Recitation 6b: Secrets, mTLS, and Authentication/Authorization

Hardening the Distributed Library System

Michael J. May

1 December 2026

Prerequisite. This recitation continues directly from the previous recitation creating a distributed library services with customer, librarian, and checkin services. You should already have the `customer-service`, `librarian-service`, `checkin-service`, and `mongo` containers running together on `library-net` via Docker Compose, using Gradle's Kotlin DSL with a `libs/` directory alongside each application jar. Everything below builds on that project directly.

Background. In the previous recitation, we got the four containers to talk to each other. It didn't ask one hard question: *who else can talk to the containers?* Right now, any process that can reach `library-net` can open an unauthenticated, unencrypted connection straight to MongoDB and read or rewrite every book and every customer record. Our three Java apps run for anyone who launches the container. There is no login at all. Let's fix both of those problems now:

Transport security Every connection to MongoDB is mutual TLS (mTLS): the database proves its identity to the app, and the app proves its identity to the database, using certificates signed by a Certificate Authority (CA) we control.

Database-layer authorization Each service's certificate is its MongoDB username (via the `MONGODB-X509` mechanism). Each one is granted only a narrow set of collection privileges it needs.

Application-layer authentication Human users (customers, librarians) log in with a username and a password that is never stored in plaintext.

Application-layer authorization A logged-in user's role determines which commands they can run, independent of the database-layer grants.

Secret hygiene None of the secrets appears in a Docker image layer, a Compose file committed to git, an environment variable visible to docker inspect, or a compiled JAR.

1 Part 0: What "Secret" Means Here, and Our Threat Model

Before writing code, let's set out a vocabulary that will drive the decisions.

Config vs. Secret. A piece of information is *config* if disclosing it does not help an attacker (e.g., the MongoDB port number, a service's hostname, the name of a collection). It is a *secret* if disclosing it does (e.g., a private key, a password, a token, a database connection string that embeds a password). Configs are safe to bake into an image or commit to git. Secrets are not because the image itself might be handed to someone (a customer, a CI runner, a public registry) and because every layer of a Docker image is individually inspectable, even after a later layer "deletes" the file.

Our goals for this recitation, things an attacker must not be able to do:

1. Read or tamper with traffic between an app and MongoDB by joining the Docker network (mitigated by mTLS).
2. Connect to MongoDB directly and run arbitrary queries, even from inside `library-net` (mitigated by requiring a CA-signed client certificate *and* an authorized MongoDB user for every connection).
3. Recover any private key, password, or token by inspecting a shipped Docker image, a compiled JAR, a Compose file in git, or `docker inspect` output (mitigated by runtime-only secret injection, covered in Part 2).
4. Use `customer-service`'s database access to modify things only `librarian-service` should be able to touch, even if a bug in `customer-service` were exploited (mitigated by least-privilege roles at the database layer, covered in Part 3).
5. Use any app without a valid human login, or perform an action a logged-in user's role does not permit (mitigated by application-layer authentication and authorization, Parts 5–6).

Out of scope. This recitation does not cover securing the host machine itself, protecting against a fully compromised Docker daemon, or production-grade secret managers (HashiCorp Vault, AWS Secrets Manager, Docker Swarm/Kubernetes secrets). File-based Compose secrets are the right tool for *this* recitation and for local development; Part 8 discusses how the same design generalizes to those production systems without any application code changes.

2 Part 1: Building a Private Certificate Authority

Mutual TLS needs three kinds of certificates, all signed by a Certificate Authority (CA) that both sides trust: a **server certificate** for `mongo`, and one **client certificate per service**. Each client certificate's Subject (CN) will double as that service's MongoDB username.

