

# 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)

**Solutions**

---

(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 |
|------------------------------------|--------------------------------|---------------------------------|
| GET /photo-share.html              | Yes <b>No</b>                  | Yes <b>No</b>                   |
| GET /compiled/photoShare.bundle.js | Yes <b>No</b>                  | Yes <b>No</b>                   |
| POST /admin/login                  | Yes <b>No</b>                  | <b>Yes</b> No                   |
| GET /user/list                     | <b>Yes</b> No                  | Yes <b>No</b>                   |
| GET /user/641522690b7c2361d10cba67 | <b>Yes</b> No                  | Yes <b>No</b>                   |
| POST /photos/new                   | <b>Yes</b> No                  | Yes <b>No</b>                   |
| POST /admin/logout                 | <b>Yes</b> 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.

Express session does not put the session state into the cookie. It puts the session state objects into a "session store" and puts a pointer into the store (i.e. sessionID) in the session cookie. Using this scheme more session state means a bigger session state object but doesn't increase the size of the sessionID.

## 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

CDNs work by taking unchanging content and distributing it around the world to have quick access to browsers all over the world. Because the content isn't changing and can't change quickly due to the distributed nature of the CDN, the CDN could safely get a max-age parameter so the browsers cache the content and not have to go back to the CDN on every access. This would save the CDN bandwidth costs and speed access to content from the browser's point of view. The value for the max-age would need to factor in what delays the CDN wants to guarantee before new content is visible in every browser.

(B) Access-Control-Allow-Origin:

If the CDN is operating from a different origin than the web application, the same-origin policy would isolate its resources from the web application. The CDN can get around this problem by putting the web applications origin (or a wildcard) into the Access-Control-Allow-Origin header so the resources would be able to the web application.

### 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' />"
```

Both statements add a script tag to the element that fetches some JavaScript from a URL and executes it. The URL used in the first statement is slightly different from the second:

```
http://www.example.com/myJS.js
```

while the second uses:

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

Without knowing what is in `myJS.js` the code would need to trust it doesn't do something harmful in both cases. In the presence of a man-in-the-middle attack the HTTP protocol used in the first example means the attacker could set the JavaScript to anything they wanted. The use of HTTPS in the second prevents this attack and hence would be less scary.

## 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.

REST API operations that span across multiple resources would need to be done with multiple REST calls from the browser to web server. This kind of multiple operations make failure handling much more difficult since you would need to handle failure than happened after some but not all the operations have been performed. The fact that the operations are being done over the Internet greatly increases the chances of something going wrong.

With GraphQL and RPC-based approach, a single remote operation can operate across multiple resources, greatly simplifying how failures are handled.

## 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.

Adding `async` to a function means it now returns a Promise that resolves into the return value. If the caller expected the callback function to return a count, it wouldn't like the Promise object returned instead. It would likely try to use it as a Number and hit some kind of failure.

- (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.

When an `async` function is called it returns at the point of the first `await` call in the function which could be before all the processing of the function has been completed. If the callback function really needed to wait until "everything is done" it won't be doing that.

## 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.

An RPC is a request-response system where the parameters of the call are packaged into a message that is sent to the server and the remote routine is invoked with the parameters. The result of the remote routine is returned to in a message to the caller. This match the POST verb well since it can have a body in the request (the parameters) and a body in the response (the result) of the remote machine's invocation. The PUT and GET only allow a body in one direction so multiple requests would be needed to transfer in both directions.

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

HTTP is a request-response protocol where browsers generate requests that web servers respond to. Browsers typically don't include web servers so there isn't a way for web server to communicate back with the browser using HTTP requests. If RPC calls from the web server to browser are needed, something like WebSockets that allow a request-response from the web server to browser and back can be used.

## 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.

TCP/IP connections using "localhost" require the process to do a name lookup of "localhost" using a system called the Domain Name System (DNS). That process is the same for both the Node.js and MongoDB connection as well as the Browser and Node.js connection. A TCP/IP connection processing is the same in both cases hence wouldn't be any different if the two sides used different values for localhost.



## 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?

Mongoose's object definition language allows us to define the schema for the MongoDB object collections so that we can be ensured that objects have the desired properties of the correct types. Errors that cause incorrect property values to be written to objects will be rejected before they can mess up the database. Code reading database object can assume that everything read from the database will the specified property names and types.

### Problem #9 (8 points)

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

Primary index is the index around which you can organize the storage, but in MongoDB, the storage is already organized around the id of an object. Hence, in MongoDB we 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 }));
```

