LECTURE 25

REST API Development and Authentication

Designing clean, consistent endpoints — then locking them down with sessions, JWTs, and bcrypt.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. REST principles: resources, URIs, HTTP verbs, and statelessness
2. Designing CRUD endpoints with consistent JSON response shapes
3. Session-based vs. token-based authentication (JWT)
4. Hashing passwords safely with bcrypt
5. Protecting routes with authentication middleware and role-based access control

What Is REST?

1. REST (Representational State Transfer): design principles for APIs built around resources — the "nouns" of your app (a user, a post, an order)

2. Operations on resources go through URLs and standard HTTP methods

3. No custom action per operation — avoid /getUser or /deleteUserById

Resources and URIs

Resource	Collection URI	Single Resource URI
Users	/api/users	/api/users/:id
Posts	/api/posts	/api/posts/:id
Comments (nested)	/api/posts/:postId/comments	/api/posts/:postId/comments/:id

Plural Nouns, No Verbs

TIP

Use plural nouns for resource names (/users, not /user), and never put a verb in the URI (avoid /getUsers or /deleteUser/5) — the HTTP method already communicates the action.

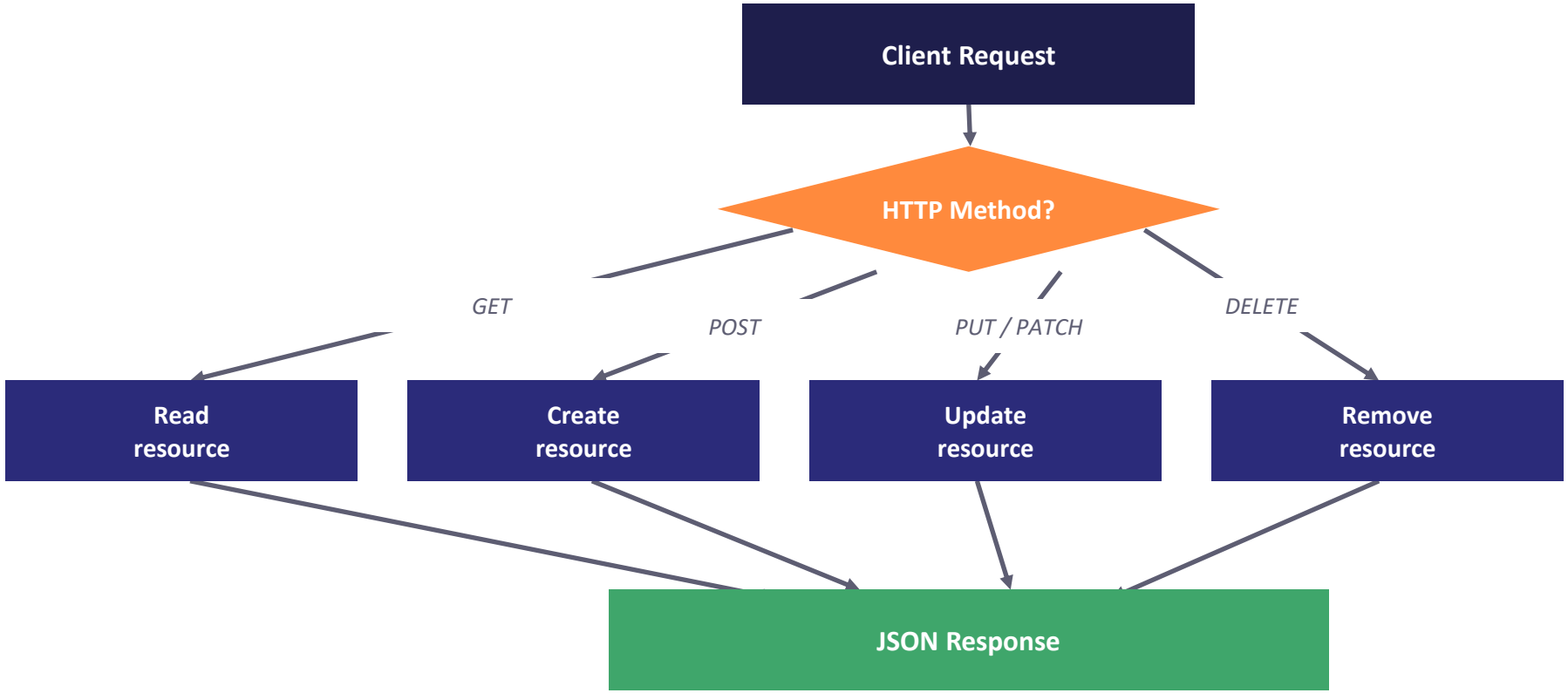
HTTP Verbs: What to Do With a Resource

Method	Meaning	Example
GET	Read a resource, never changes data	GET /api/users/5
POST	Create a new resource	POST /api/users
PUT	Replace a resource entirely	PUT /api/users/5
PATCH	Partially update a resource	PATCH /api/users/5
DELETE	Remove a resource	DELETE /api/users/5