Step 1.1: The certificate generation script

Create `generate-certs.sh` in the `library-system/` root directory. Make sure it's not inside any service folder, so it won't accidentally end up in a Docker build context). Note that this is in Linux format, so you'll need to run the script in WSL2 or a Linux Ubuntu image:

```
#!/usr/bin/env bash
set -euo pipefail

OUT=secrets
SERVICES="customer-service librarian-service checkin-service"

mkdir -p "$OUT/ca" "$OUT/mongo"
for svc in $SERVICES; do mkdir -p "$OUT/$svc"; done

# ---- 1. Root CA -------------------------------------------------------
# In a real deployment this key lives offline, e.g. on an HSM or an air-gapped
# machine, and is used only to sign new certificates, never by a running service.
openssl genrsa -out "$OUT/ca/ca.key" 4096
openssl req -x509 -new -nodes -key "$OUT/ca/ca.key" -sha256 -days 3650 \
  -subj "/O=SE424-Library/CN=Library Lab Root CA" \
  -out "$OUT/ca/ca.crt"

# Java needs the CA certificate wrapped as a PKCS12 truststore, plus a
# password for that truststore. Since a truststore holds only PUBLIC
# certificates, its password protects integrity, not confidentiality --
# a fixed value is acceptable, but we still keep it in a file, not in code.
echo -n "truststore-not-a-secret" > "$OUT/ca/truststore-password.txt"
keytool -importcert -noprompt -alias library-ca \
  -file "$OUT/ca/ca.crt" \
  -keystore "$OUT/ca/truststore.p12" -storetype PKCS12 \
  -storepass:file "$OUT/ca/truststore-password.txt"

# ---- 2. Server certificate for mongo ----------------------------------
openssl genrsa -out "$OUT/mongo/server.key" 2048
openssl req -new -key "$OUT/mongo/server.key" \
  -subj "/O=SE424-Library/CN=mongo" \
  -out "$OUT/mongo/server.csr"
openssl x509 -req -in "$OUT/mongo/server.csr" \
  -CA "$OUT/ca/ca.crt" -CAkey "$OUT/ca/ca.key" -CAcreateserial \
  -days 825 -sha256 \
  -extfile <(printf "subjectAltName=DNS:mongo") \
  -out "$OUT/mongo/server.crt"
cat "$OUT/mongo/server.key" "$OUT/mongo/server.crt" > "$OUT/mongo/server.pem"

# ---- 3. One client certificate per service ----------------------------
for svc in $SERVICES; do
  openssl genrsa -out "$OUT/$svc/client.key" 2048
  openssl req -new -key "$OUT/$svc/client.key" \
    -subj "/O=SE424-Library/OU=services/CN=$svc" \
    -out "$OUT/$svc/client.csr"
  openssl x509 -req -in "$OUT/$svc/client.csr" \
    -CA "$OUT/ca/ca.crt" -CAkey "$OUT/ca/ca.key" -CAcreateserial \
    -days 825 -sha256 \
    -out "$OUT/$svc/client.crt"
```

```
50
51    # A random, per-service keystore password -- generated once, never typed
52    # by a human, never reused across services.
53    openssl rand -base64 24 > "$OUT/$svc/keystore-password.txt"
54    openssl pkcs12 -export \
55      -in "$OUT/$svc/client.crt" -inkey "$OUT/$svc/client.key" \
56      -certfile "$OUT/ca/ca.crt" -name "$svc" \
57      -passout "file:$OUT/$svc/keystore-password.txt" \
58      -out "$OUT/$svc/client.p12"
59  done
60
61  echo "Done. Everything under $OUT/ is generated locally and must noot be
        committed."
```

Run it once from `library-system/`:

```
chmod +x generate-certs.sh
./generate-certs.sh
```

Step 1.2: Keep the files out of git and out of any image

Create (or extend) `.gitignore` in `library-system/`:

```
secrets/
*.key
*.p12
*.pem
.env
```

Create a `.dockerignore` at the root of every service folder (i.e., `customer-service/.dockerignore`, and the same file in the other two service folders):

```
secrets/
*.key
*.p12
*.pem
.env
.git
```

Why Both Files

`.gitignore` stops you from committing the secrets directory. It does not stop `docker build` from reading it: the build context is whatever files are on disk, tracked by git or not. `.dockerignore` is the file that protects a build. Without it, a careless `COPY . .` in a Dockerfile would ship `secrets/` into an image layer, even though it was never committed to version control. You need both.

