Web Technologies and Full Stack Development

Web security and deployment

The notes for the lesson “Web security and deployment”: chapters 6.8 to 6.9, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

`2 chapters` `Chapters 6.8 to 6.9`

About these notes

These are the notes for the University free n easy video lesson **Web security and deployment** on Web Technologies and Full Stack Development: chapters 6.8 to 6.9, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

6.8 Authentication and web security: sessions, JWT, XSS

Authentication answers one question: who is this user? A session keeps the answer on the server and gives the browser only a random session id in a cookie. A JWT keeps the answer in a signed token that the client sends back. Passwords are stored as bcrypt hashes, never as plain text. XSS is the opposite problem: attacker script runs inside your page and acts as the user. Every login on IRCTC, a UPI app or a college portal depends on these ideas.

You need: Express middleware and the request chain (6.4); user records from MySQL or MongoDB (6.6, 6.7); and HTTP headers (an earlier lesson, HTTP and HTTPS).

Passwords and server-side sessions

Never store a password as plain text. Store a slow hash made by bcrypt. bcrypt adds a random salt and repeats the work for a cost factor, usually 10 to 12 rounds. The salt is stored inside the hash string, so no separate salt column is needed. On login, fetch the stored hash and call `bcrypt.compare(enteredPassword, storedHash)`. It returns true or false. The cost makes one check cheap for the server and millions of guesses expensive for an attacker.

A session keeps the login proof on the server. `express-session` creates a session object, gives it a random id, and sends that id in a cookie. Session data lives in a store: memory for a demo, a database or Redis for production. Each later request carries the cookie; middleware finds the

session and sets `req.session.user`. Logout calls `req.session.destroy()`, so the same cookie stops working.

1. Look up the user row by email or username.
2. Compare the entered password with the stored bcrypt hash.
3. Create the session and set the session cookie.
4. On each later request, load the session from the cookie.

Cookie flags matter. HttpOnly stops JavaScript from reading the cookie, so an XSS bug cannot steal the session id. Secure sends the cookie only over HTTPS. SameSite=Lax or Strict stops another site from making the browser send the cookie on a cross-site request, which reduces CSRF. Use sessions for a browser app on one domain: the server can revoke access at once, and the cookie is small. A session id is a random key to server state, so use at least 32 random bytes and rotate the id after login.

JWT: three parts, signed not encrypted

A JWT has three parts separated by dots: header.payload.signature. The header and payload are base64url-encoded JSON. Base64url is encoding, not encryption: anyone who copies the token can decode and read the payload. The signature is computed with a secret key known only to the server. The server checks the signature and the exp claim on every request; if either fails, it returns 401 Unauthorized.

```
token = base64url(header) + '.' + base64url(payload) + '.' + base64url(HMAC-SHA256(se
```

header names the signing algorithm; payload carries claims such as sub, iat and exp; secret is the server key; HMAC-SHA256 is the signing function; base64url is URL-safe base64 encoding, not encryption.

Standard claims are sub (user id), iat (issued at, seconds since 1970-01-01T00:00:00Z) and exp (expiry, same clock). The client sends the token as `Authorization: Bearer <token>`. Middleware verifies it and sets `req.user`. Services can verify a token without a shared session store, so JWT suits mobile apps and several APIs. Logout is weaker: the client drops the token, but a copied token works until exp. A server-side blacklist of token ids (jti) or short-lived tokens with refresh tokens limits that. Do not store a JWT in localStorage: any XSS script can read it. Keep it in memory or in an HttpOnly cookie.

Choose sessions for a browser app on one domain; choose JWT for mobile clients and multiple services. Neither replaces HTTPS: a token or cookie on plain HTTP can be copied in transit.