The Express session manages the session state of the app. It is responsible for creating the sessionID cookie and fetching the session state on each request. Since it operates by fetching and setting up `request.session` on every request it will call `next()` on every request or pass control along. Once session state is created it will write `request.session` on every request. To initialize the session cookie it will need to add a set-cookie to a `response`.

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

The body parser module is responsible for parsing the JSON in the body of the request and assigning it to `request.body`. It will need to call `next()` on every request to pass control along. For requests that contain a JSON encoded body (e.g. POST, PUT) it will need to both read `request` and write `request.body`. It doesn't have any obvious need for reading or writing `response`.

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

Express static is responsible for serving the static files. It will call `next()` on all requests except those with GET of URLs that map to a file in the local file system. To determine the GET and the URL it will need to read every `request`. If it is a GET of file, it will need to fill in `response` and `response.body` and call `response.send()` to send the response out.

### 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       | <b>never</b> | <b>infrequently</b> |
| accept       | <b>never</b> | <b>often</b>        |
| read/receive | <b>often</b> | <b>often</b>        |
| write/send   | <b>often</b> | <b>often</b>        |
| connect      | <b>often</b> | <b>never</b>        |

## 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.

It is not safe. Although JavaScript runs every function until it returns, the reading of the object (`User.findOne`) and the writing of the object (`function (err, user) {}`) are two different functions so other functions can be interleaved with their execution. It would be possible for another `User.findOne` to execute so we have fetched the object twice and will have one of the updates smashed.

- (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.

It is also not safe for the same reason as in (A). Having the code running in multiple Node.js instances in a scaleout architecture will make it much more likely to hit this.

## 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?

The secret is used to generate the Message Authentication Code (MAC) for the sessionID cookie. This is signing the cookie. This MAC is intended to prevent an attacker from tamper with or forging the sessionID cookie. If the attacker knows the secret they can generate valid MAC so they can try different sessionIDs and if they can discover the sessionID of another active user session, they can hijack it.

## 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.

Even if our web app does use an REST endpoint, our threat model includes an attacker getting ahold the session cookie and launching arbitrary HTTP requests against our web server. The "dead code" isn't really dead and an attacker can exploit any security problems in it.

- (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.

Even if we don't need to worry about "man-in-the-middle", we still need to worry about an attacker getting the session cookie and launching arbitrary HTTPS requests against our web server. An attacker can exploit any security problems in any exposed endpoint.

## 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.

To listen on an event in JavaScript you do: `event.on("EventName", functionToCall)` so the programmer gets to specify the function that is called when the event is emitted. This function can do anything the function passed as a callback can do. There is no loss in power.





## 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.

If you load the web application using the `file:` scheme, the origin of the web app will be `file:.` If you then try to do an XMLHttpRequest to fetch the model data from `localhost:3000` the request will be rejected by the DOM under the single origin policy.

## 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.

Under the man-in-the-middle attack, the single HTTP requests can have its response modified to anything the attacker wants including display things that could mislead the user.

## 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.

With a scale-out architecture requests are load-balanced across a number of instances. If the failure is due to an instance having a problem (e.g. overloaded, crashed, etc.) then immediately trying again could very well land on a different instance that is running fine. With this property a strategy of immediately retrying failed requests can be fruitful.

## 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 Attack

No. DDOS works by exhausting some resource or resources of a target system so legit requests fail. Given that resources such as the network connection can be exhausted, it would be hard for an npm module to prevent this.

### (B) SQL Injection Attack

Yes. SQL Injection works by inputting SQL queries into the application's input fields to obtain protected information. A node module can be used on the server side to build SQL queries that know how to correctly escape input data.

### (C) Cross-site Scripting Attack

No. Cross-site scripting attacks target the browser. Browser thinks the script came from a trusted source and just executes it. A node module on the application side cannot protect against this.