Checkpoint: Part 1

Run `ls secrets/customer-service/`. You should see `client.key`, `client.crt`, `client.p12`, and `keystore-password.txt`. Open `keystore-password.txt` and confirm it is a random string, not something predictable like `"changeit"` or `"password"`.

3 Part 2: Never Let a Secret Touch an Image Layer

Before wiring TLS into Mongo or the apps, let's establish the discipline that makes the rest of this recitation meaningful: a secret that ends up inside a built image is compromised forever, even if you later remove it in a subsequent instruction.

Step 2.1: See the problem yourself

Let's make a fake secret and put it by accident into the image, but then immediately remove it. Run the following command inside the customer-service/ directory.

```
echo Leaked key! > demo.key
```

Temporarily add this line to `customer-service/Dockerfile`, right after the `FROM eclipse-temurin...` line in the runtime stage:

```
COPY ./demo.key /tmp/leaked.key
RUN rm /tmp/leaked.key
```

Before you build it, delete the .dockerignore file you made for the customer-service module. Then build it and inspect the layers *without ever running the container*:

```
docker build -t leak-demo .
docker history --no-trunc leak-demo | grep -i leaked
```

You'll see the `COPY` instruction that added the key as its own layer even though the next instruction deleted the file. Anyone who receives this image by pulling it from a registry or by running `docker save leak-demo | tar -xO` can extract that layer and recover the private key. Remove the demo.key file, replace the .dockerignore file you deleted, and remove the two lines we added. Rebuild. We won't use this pattern anywhere else in the recitation.

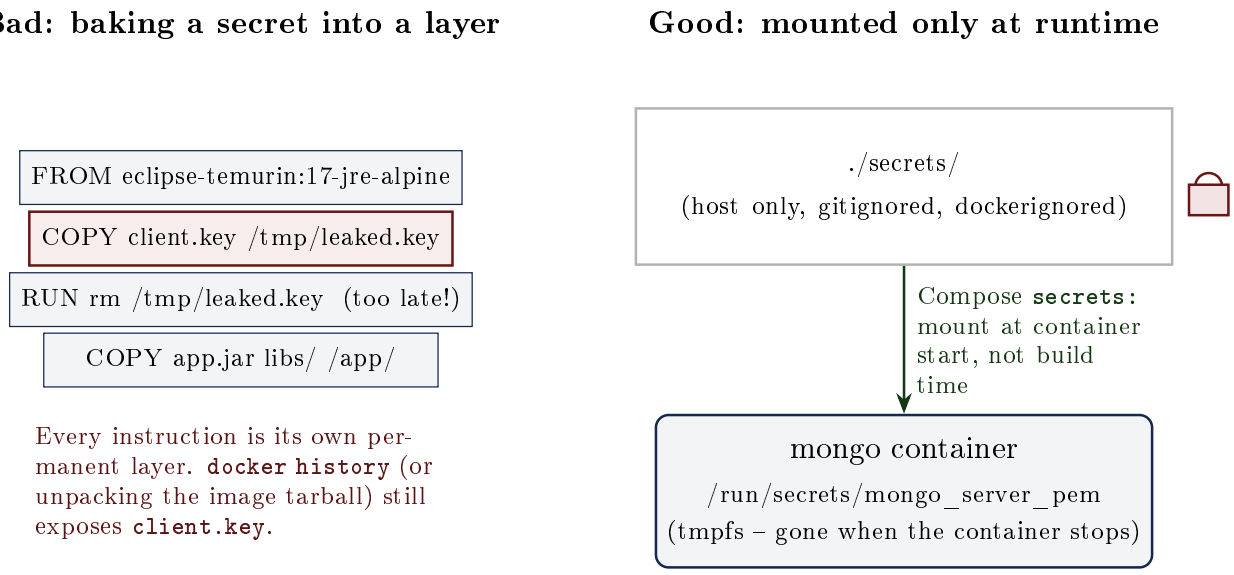


Figure 1: The same secret material handled two ways. On the left, a `COPY` instruction makes the key part of the image's permanent history. On the right, Compose mounts the file into the running container's tmpfs at `/run/secrets/`, where it never touches an image layer and disappears when the container stops.

