APIs can not.

Problem #4 (12 points)

The Domain Name System (DNS) is the system used by browsers that allow URLs to contain hostnames (e.g. `www.stanford.edu`) rather than the actual IP address of the web server (e.g `146.75.94.133`). Although a Content Distribution Network (CDN) is not a browser, it also utilizes the DNS system. Describe how a CDN uses DNS.

Unlike a browser that uses a DNS client to map the hostname in URLs to IP addresses, CDN implements a DNS server that maps the hostname in the URLs given to CDN clients to the IP address of a geographically nearby web server. Using this approach, a CDN can have the same URL connect to a server near the browser.

Problem #5 (10 points)

REST APIs have web servers export the abstraction of resources that the client can access and manipulate. For example, the client can send an HTTP GET request to read a particular resource or an HTTP POST request to create a resource.

In systems, we describe an operation as being **atomic** if the operation as a whole will either happen or not. An atomic operation can't only partially update the state. The term gets its name from the notion that an atom is indivisible.

Describe the problem of using a REST API to implement an atomic operation that would create two resources.

Since REST APIs use HTTP POST requests to create a resource, creating two resources would require two independent POST requests. Even if these two requests were done back-to-back, it is possible some kind of failure would result in only one of the requests happening. This kind of partial update in the face of a failure is incompatible with the definition of atomic operations.

Problem #6 (16 points)

We saw in class that code injection attacks can happen both in the browser and in the web server. Although these are very different environments potentially located on different continents, an attack in one location can be used to set up an attack in the other location. For each of the scenarios below, describe how an attack would work.

A. A code injection attack in a browser leads to a code injection attack on the server.

   An adversary can first stage a reflected cross-site scripting (XSS) attack, in order to send a malicious request to the server while making it look like it's coming from a legitimate user. The malicious request payload could then trigger a SQL injection attack, thus (as an example) erasing the website's production database.

B. A code injection attack on the server leads to a code injection attack in the browser.

   An adversary can first leverage a SQL injection attack to store some malicious HTML code in the database (e.g., replace Taylor Swift's Twitter biography with a <script> tag that loads a cryptocurrency miner). Subsequently, when legitimate users visit Taylor Swift's Twitter profile, they fall victim to a stored XSS attack and start mining cryptocurrency for the attacker.

Problem #7 (10 points)

Message authentication codes (MACs) are normally generated on the server and sent to the client's browser. Consider modifying our Photo App to compute a MAC of each of the images stored in our system. We would then transfer an image's MAC along with the image to our web app frontend. What use, if anything, could our JavaScript code in the front end make with these image MACs? Explain your answer?

A MAC generated on the server uses a secret key known only to the server. Without this key, the JavaScript code in the front end can't do much with MAC. It looks to the frontend like a random sequence of bits that is passed along with each image. There isn't much use the frontend can make of it.

It also does not make sense to make the frontend know the secret key, since then anyone who can run the frontend can find the key and use it to forge more MACs.

Problem #8 (12 points)

A. Describe what **extended validation certificates** have in addition to normal **certificates** when issued by a certificate authority.

These are the highest levels of TLS certificates available. They verify the identity of the website/domain owner with various checks and thus increase the trust users have when interacting with a domain. So, not only is the domain ownership verified but also the identity of the entity running the domain is verified.

B. Describe the problem **extended validation certificates** are trying to address.

These certificates are trying to address phishing attacks, since it's relatively simple to verify a domain for free online and appear trustworthy to users. The extra identity checks offer another layer of security and ensure that a site owner is verified as a safe entity.

Problem #9 (16 points)

In language environments with threads like Java and C++ there is usually a "sleep" function call that will pause the thread's execution for some amount of time. For example, the function:

```
function test(x) {
    console.log("A");
    let p = sleep(x);
    console.log("B");
}
test(10);
```

would output A followed by B 10 seconds later.

A. In JavaScript, you could write a "sleep" function that simply looped reading the time until time had advanced "x" seconds. The above function would have the same A followed by B 10 seconds later functionality. Explain why this wouldn't be considered an acceptable way to implement sleep in JavaScript.

   JavaScript is single-threaded. Having a sleep function loop for 10 seconds will block execution for 10 seconds, which is an undesired behavior. The web app would not be able to process during that 10 seconds any user input that used events.

B. JavaScript has promises to deal with this problem. Assume the "sleep" in the above code is a JavaScript function that returns a promise that is resolved in "x" seconds. Explain why the promise version would no longer exhibit the expected behavior (i.e. A followed by B in 10 seconds) and show what changes would need to be made to have the JavaScript work. Note that test is not declared to be an async function so await is not available.

   If sleep returns a promise the above function would log 'A', create and return the promise, and then log 'B'. The creating and returning the promise will be fast so ';A" and "B" will have 0 seconds between then regardless of the number passed to `sleep`.

   To fix this we need to wait until the promise resolves to do log the 'B'. Like:

```
function test(x) {
    console.log("A");
    let p = sleep(x);
    p.then(() => { console.log("B"); });
}
```

Problem #10 (10 points)

The Express.js session module we used in our web server generated a cookie that contains a pointer to the session state stored in the web server's memory. The session state of the photo app was quite small in size so we could replace the pointer with the state itself. Assume we keep security used to protect the pointer and have it protect the session state.

Would the approach of keeping the session state in the cookie better scale to a large number of users, compared to the original pointer-based approach? Justify your answer.