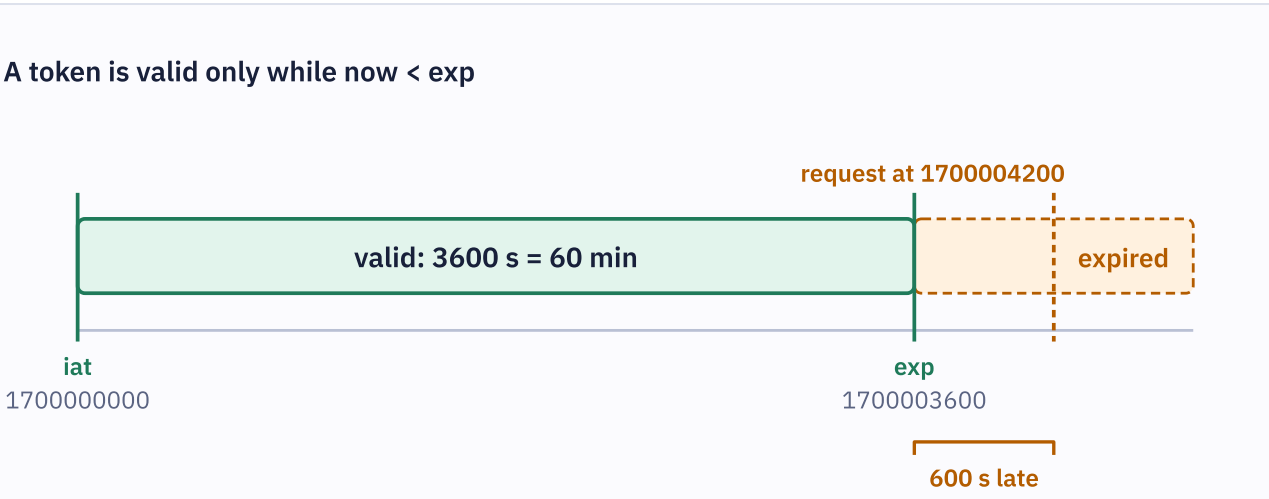

Figure 6.8.1 The token is valid from iat to exp, 3600 s or 60 minutes. The request arrives 600 s after exp, so the server rejects it with 401 Unauthorized.

XSS and the wider security net

XSS (cross-site scripting) happens when attacker-controlled script runs inside your page origin. The script can read the DOM, call your APIs as the user, and steal anything JavaScript can reach. There are three common types. Stored XSS saves the script in your database, for example in a comment, and serves it to every reader. Reflected XSS puts the script in the current request and echoes it back in the response. DOM XSS happens when your own client code writes user text into the page, such as `element.innerHTML = userText`.

Defend on output. Escape text for the context where it appears, or use `textContent` instead of `innerHTML`. Never build HTML by joining user strings. Add a Content-Security-Policy header, for example `default-src 'self'`, so injected inline script is blocked. Validate input on the server as well; client checks can be skipped. The helmet middleware sets several security headers, but it is not a substitute for escaping.

Other attacks are easy to mix up. SQL injection changes a query through unescaped input; use ? stand-ins (see 6.6). CSRF tricks the browser into sending an authenticated request to your site; SameSite cookies and a CSRF token reduce it. HTTPS encrypts the whole exchange, and Secure cookies refuse to travel on HTTP. XSS runs code in your origin; CSRF sends a request from another origin with your cookie.

WORKED EXAMPLE

Question. A JWT is issued with iat = 1700000000 and exp = 1700003600. (a) Find its lifetime in seconds and in minutes. (b) A request arrives at 1700004200. How many seconds after expiry is it, and what should the server return?

1. Lifetime = exp − iat = 1700003600 − 1700000000 = 3600 s.
2. In minutes: 3600 ÷ 60 = 60 min.
3. The token is valid only while now < exp. At now = 1700004200, now − exp = 1700004200 − 1700003600 = 600 s.
4. 600 s = 10 min past expiry, so the server must reject the token with 401 Unauthorized.

Answer. Lifetime 3600 s (60 min); the request is 600 s after expiry, so the server returns 401 Unauthorized.

EXAM TRAPS

- A JWT is signed, not encrypted; base64url decodes to readable JSON, so never put secrets in the payload.
- Storing a JWT in localStorage exposes it to any XSS script; keep it in memory or an HttpOnly cookie.
- A server must check both the signature and exp; decoding the payload alone proves nothing.
- XSS injects script into your page origin; CSRF makes the browser send a request from another site with your cookie.
- bcrypt cost 10-12 is not the salt; the salt is inside the hash and is not secret.