Step 2.2 — Declare secrets in Compose

At the top level of `docker-compose.yml` (a sibling of `services:`), declare all of the files this recitation needs as a named secret:

```
secrets:
  ca_cert:
    file: ./secrets/ca/ca.crt
  ca_truststore:
    file: ./secrets/ca/truststore.p12
  ca_truststore_password:
    file: ./secrets/ca/truststore-password.txt
  mongo_server_pem:
    file: ./secrets/mongo/server.pem
  mongo_root_username:
    file: ./secrets/mongo_root_username.txt
  mongo_root_password:
    file: ./secrets/mongo_root_password.txt
  customer_client_p12:
    file: ./secrets/customer-service/client.p12
  customer_keystore_password:
    file: ./secrets/customer-service/keystore-password.txt
  librarian_client_p12:
    file: ./secrets/librarian-service/client.p12
  librarian_keystore_password:
    file: ./secrets/librarian-service/keystore-password.txt
  checkin_client_p12:
    file: ./secrets/checkin-service/client.p12
  checkin_keystore_password:
    file: ./secrets/checkin-service/keystore-password.txt
  librarian_bootstrap_password:
    file: ./secrets/librarian_bootstrap_password.txt
```

Generate the two remaining files this references:

```
echo -n "root" > secrets/mongo_root_username.txt
openssl rand -base64 24 > secrets/mongo_root_password.txt
openssl rand -base64 18 > secrets/librarian_bootstrap_password.txt
```

A service opts into a secret by listing its name under its own `secrets:` key; Compose mounts it read-only at `/run/secrets/<name>` inside that container only. Nothing is mounted anywhere it wasn't requested. We wire these into each service in the parts that follow.

Checkpoint: Part 2

Run `docker build -t leak-demo customer-service/` **after** you removed the two temporary lines from Step 2.1, then run `docker history -no-trunc leak-demo`. Confirm there is no trace of any `.key`, `.p12`, or `.pem` filename anywhere in the output. This is the check you should run on *every* image before it ships, and we return to it as a final step in Part 8.

4 Part 3 — Configuring MongoDB for mTLS and Least-Privilege Roles

Step 3.1 — A mongod configuration file

Create `mongo-config/mongod.conf`:

```
net:
  tls:
    mode: requireTLS
    certificateKeyFile: /run/secrets/mongo_server_pem
    CAFile: /run/secrets/ca_cert
    allowConnectionsWithoutCertificates: false
security:
  authorization: enabled
setParameter:
  authenticationMechanisms: MONGODB-X509,SCRAM-SHA-256
```

`requireTLS` means every connection — including ones from inside `library-net` — must be TLS, and `allowConnectionsWithoutCertificates: false` means the client must present a certificate signed by our CA just to complete the handshake, before any authentication even happens. `authorization: enabled` turns on MongoDB's access control, and listing both mechanisms lets us use passwordless `MONGODB-X509` identity for our three services while still allowing a traditional username/password (`SCRAM-SHA-256`) for the one human-operated administrative account we bootstrap next.

Step 3.2 — Bootstrap the root user without ever typing its password

The official `mongo` image supports the same `_FILE` convention we'll use everywhere in this recitation: point an environment variable at a file, and the entrypoint script reads the file itself. Update the `mongo` service in `docker-compose.yml`:

```
services:
  mongo:
    image: mongo:7
    container_name: mongo
    command: ["mongod", "--config", "/etc/mongo/mongod.conf"]
    environment:
      MONGO_INITDB_ROOT_USERNAME_FILE: /run/secrets/mongo_root_username
      MONGO_INITDB_ROOT_PASSWORD_FILE: /run/secrets/mongo_root_password
    secrets:
      - ca_cert
      - mongo_server_pem
      - mongo_root_username
      - mongo_root_password
    volumes:
      - mongo-data:/data/db
      - ./mongo-config/mongod.conf:/etc/mongo/mongod.conf:ro
      - ./mongo-init:/docker-entrypoint-initdb.d:ro
    networks:
      - library-net
```

Notice what did *not* change from the previous recitation: we still use a named volume for `/data/db` and a bind mount for `/docker-entrypoint-initdb.d`. We are layering security on top of the same volume architecture, not replacing it.