The approach of keeping the session state in the cookie scale to a large number of users is better than the original approach. With a pointer in the cookie, we have to store the session object in some session store. As the number of users and hence the number of session objects gets large, this session store will be large. On the other hand, if we store the session state directly in the cookie, we don't need this session store anymore and need not worry about the amount of space it uses. By storing the session state in the cookie, we get the user's browser to store the session state for us.

Problem #11 (10 points)

An eavesdropper looking at the packets flowing between the browser and a web server would see very different content depending if HTTP or HTTPS protocol is being used. Nevertheless, the Express.js handlers in the web server can be identical for the two protocols. Explain how protocols with different packet contents can use the same Express.js handlers?

Networking stacks operate in layers, and HTTP/HTTPS operate at a lower layer than the HTTP request processing that Express.js handlers do. HTTP and HTTPS are really the same protocol, the only difference being that HTTPS applies TLS encryption to underlying network packets while HTTP transmits them in plain text. Since Express.js operates at a higher layer (i.e., the packets Express.js deals with are already decrypted), it can provide a consistent API for both HTTP and HTTPS even though the network packets are different.

Since both HTTP and HTTPS traffic will be routed to the same Express.js handler, developers can check the secure attribute, or the protocol attribute of request to determine which protocol it's using.

Problem #12 (15 points)

Cryptography has been helpful for addressing some of the attacks that web applications face. For each of the following attack types, state if cryptography could be helpful and if so, how.

A. Network Attacks

Encryption of the HTTP traffic between the browsers and web server can defeat eavesdropper or man-in-the-middle type network attacks.

B. Session Attacks

Encrypting the HTTP traffic also prevents eavesdroppers from stealing session cookies to do session attacks. MACs can also been used to prevent the forging of session cookies.

C. Code Injection Attacks

Unencrypted HTTP makes it easy for a man-in-middle network attack to inject script tags into the HTML being fetched into the browser. Encrypting prevent this kind of attack.

Problem #13 (12 points)

The concept of a “**done callback**” function is widely used in JavaScript library interfaces. Rather than returning a value directly, the library routine will call the provided callback function at some later time with the requested value. For example:

```
      fs.readFile(fileName, doneCallback);
```

On the other hand, if the library routine has multiple different kinds of things it can return at different times, it can accept multiple callbacks. For example:

```
      routine(args, doneCallback1, doneCallback2, doneCallback3, …);
```

where the different callback functions are used to return the different values.

Things get complicated if there are many different return values that the caller may or may not be interested in. Explain the mechanism in Node.js that allows this kind of library interface to be more cleanly implemented. Describe how it works better than the multiple callback method.

Node.js primarily uses the event listener pattern (sometimes also called the observer pattern). Under this pattern, each interested party can "subscribe"/"listen" to only those "events" that it is interested in, where each event can hold a value. As an example,

```
      routine(args, callback1, callback2);
```

could be rewritten as:

```
      const emitter = routine(args);
      emitter.on('event1', callback1);
      emitter.on('event2', callback2);
```

Compared to a multiple-callback approach, the event listener pattern has several advantages:

1. Rather than having to specify every argument for `routine()`, the caller only needs to specify callbacks for the events that it is interested in.
2. Rather than relying on the position of each argument, the event listener pattern allows the caller to identify each callback function by a string (the name of the event). This is less error-prone for the programmer, and simplifies refactoring when the maintainer of the library inevitably wishes to add a new event or remove an existing one.
3. The event listener pattern naturally allows multiple callbacks for a single event, so several parties could listen on any set of events of interest to them on the same emitter.

An alternative to the traditional event listener pattern is using an "options" object to specify callbacks. This carries the same advantages 1 and 2 as event listeners, but not advantage 3. Example:

```
      routine(args, { event1: callback1, event2: callback2 });
```

However, this approach is less commonly seen in Node.js.

Problem #14 (11 points)

HTTP uses TCP/IP protocol for communication. TCP/IP is what is known as a connection-oriented protocol where the protocol starts by establishing a connection between two machines much like a telephone call.

Explain how a scale-out architecture for web servers can be accomplished if the browser thinks it is calling up some machine by IP address for each request it sends.

Although HTTP runs on top of TCP/IP which is a connection-oriented byte-stream protocol so the browser thinks it is connecting to a particular web server, the scale-out architecture for web servers uses some kind of load balancing approach that will spread the connections across a set of web servers. One common way of doing this is to deploy a special network router in front of the pool of web servers that will forward connections to a single IP address (the router's address) to one of the many available web servers (each with a different IP address).

Problem #15 (10 points)

The single-threaded nature of the JavaScript runtime in Node.js means that no two HTTP requests can be executing JavaScript at the same time. In spite of this limitation, Node.js can maintain multiple requests being processed at the same time by the MongoDB database. Explain how this is possible without concurrent JavaScript execution?

Although JavaScript functions never execute concurrently, HTTP request processing is broken in many functions and many requests can be between different processing functions at the same time. Since some of these processing functions are "start database request" multiple database requests can be active at one time.

Node.js has support for event and event queue. When a request is sent to MongoDB, the request won't be blocking. Instead, we first send out the request, and while waiting for the request to be processed by MongoDB we can run other code. When the request to MongoDB comes back, an event of executing its callback function will be added to event queue and then executed in order. Therefore, it is possible for our server to maintain multiple requests being processed by MongoDB at the same time.