CHECK YOURSELF

– REMEMBER **What does bcrypt.compare(entered, storedHash) return?** 2 marks

true if the entered password matches the hashed password, false otherwise.

– REMEMBER **Name the three dot-separated parts of a JWT.** 2 marks

header, payload, signature.

– UNDERSTAND **Why can a user read a JWT payload but not forge a new signature?** 3 marks

The payload is base64url encoded, not encrypted. The signature needs the secret key with HMAC-SHA256, so changing the payload breaks the signature.

– APPLY **A token has iat = 1000 and exp = 1600. A request arrives at 1700. How many seconds late is it, and what status should the server return?** 4 marks

1700 – 1600 = 100 s late. The exp time has passed, so return 401 Unauthorized.

– ANALYZE **A session cookie has HttpOnly but not SameSite. Which attack does HttpOnly stop, and which remains?** 4 marks

HttpOnly stops XSS from reading the cookie. CSRF can still send the cookie on a cross-site request unless SameSite or a CSRF token is used.

– UNDERSTAND **Give the difference between stored XSS and reflected XSS.** 3 marks

Stored XSS saves the script in the database and serves it to many users; reflected XSS comes in one request and is echoed in that response.

– APPLY **Which cookie flags should an HTTPS-only browser session set?** 3 marks

HttpOnly, Secure, and SameSite=Lax or Strict.

– ANALYZE **Why is bcrypt cost 4 a bad choice even though it is faster?** 6 marks

Low cost makes offline password guessing cheap. Cost 10-12 makes each guess slow, while one login check stays acceptable for the server.

BEYOND THE SYLLABUS

Not for the exam. Industry adds OAuth 2.0 and OpenID Connect so a user can log in through Google or a bank identity provider without sharing a password. Passkeys (WebAuthn) replace passwords with public-key cryptography. Refresh-token rotation and short access tokens reduce the damage of a stolen JWT. Search for OWASP Top 10, Content-Security-Policy, and WebAuthn to go deeper.

6.9 Deployment: Docker, cloud hosting and CI/CD

A Docker image packs your Express app, Node runtime and exact dependencies into one file. A container runs that image the same way on your laptop and on a cloud server. CI/CD turns a git push into install, test, build, push and deploy steps without a human copying files. This chapter shows the Dockerfile order that keeps builds fast, how secrets arrive as environment variables, the main cloud hosting choices, and a pipeline you can trace from push to a container listening on port 3000.

You need: 6.4 for the Express app being shipped and 6.2 for package.json, npm ci and scripts.

Image, container and Dockerfile order

An image is a frozen file system plus a start command. A container is that image running as a process. The image is the recipe; the container is the cooked dish. One image can start many containers, like one class can create many objects. The same image behaves the same on a laptop and a cloud server because it carries Node and the exact packages from package-lock.json.

A Dockerfile lists instructions in order. Each instruction creates a layer, and Docker caches a layer until its inputs change. Read this file from top to bottom:

```
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
EXPOSE 3000
CMD ["node", "server.js"]
```

FROM picks node:20-alpine as the base. WORKDIR /app makes /app the working directory. The first COPY brings only package.json and package-lock.json, so RUN npm ci is cached until those two files change. The second COPY brings your source code; editing server.js does not reinstall packages. EXPOSE 3000 documents the intended port. CMD is the foreground process that keeps the container alive. A .dockerignore file keeps node_modules, .git, .env and logs out of the image.

`docker build -t shop .` builds an image named shop from the current directory. `docker run -p 8080:3000 shop` maps host port 8080 to container port 3000. Host port comes first. Open http://localhost:8080; the app still listens on 3000 inside.

Configuration, Compose and hosting

Passwords and database URLs are configuration, not code. Put them in environment variables at run time: `docker run -e DATABASE_URL=... shop`, or an env_file in Compose. Never write them in the Dockerfile or an ENV line; layers keep old values and anyone who pulls the image can read them.