The root username and password never appear in `docker-compose.yml`, never appear in `docker inspect mongo` (unlike a plain `environment: MONGO_INITDB_ROOT_PASSWORD: hunter2`, which *would* show up there), and never appear in a shell history, because no human

ever types them — the entrypoint script reads the files directly.

Step 3.3 — Create one narrow role and one x.509 user per service

Because `docker-entrypoint-initdb.d` scripts run *after* the root user is created and are already authenticated as root, we can create our least-privilege roles and identities declaratively, with no manual `mongosh` session required. Create `mongo-init/01-create-app-roles.js`:

```
db = db.getSiblingDB("librarydb");

// Least privilege: each role can touch only the collections and actions
// that service genuinely needs, and nothing else.
db.createRole({
  role: "customerServiceRole",
  privileges: [
    { resource: { db: "librarydb", collection: "books" }, actions: ["find", "update"] },
    { resource: { db: "librarydb", collection: "users" }, actions: ["find"] }
  ],
  roles: []
});

db.createRole({
  role: "librarianServiceRole",
  privileges: [
    { resource: { db: "librarydb", collection: "books" }, actions: ["find", "insert", "update"] },
    { resource: { db: "librarydb", collection: "users" }, actions: ["find", "insert"] }
  ],
  roles: []
});

db.createRole({
  role: "checkinServiceRole",
  privileges: [
    { resource: { db: "librarydb", collection: "books" }, actions: ["find", "update"] }
  ],
  roles: []
});

// Each client certificate's Subject DN becomes a MongoDB username in the
// special $external database -- there is no password to store or leak.
var external = db.getSiblingDB("$external");

external.createUser({
  user: "CN=customer-service,OU=services,O=SE424-Library",
  roles: [{ role: "customerServiceRole", db: "librarydb" }]
});
external.createUser({
  user: "CN=librarian-service,OU=services,O=SE424-Library",
  roles: [{ role: "librarianServiceRole", db: "librarydb" }]
});
external.createUser({
  user: "CN=checkin-service,OU=services,O=SE424-Library",
  roles: [{ role: "checkinServiceRole", db: "librarydb" }]
});
```

Notice `checkinServiceRole` cannot `insert`. The checkin kiosk should only ever flip an existing book's status back to `AVAILABLE`; it has no legitimate reason to create new book records.

If a bug (or a compromise) in `checkin-service` tried to insert a document, MongoDB itself would reject it, independently of anything the application code does or fails to check. This is defense in depth: the database enforces the same boundary the application is supposed to enforce, so one bug is not enough to cross it.

