

CS 142 Final Examination

Winter Quarter 2023

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	10	10	8	8	10	10	8	10	8	12
Problem	#11	#12	#13	#14	#15	#16	#17	#18	#19	Total
Points	10	10	8	10	10	8	8	10	12	180

Only the front side of the exam pages will be scanned. Do not write answers on the back of the pages.

Problem #1 (10 points)

The Node.js web server (`webServer.js`) you built for your photo-sharing app used Express Session to provide session state (i.e., the `app.use(session({secret: "secretKey", resave: false, saveUninitialized: false}))`; line we had you add in Project 7). The Express Session code creates a session cookie to track the session state you use. A disadvantage of using cookies for this is it can increase the size of the HTTP header in both the HTTP request (i.e. `cookie`) and HTTP responses (i.e. `set-cookie`). Fortunately, not every request and response is required to have an enlarged header. Below is a list of HTTP requests from the photo-sharing app that has the user "took" log in, upload a photo, and log out. Assume it is being run on a cold browser (i.e., no cookies or cached data). For each list request, mark if it has an enlarged header:

A)

HTTP Request	HTTP Request Header has cookie		HTTP Response Header has cookie	
	Yes	No	Yes	No
GET /photo-share.html	Yes	No	Yes	No
GET /compiled/photoShare.bundle.js	Yes	No	Yes	No
POST /admin/login	Yes	No	Yes	No
GET /user/list	Yes	No	Yes	No
GET /user/641522690b7c2361d10cba67	Yes	No	Yes	No
POST /photos/new	Yes	No	Yes	No
POST /admin/logout	Yes	No		

B)

Express session uses an optimization that allows the session state to get bigger without the cookie getting bigger. Describe how more session state doesn't require more cookie bytes.

Problem #2 (10 points)

A Content Distribution Network (CDN) gets content from the web application developer and ends up responding to HTTP requests from the web app users' browsers. The CDN gets to decide what header fields it sets in the HTTP responses it sends out. For each of the following header fields, state if the CDN would have a reason to set it and if so, what value would they use.

(A) Cache-Control: max-age parameter

(B) Access-Control-Allow-Origin:

Problem #3 (8 points)

In the HTTP lecture, the following JavaScript code fragment was listed on a slide:

```
elm.innerHTML =  
  "<script src='http://www.example.com/myJS.js' type='text/javascript' />"
```

We can assume that the code was loaded from the same web server as the script:

```
https://www.example.com
```

The fragment was labeled "Scary but useful". Is it any more or less scary if the code was changed to be:

```
elm.innerHTML = "<script src='/myJS.js' type='text/javascript' />"
```

Problem #4 (8 points)

Explain the problem with REST APIs when dealing with operations that span across multiple resources that are not present in GraphQL or RPC-based APIs.

Problem #5 (10 points)

Some students ran into problems coding the Project 6 Express handlers like

```
app.get('/user/list', function (request, response) {
```

The problem they ran into was that they tried to code the handler using `await` and got the error:
`SyntaxError: await is only valid in async functions`

In response to this error, they changed the handler setup code to be:

```
app.get('/user/list', async function (request, response) {
```

and not only did the error go away, it worked as expected.

In general, changing a function that is called by some other module (i.e. a callback function) to be an async function is not guaranteed to work. For each of the callback function scenarios below, explain why the addition of the `async` and `await` keywords could fail.

(A) A module accepts a callback function (`callback`) that returns a count of items. The module does `let itemCount = callback();`. The `async/await` change allows the callback function to read from the database.

(B) A module accepts a callback function (`doAllCallback`) that the module calls before signaling that everything is done. The module assumes that all of the processing of `doAllCallback` is finished when the callback function returns. The `async/await` change allows the `doAllCallback` function to read from the database.

Problem #6 (10 points)

When implementing RPC systems that can be used between browsers and web servers, some communication mechanism is needed. Since the browser and web servers already communicate using the HTTP protocol, it is an obvious choice to base an RPC implementation on.

(A) When using an RPC system to make calls from the browser to the web server, the RPC system often uses the POST verb. Explain why POST is preferred for RPC over verbs like PUT or GET.

(B) RPC systems that allow calls from the web server to the browser often choose not to run over HTTP. Explain why.

Problem #7 (8 points)

The operating system socket system calls are used to speak the TCP/IP protocol between the Node.js webServer.js and the MongoDB database server. Using this TCP/IP connection, the MongoDB node modules can send queries and receive results from the database. Since both the Node.js process and the MongoDB server are running on the same machine, the TCP/IP connection can be made using the hostname of "localhost".

This quarter some students' laptops ran into a problem. There are now two localhost addresses, one using IPv4 addressing and one that uses the newer IPv6. The problem occurred because there was a disagreement between the web server running in Node.js and MongoDB which localhost address (IPv4 or IPv6) to use. We were able to patch around this problem by forcing an agreement on which address to use.

The browser and webServer.js also use the hostname "localhost" to communicate. We load our web application from `http://localhost:3000/photo-share.html`. Although we haven't seen a mismatch of which "localhost" to use here, is there something in the browser environment, HTTP, JavaScript, etc., that would prevent this same problem? If so, describe it. If not, explain why.