Docker Compose describes several services in one YAML file. A typical file has app and db services: app builds from ., starts after db, and gets DATABASE_URL pointing at the service name db. Compose starts db before app, but does not wait for the database to be ready; add a health check with a depends_on condition, or make the app retry the connection. db uses a named volume so data survives a container restart. Compose creates one private network, so app reaches db by name, not by localhost.

- IaaS: a VPS with root access; you install Docker and manage updates.
- PaaS: a platform builds from your Git push and runs the container for you.
- Managed database: the provider runs MySQL or MongoDB with backups and failover.
- SaaS: ready software you use, not a place to deploy your code.

In production a reverse proxy such as Nginx listens on ports 80 and 443 and forwards requests to 127.0.0.1:3000. Keep port 3000 off the public internet. Logs go to three places: `docker logs` for app stdout/stderr, Nginx access.log and error.log for requests, and your cloud log viewer if the platform collects them.

The CI/CD pipeline

CI/CD is one automatic path from commit to live container. A CI runner starts on every push.

1. Checkout the commit and run `npm ci`; it installs exactly package-lock.json, the same as local.
2. Run `npm test`. A failing test stops the pipeline before any image is built.
3. Build the image and tag it with the commit SHA: `docker build -t registry.example.com/shop:${SHA} .`.
4. Push that tag to a registry.
5. On the deploy host, pull the tag, inject environment variables, stop the old container, start the new one on port 3000, and check /health.
6. Keep the previous tag. If the health check fails, redeploy the previous tag: that is the rollback.

A tag like latest is not a version; the commit SHA is. A registry stores images (public or private), not source commits like a Git repository. One run takes install 40 s, test 25 s, build 30 s and

deploy 15 s. A manual copy-and-restart can take 20 min.