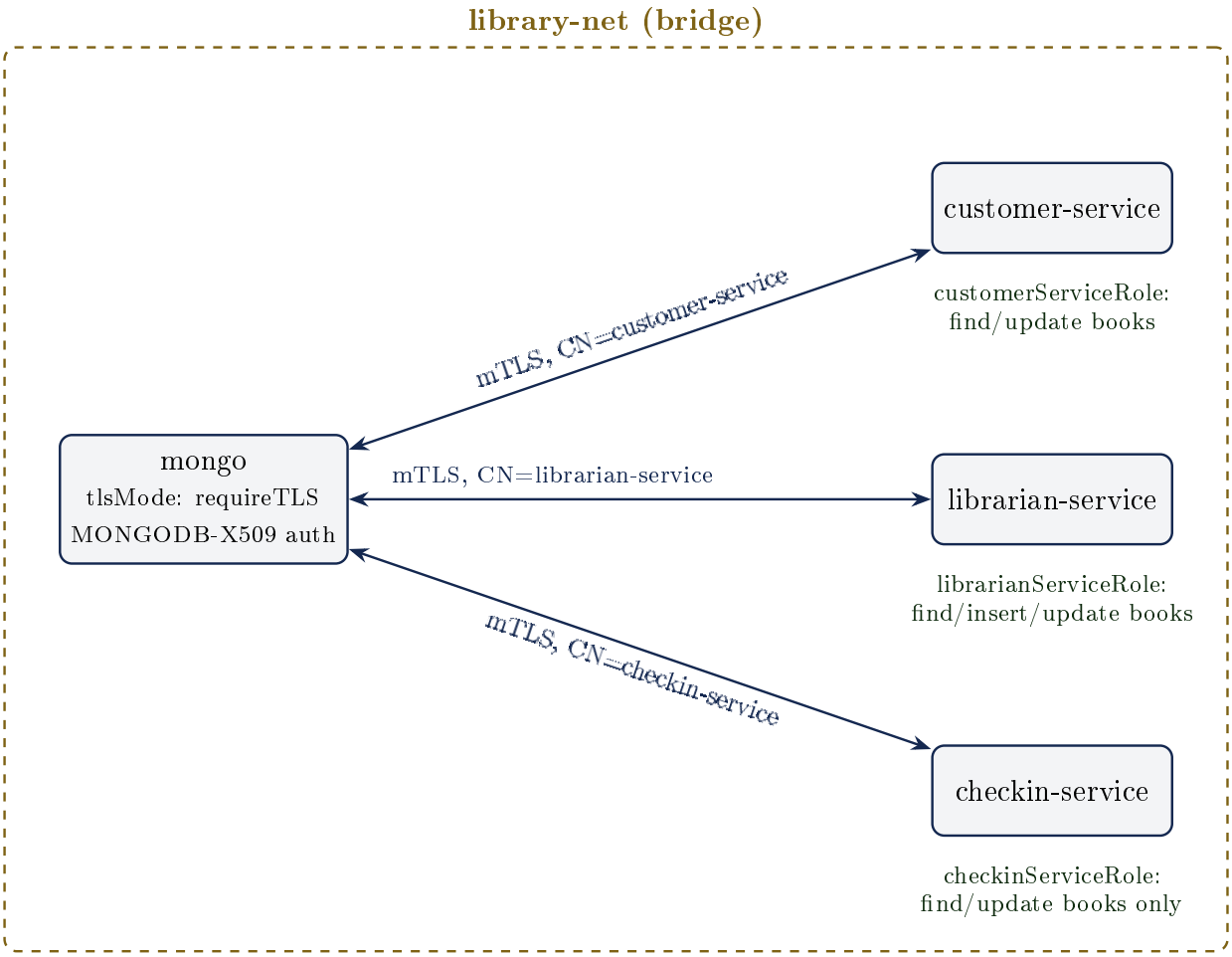


Figure 2: Every connection to `mongo` is mutual TLS, and every service's certificate CN is also its MongoDB username. Each username maps to a role that grants only the specific collection actions that service needs — database-layer authorization, independent of whatever the application code does.

Checkpoint: Part 3

Run `docker compose up -d mongo` against a clean volume (`docker compose down -v` first), then check the logs: `docker compose logs mongo`. Confirm you see MongoDB report that it is listening with TLS enabled, and that the init script ran without errors. Do *not* proceed to Part 4 until this is clean — a broken `mongod.conf` will make every later step fail in confusing ways.

5 Part 4 — Teaching the Java Apps to Speak mTLS

Every service needs the same two pieces of plumbing: a way to read a secret file named by an environment variable, and a way to turn our PKCS12 keystore and truststore into an `SSLContext`. We write each once and copy the file into all three service source trees, the same pattern used before for the small model classes.