Statelessness

1. Each request must carry everything the server needs to process it — an auth token, for example
2. The server should not rely on remembering anything about the client from a previous request
3. This is why token-based auth fits REST APIs especially well — though session-based APIs remain common, especially for server-rendered apps

Every Request, Routed by Method



GET: List and Read One

```
// GET /api/posts — list
router.get("/posts", async (req, res) => {
  const posts = await Post.find();
  res.status(200).json({ data: posts });
});

// GET /api/posts/:id — read one
router.get("/posts/:id", async (req, res) => {
  const post = await Post.findById(req.params.id);
  if (!post) return res.status(404).json({ error: "Post not found." });
  res.status(200).json({ data: post });
});
```

POST, PATCH, DELETE

```
// POST /api/posts — create
router.post("/posts", async (req, res) => {
  try {
    const post = await Post.create(req.body);
    res.status(201).json({ data: post });
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

// DELETE /api/posts/:id
router.delete("/posts/:id", async (req, res) => {
  const post = await Post.findByIdAndDelete(req.params.id);
  if (!post) return res.status(404).json({ error: "Post not found." });
  res.status(204).send(); // No Content
});
```

Keep the Response Shape Consistent

NOTE

Successful responses wrap the payload in { data: ... }, errors use { error: ... }, and the status code always matches what happened (200 OK, 201 Created, 204 No Content, 404 Not Found, 400 Bad Request). One shape, every endpoint.

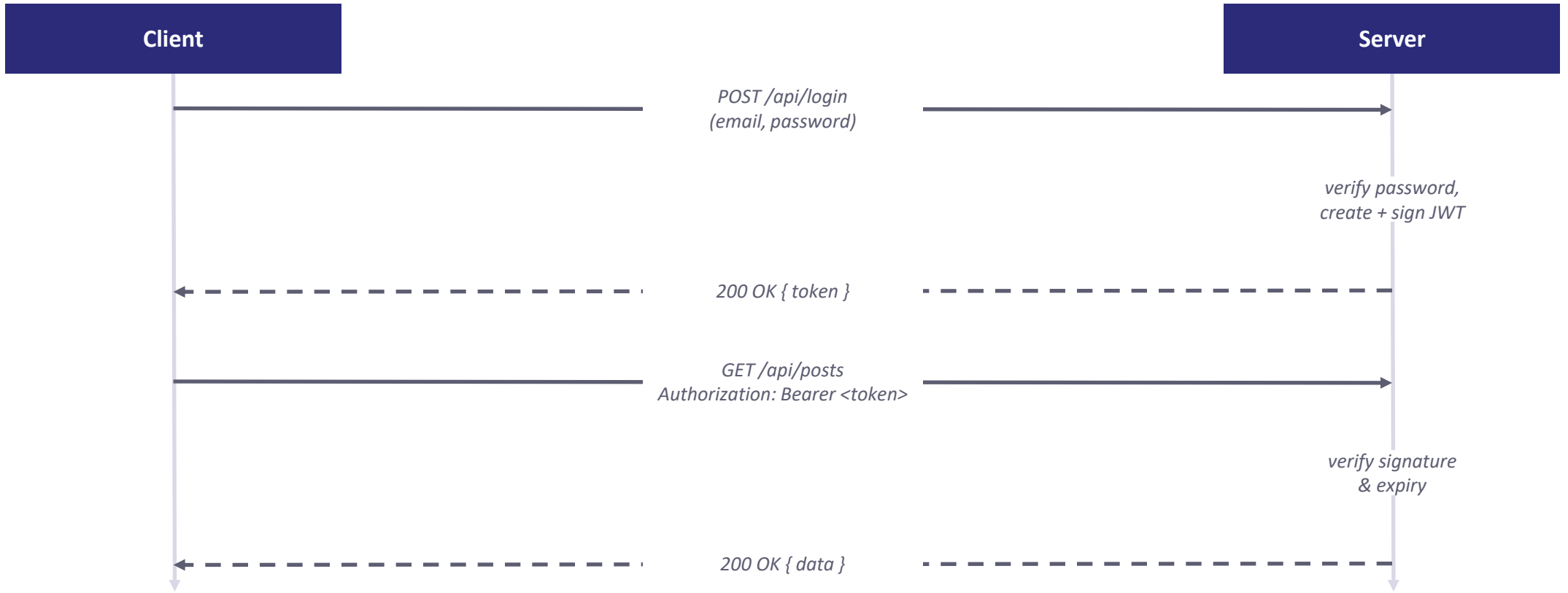
Session-Based Authentication (Recap)

1. After login, the server creates a session (data stored server-side) and sends a session ID in a cookie
2. On every future request, the browser attaches that cookie automatically and the server looks up the session
3. Storage lives on the server; the client only holds a small ID
4. Fits naturally with server-rendered (SSR) apps — but harder to scale across multiple servers sharing one session store

Token-Based Authentication (JWT)

1. The server issues the client a self-contained token after login — no server-side memory required
2. The client stores the token and sends it back on every request, usually in an Authorization header
3. The server verifies the token's authenticity without a storage lookup
4. A JWT is a string of three dot-separated parts: header (metadata), payload (data like user ID/role), and signature (proves it wasn't tampered with)

Issuing and Verifying a JWT



The client stores the token after login and resends it on every request — the server never has to remember anything between requests.