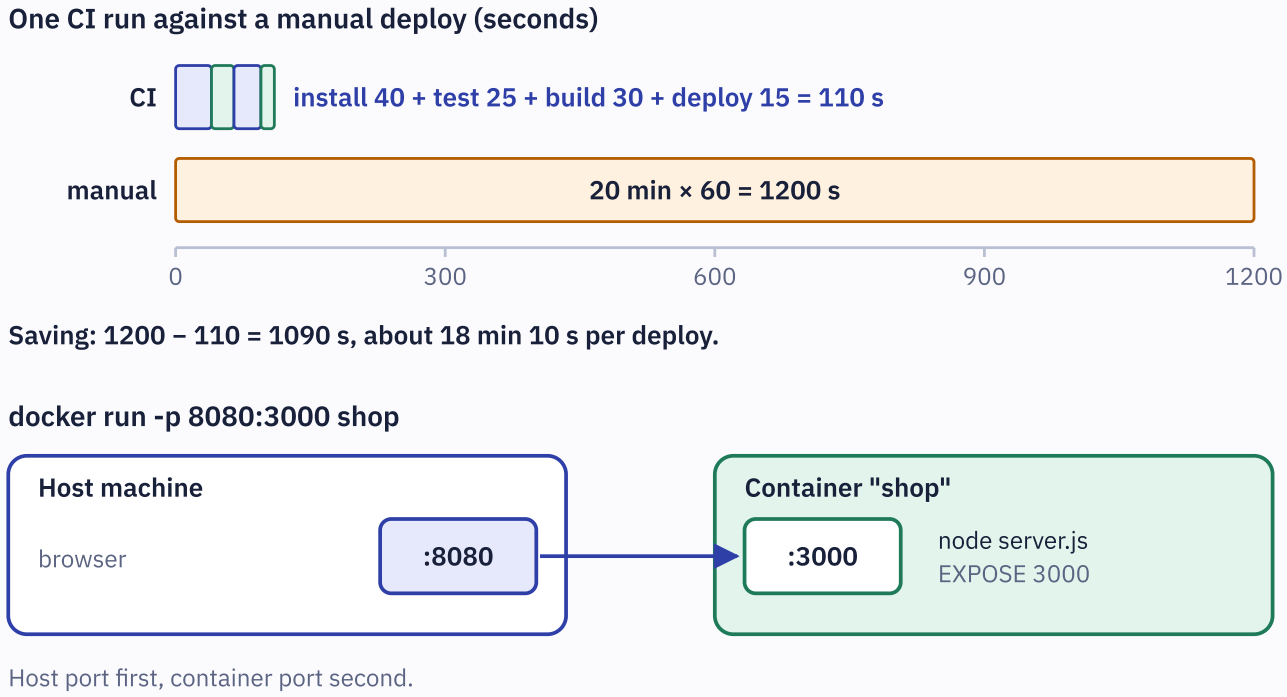

Figure 6.9.1 The four CI stages take 40 + 25 + 30 + 15 = 110 s against 1200 s by hand, saving 1090 s. docker run -p 8080:3000 sends host port 8080 to the app listening on 3000 inside the container.

WORKED EXAMPLE

Question. A shop API listens on port 3000 inside its container. Give the Dockerfile instruction order and the build and run commands to serve it on host port 8080. A CI run takes install 40 s, test 25 s, image build 30 s and deploy 15 s; manual deployment takes 20 min. Find total CI seconds and seconds saved.

Python program.

```
total = 40 + 25 + 30 + 15
manual = 20 * 60
print(total)
print(manual - total)
```

1. Dockerfile order: FROM node:20-alpine, WORKDIR /app, COPY package*.json ./, RUN npm ci, COPY . ., EXPOSE 3000, CMD ["node", "server.js"].
2. Build: docker build -t shop .
3. Run: docker run -p 8080:3000 shop. Host port 8080 comes first, container port 3000 second.
4. CI total = 40 + 25 + 30 + 15 = 110 s.
5. Manual time = 20 min × 60 s/min = 1200 s.
6. Saving = 1200 − 110 = 1090 s.

Dry run

STAGE	SECONDS
install	40
test	25
build	30
deploy	15

Output

```
110
1090
```

Answer. Total CI time is 110 s, and the saving is 1090 s (about 18 min 10 s).

EXAM TRAPS

- Writing COPY . . before RUN npm ci throws away the layer cache on every code change and reinstalls everything.
- Reversing the port mapping: -p 3000:8080 is wrong when the app listens on 3000 and you want host 8080; use -p 8080:3000.
- Putting passwords in the Dockerfile or an ENV line; they stay inside image layers even if a later line deletes them.
- Thinking EXPOSE publishes the port. EXPOSE only documents it; docker run -p or Compose ports is needed.
- Running a daemon in CMD such as a background process; when PID 1 exits, the container stops.

CHECK YOURSELF

– REMEMBER **Define image and container. Which one is running?** 2 marks

An image is a frozen file system plus a start command. A container is that image running as a process.

– UNDERSTAND **Why does the Dockerfile copy package*.json before the source code?** 3 marks

So RUN npm ci is cached until package.json or package-lock.json changes. Source edits then reuse the installed node_modules layer.

– APPLY **An app listens on container port 3000. Write the command to expose it on host port 8080.** 2 marks

docker run -p 8080:3000 shop. Host port 8080 comes first; container port 3000 comes second.

– ANALYZE **npm test fails in the CI pipeline. What happens next, and why is that useful?** 3 marks

The pipeline stops before docker build and push. A broken commit never reaches the registry or deploy host, so production keeps the previous working tag.

– APPLY **A CI run takes install 50 s, test 20 s, build 40 s and deploy 10 s. Find the total seconds.** 2 marks

50 + 20 + 40 + 10 = 120 s.

– ANALYZE **Why is a secret in a Dockerfile dangerous even if the next line deletes the file?** 4 marks

Each instruction is a layer. The secret remains in the earlier layer, and anyone who pulls the image can inspect layers and read it. Use run-time environment variables or a secret store.

– UNDERSTAND **What does Nginx do in front of an app on port 3000?** 3 marks

It listens on 80 and 443, forwards requests to 127.0.0.1:3000, handles TLS, and keeps port 3000 off the public internet.