Step 4.1 — SecretsLoader.java

Create `customer-service/src/main/java/lib/SecretsLoader.java` (and the identical file in the other two services):

```java
package lib;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

/**
 * Resolves a runtime secret the same way the official Docker images do:
 * read the VALUE from the file named by a "<VAR>_FILE" environment
 * variable. The secret's contents are never stored in an environment
 * variable themselves, so they never show up in `docker inspect`, a
 * process listing, or a crash log that happens to dump the environment.
 */
public final class SecretsLoader {
    private SecretsLoader() {}

    public static String readFileSecret(String envVarName) {
        String path = System.getenv(envVarName);
        if (path == null) {
            throw new IllegalStateException("Missing required environment
                variable: " + envVarName);
        }
        try {
            return Files.readString(Path.of(path)).strip();
        } catch (IOException e) {
            throw new IllegalStateException("Could not read secret file at "
                + path, e);
        }
    }
}
```

Step 4.2 — MongoTls.java

Create `customer-service/src/main/java/lib/MongoTls.java` (and the identical file in the other two services):

```java
package lib;

import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManagerFactory;
import java.io.FileInputStream;
import java.security.KeyStore;

public final class MongoTls {
    private MongoTls() {}

    /**
     * Builds an SSLContext from a PKCS12 client keystore (our identity) and
     * a PKCS12 truststore (who we trust -- just our own CA). Every path and
```

```
     * every password is resolved through SecretsLoader, never hardcoded.
     */
    public static SSLContext buildSslContext(String keystoreFileEnv, String
        keystorePwEnv,
                                             String trustStoreFileEnv,
                                                 String trustStorePwEnv)
                                                 throws Exception {
        char[] keystorePw = SecretsLoader.readFileSecret(keystorePwEnv).
            toCharArray();
        KeyStore keyStore = KeyStore.getInstance("PKCS12");
        try (var in = new FileInputStream(SecretsLoader.readFileSecret(
            keystoreFileEnv))) {
            keyStore.load(in, keystorePw);
        }
        KeyManagerFactory kmf = KeyManagerFactory.getInstance(
            KeyManagerFactory.getDefaultAlgorithm());
        kmf.init(keyStore, keystorePw);

        char[] trustStorePw = SecretsLoader.readFileSecret(trustStorePwEnv).
            toCharArray();
        KeyStore trustStore = KeyStore.getInstance("PKCS12");
        try (var in = new FileInputStream(SecretsLoader.readFileSecret(
            trustStoreFileEnv))) {
            trustStore.load(in, trustStorePw);
        }
        TrustManagerFactory tmf = TrustManagerFactory.getInstance(
            TrustManagerFactory.getDefaultAlgorithm());
        tmf.init(trustStore);

        SSLContext ctx = SSLContext.getInstance("TLS");
        ctx.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
        return ctx;
    }
}
```

Step 4.3 — Wire it into CustomerApp's Mongo connection

Replace the connection setup at the top of `CustomerApp.main`:

```
package lib;

import com.mongodb.MongoClientSettings;
import com.mongodb.MongoCredential;
import com.mongodb.ServerAddress;
import com.mongodb.client.*;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Updates;
import org.bson.Document;
import org.mindrot.jbcrypt.BCrypt;
import javax.net.ssl.SSLContext;
import java.util.List;
import java.util.Scanner;

public class CustomerApp {
    public static void main(String[] args) throws Exception {
        SSLContext ssl = MongoTls.buildSslContext(
                "MONGO_CLIENT_KEYSTORE_FILE", "
                    MONGO_CLIENT_KEYSTORE_PASSWORD_FILE",
                "MONGO_TRUSTSTORE_FILE", "MONGO_TRUSTSTORE_PASSWORD_FILE");

        MongoClientSettings settings = MongoClientSettings.builder()
                .applyToClusterSettings(b -> b.hosts(List.of(new
                    ServerAddress("mongo", 27017))))
```

```
                .applyToSslSettings(b -> b.enabled(true).context(ssl))
                .credential(MongoCredential.createMongoX509Credential())
                .build();

        try (MongoClient client = MongoClients.create(settings)) {
            MongoDatabase db = client.getDatabase("librarydb");
            MongoCollection<Document> books = db.getCollection("books");
            MongoCollection<Document> users = db.getCollection("users");

            // ... login and command loop continue in Part 5 ...
        }
    }
}
```

`MongoCredential.createMongoX509Credential()` takes no username argument at all — the driver reads it from the certificate we just presented during the TLS handshake. There is no string in this file that, if leaked, would let someone else authenticate as `customer-service`; the only thing that can do that is the private key inside `client.p12`, which never leaves its secret mount.