Problem #8 (10 points)

The issue with "localhost" described in Problem #7 is made more complex since we speak to the database using two different node modules authored by different groups. Our code uses the Mongoose module, which talks to the MongoDB Node client library, which can speak to the MongoDB data server. When something goes wrong with this communication path, it's hard to tell which module (Mongoose or MongoDB client library) is at fault. What benefits would be lost if we got rid of Mongoose and talked directly to the MongoDB module?

Problem #9 (8 points)

Explain why relational databases can have both primary and secondary indexes but MongoDB users can only declare secondary indexes.

Problem #10 (12 points)

By the time we got to Project 7, our Node.js webServer.js was using multiple Express middleware modules. Each of these modules defines a handler function that takes three parameters: `function handler(request, response, next)`. Although these handlers are called on every incoming HTTP request, not all of the parameters are accessed on every call. For each of the following middleware insertion lines:

1. describe the middleware's use of the three parameters: **request**, **response**, **next**.
2. indicate if each parameter is accessed on every, some, or no calls.

```
app.use(session({ secret: "secretKey", resave: false, saveUninitialized:
false }));
```

```
app.use(bodyParser.json());
```

```
app.use(express.static(__dirname));
```

Problem #11 (10 points)

Browsers and web servers both talk TCP/IP and use the operating system socket abstraction to send and receive data formatted as HTTP requests and responses. In the table below, for each of the socket system calls, list if it is used **often**, **infrequently**, or **never** when doing HTTP communication.

System Call	Browser	Web Server
listen		
accept		
read/receive		
write/send		
connect		

Problem #12 (10 points)

The code fragment listed in the Database slides to update a User object with the id `user_id` was shown as

```
User.findOne({_id: user_id}, function (err, user) {  
    // Update user object - (Note: Object is "special")  
    user.save();  
});
```

The pattern works by first reading the object from the MongoDB database into a Mongoose persistent object that is updated and calling the `save` method on it causing the modified object to be written back to the database.

(A) This kind of read-update-write sequence is scary in a system with threads since we need to worry about multiple threads trying to update the same `user_id`. Assuming we have only a single Node.js web server this quarter, is there something about JavaScript, Node.js, or Express.js that makes this safe? Explain your answer and if the answer is no, provide an example.

(B) Does the answer to the question in part (A) change if we assume this code is running in a Node.js instance that is part of the scale-out architecture? Explain your answer and if the answer is no, provide an example.

Problem #13 (8 points)

When we added Express Session to our web server we used the JavaScript expression:

```
app.use(session({secret: 'badSecret'}));
```

`secret` is a required parameter object to the session module that is used to "sign" the session cookie. If we really used something like 'badSecret', an easily guessable secret, what could an attacker do to our web application?

Problem #14 (10 points)

Our React web application works by having front-end JavaScript code making REST calls to resources we implement using Express handlers in a Node.js web server. If we implement a REST endpoint in the web server but never add the corresponding JavaScript code to call this endpoint, we can have "dead code" that is never executed by our web application.

(A) Explain why our threat model requires that even "dead code" be hardened against attackers.

(B) If we always use HTTPS to defeat any possible "man-in-the-middle" attacks, do we still need to harden "dead code"? Explain your answer.

Problem #15 (10 points)

To allow React child components to communicate with their parent components, we showed a pattern that involved passing a function (a callback function) as a property from the parent to the child. The child can then call the callback function to communicate with the parent.

We presented an alternative way to support this communication using state managers like Redux. Rather than passing callback functions, this communication is done by having the parent subscribe (listen) for an event and having the child emit the event.



The callback function as a prop to a component approach got much power because the callback function could execute arbitrary JavaScript code. The state manager approach has the parent subscribe and listen for an event.

Does the callback function approach's ability to execute arbitrary JavaScript code give it more power than the state manager's listening for events? Explain your answer.

Problem #16 (8 points)

In the initial projects (Project 1-3) we didn't run a web server. It was possible to point the browser at the local filesystem using the `file:` scheme and read the HTML and CSS files that way. Using the `file:` scheme works to load our React web application bundle as well but any requests to model data fail to even go out of the browser. Explain why the browser won't let us load model data when the bundle is loaded using the `file:` scheme.

Problem #17 (8 points)

The Chrome browser shows a lock icon (e.g. ) when the page is fetched using HTTPS and a warning message ( **Not Secure**) if the page is fetched using HTTP. If the page is fetched using HTTPS and makes a single HTTP access (e.g. fetch an image or stylesheet) it is displayed with the warning message. Explain why a single HTTP can ruin the security of a web page.

Problem #18 (10 points)

For many things in life that fail, adopting a strategy of immediately retrying the failed operation is not often a successful strategy. Web applications using a scale-out architecture are sometimes an exception to this rule. Explain the reason for this.

Problem #19 (12 points)

One of the great things about programming with Node.js is the massive repositories of node modules that give easy access to much functionality useful for building web applications. Pretty much anything you might want can be "npm install"-ed.

One web area this npm approach has been less successful at addressing is security. For each of the listed problem areas, state if a node module could be useful. Explain your answer.

(A) Distributed Denial of Service

(B) SQL injection attack

(C) Cross-site scripting attack