BEYOND THE SYLLABUS

Not for the exam. Production teams add health checks, blue-green and canary releases, image vulnerability scanning, and Kubernetes to schedule many containers. Search for Docker multi-stage builds, GitHub Actions OIDC to a cloud registry, and Kubernetes readiness probes.

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
JWT	A signed token of the form header.payload.signature; the server checks the signature and trusts the payload without a lookup.	6.8
XSS	Cross-site scripting: attacker JavaScript runs inside your page's origin because your page printed untrusted input.	6.8
Docker image and container	The image is a frozen filesystem plus startup command built from a Dockerfile; the container is one running instance of it.	6.9

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Apply

– APPLY **Implement a login endpoint in Express twice: once with express-session and once with a JWT. Show where the credential is stored in each case and how a protected route checks it.** 15 marks

bcrypt hash comparison at login; session: store user id in req.session, cookie carries the id, server-side store; JWT: sign with a secret, send as Authorization: Bearer, verify in middleware; cookie options httpOnly, secure, sameSite; logout: destroy session versus client discards token; revocation is easy for sessions, hard for JWT without a blocklist.

Analyze and evaluate

– EVALUATE **A teammate stores the JWT in localStorage because it is simple. Evaluate this choice against an HttpOnly cookie, considering XSS, CSRF and mobile clients.** 12 marks

localStorage is readable by any script in the origin, so XSS steals the token; HttpOnly blocks document.cookie but the browser still sends the cookie, so CSRF appears; sameSite and CSRF tokens counter that; mobile apps have no cookie jar, so header tokens fit better; short expiry plus refresh token limits damage; escaping output and a Content-Security-Policy remove the XSS root cause.

– EVALUATE **Explain how Docker and a CI/CD pipeline together remove the phrase "it works on my machine". Describe the stages from git push to a running container and evaluate the risk of deploying the :latest tag.** 12 marks

Dockerfile pins the runtime, OS libs and start command; image built once, container identical on laptop and server; pipeline stages: install, lint, test, build image, push to registry, deploy; secrets injected as environment variables, not baked into the image; :latest is mutable, so a rollback is ambiguous and two servers may run different code; tag by commit SHA and keep the lockfile for exact dependencies.

Glossary (terms used in this lesson)

bcrypt — A deliberately slow password hashing function that adds a random salt, so identical passwords give different hashes.

CSRF — Cross-site request forgery: another site makes the browser send a request that carries the victim's cookie automatically.

Dockerfile	The recipe file listing base image, copied files, dependency install and start command used to build an image.
Environment variable	Configuration passed to a process at run time, such as PORT or DATABASE_URL, so the same image works in every environment.
HttpOnly cookie	A cookie the browser sends with requests but hides from document.cookie, so injected script cannot read the session id.
node_modules	The folder where npm installs packages; bare-name require() searches it upward from the current file.

Sources and further reading

Textbooks

- Mario Casciaro, Luciano Mammino, *Node.js Design Patterns*: Chapters on the Node platform, the event loop, streams and modules.
- Ethan Brown, *Web Development with Node and Express*: Routing, middleware, static files, REST APIs and deployment.
- Evan M. Hahn, *Express in Action*: Express routing, middleware and building APIs.
- Shannon Bradshaw, Eoin Brazil, Kristina Chodorow, *MongoDB: The Definitive Guide*: Documents, collections, queries and the MongoDB shell.

Open web resources

Each link below was opened by the book script on 11 Nov 2026 and returned a page; the pages were not read line by line.

- Node.js API documentation: nodejs.org/docs/latest/api
- Express web framework documentation: expressjs.com
- npm documentation: docs.npmjs.com
- MongoDB manual: www.mongodb.com/docs/manual
- OWASP Top Ten web application security risks: owasp.org/www-project-top-ten
- Docker get started guide: docs.docker.com/get-started

Syllabus: the University free n easy universal syllabus, built from top universities' syllabi.

Built for the University free n easy YouTube channel from the universal syllabus of Web Technologies and Full Stack Development (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.