Issuing a Token at Login

```
npm install jsonwebtoken

function issueToken(user) {
  return jwt.sign(
    { userId: user._id, role: user.role },
    process.env.JWT_SECRET,
    { expiresIn: "1h" }
  );
}

app.post("/api/login", async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  const validPassword = user && (await bcrypt.compare(password, user.passwordHash));
  if (!validPassword) return res.status(401).json({ error: "Invalid email or password." });
  res.status(200).json({ data: { token: issueToken(user) } });
});
```

Verifying a Token

```
function verifyToken(req, res, next) {
  const authHeader = req.headers.authorization; // "Bearer eyJhbGciOi..."
  const token = authHeader && authHeader.split(" ")[1];
  if (!token) return res.status(401).json({ error: "No token provided." });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.userId = decoded.userId;
    req.userRole = decoded.role;
    next();
  } catch (err) {
    return res.status(401).json({ error: "Invalid or expired token." });
  }
}
```

Refreshing Tokens

1. A short-lived token (like the 1-hour one above) keeps expiring — apps often issue a second, longer-lived refresh token alongside it
2. When the access token expires, the client sends the refresh token to a dedicated endpoint to get a new one
3. This avoids forcing the user to log in again

The Refresh Endpoint

```
app.post("/api/refresh", async (req, res) => {
  const { refreshToken } = req.body;
  try {
    const decoded = jwt.verify(refreshToken, process.env.JWT_REFRESH_SECRET);
    const newAccessToken = jwt.sign(
      { userId: decoded.userId }, process.env.JWT_SECRET, { expiresIn: "1h" }
    );
    res.status(200).json({ data: { token: newAccessToken } });
  } catch (err) {
    res.status(401).json({ error: "Invalid refresh token, please log in again." });
  }
});
```

COMPARISON

Session-Based vs. Token-Based (JWT)

	Session-Based	Token-Based (JWT)
Identity lives	Server-side session store	Self-contained in the token
Client holds	Small session ID (cookie)	The full token
Server lookup needed?	Yes, every request	No — signature is enough
Fits well with	Server-rendered (SSR) apps	REST APIs, mobile apps, SPAs
Revoking access early	Easy — delete the session	Harder — valid until expiry

COMPARISON

Neither Is Universally "Better"

NOTE

Session-based auth remains an excellent, simple choice for traditional server-rendered applications, while JWTs shine when your API is consumed by multiple separate clients (a React SPA, a mobile app) that aren't using browser cookies at all.

Never Store Plain-Text Passwords

1. If your database is ever compromised, plain-text passwords hand the attacker every user's real password immediately

2. Instead, store a hash — the output of a one-way function that scrambles the password so it can't practically be reversed

3. bcrypt is a widely trusted password-hashing algorithm — deliberately slow, and automatically salted (random data so identical passwords hash differently)

Hashing and Comparing With bcrypt

```
npm install bcrypt

// When a user registers:
const passwordHash = await bcrypt.hash(password, 10); // 10 = salt rounds
const user = await User.create({ email, passwordHash });

// When a user logs in:
const isMatch = await bcrypt.compare(plainTextPassword, user.passwordHash);
```

bcrypt.compare() re-hashes the submitted password using the same salt and checks if the results match — it never "un-hashes" anything, because that's not mathematically possible.

Passwords Are Radioactive

WARNING

Never log, email, or send a user's plain-text password anywhere, even temporarily — and never implement "forgot password" by emailing the old password back, since you never stored it. Send a time-limited reset link/token instead.

Combining Middleware for RBAC

```
function requireRole(role) {
  return function (req, res, next) {
    if (req.userRole !== role) {
      return res.status(403).json({ error: "Forbidden: insufficient permissions." });
    }
    next();
  };
}

app.delete("/api/users/:id", verifyToken, requireRole("admin"), async (req, res) => {
  await User.findByIdAndDelete(req.params.id);
  res.status(204).send();
});
```

verifyToken runs first (confirms WHO the user is), then requireRole("admin") runs (confirms they're ALLOWED to do this). Either check failing stops the request there.

Two Middleware, Two Jobs

TIP

This uses the same middleware-chaining pattern from the Middleware lecture — authentication and authorization are two of the most common real-world uses for custom middleware.

KEY TAKEAWAYS

Lecture 25 in Six Points

1. REST organizes an API around resources identified by URIs, using standard HTTP verbs instead of custom action names.
2. REST APIs are ideally stateless — each request carries what the server needs, rather than relying on server memory.
3. Design CRUD endpoints with consistent JSON response shapes ({ data } / { error }) and accurate HTTP status codes.
4. Session-based auth keeps identity on the server; token-based auth (JWT) puts a signed, self-contained token in the client's hands.
5. JWTs are issued at login, verified on protected routes, and often paired with a longer-lived refresh token.
6. Always hash passwords with bcrypt (never plain text); chain authentication middleware with role-based access control on protected routes.