Step 4.4 — Add the environment variables and secrets in Compose

```
  customer-service:
    build: ./customer-service
    container_name: customer-service
    depends_on:
      - mongo
    environment:
      MONGO_CLIENT_KEYSTORE_FILE: /run/secrets/customer_client_p12
      MONGO_CLIENT_KEYSTORE_PASSWORD_FILE: /run/secrets/
          customer_keystore_password
      MONGO_TRUSTSTORE_FILE: /run/secrets/ca_truststore
      MONGO_TRUSTSTORE_PASSWORD_FILE: /run/secrets/ca_truststore_password
    secrets:
      - customer_client_p12
      - customer_keystore_password
      - ca_truststore
      - ca_truststore_password
    networks:
      - library-net
    stdin_open: true
    tty: true
```

`librarian-service` and `checkin-service` get the identical block, with `customer_` swapped for `librarian_` and `checkin_` respectively.

Step 4.5 — Add jbcrypt to every service's Gradle build

We'll need it in Part 5. Add one line to each `build.gradle.kts`:

```
dependencies {
    implementation("org.mongodb:mongodb-driver-sync:4.11.1")
    implementation("org.mindrot:jbcrypt:0.4")
}
```

Because we are copying dependency jars into `libs/`, the new dependency shows up automatically as one more file in that folder the next time you build.

Checkpoint: Part 4

Rebuild and run `customer-service` on its own: `docker compose run -rm customer-service`. It should connect and reach the (still empty) login section without any TLS handshake error. If you see `javax.net.ssl.SSLHandshakeException`, double check that `mongod.conf`'s `CAFile` and each service's truststore were built from the *same* `ca.crt`.

6 Part 5 — Application-Level Authentication for Human Users

mTLS answers "which *service* is this?" It says nothing about which *person* is sitting at `customer-service` or `librarian-service`. That's a separate, application-level problem, and it needs its own login step.

Why not just reuse the service's mTLS identity for the human, too? Because they answer different questions. The certificate proves the *container* is really `librarian-service` and not an imposter on the network. It says nothing about *which librarian* is using it right now, or whether they should still have access after being let go. Machine-to-machine identity and human identity are different concerns and need different credentials.

Step 5.1 — Bootstrap the first librarian account without a hardcoded password

Add this near the top of `LibrarianApp.main`, right after the Mongo connection is established:

```
MongoCollection<Document> users = db.getCollection("users");
if (users.countDocuments(Filters.eq("role", "librarian")) == 0) {
    String bootstrapPw = SecretsLoader.readFileSecret("
        LIBRARIAN_BOOTSTRAP_PASSWORD_FILE");
    String hash = BCrypt.hashpw(bootstrapPw, BCrypt.gensalt(12));
    users.insertOne(new Document("username", "librarian1")
            .append("passwordHash", hash)
            .append("role", "librarian")
            .append("mustChangePassword", true));
    System.out.println("Bootstrapped account 'librarian1'. Its password came
        from a mounted "
            + "secret file and was never printed, logged, or typed by a human
                .");
}
```

Add the matching environment variable and secret to `librarian-service` in Compose:

```
    environment:
      LIBRARIAN_BOOTSTRAP_PASSWORD_FILE: /run/secrets/
          librarian_bootstrap_password
    secrets:
      - librarian_bootstrap_password
```

(Add these alongside the mTLS environment and secrets from Step 4.4, not instead of them.)

Step 5.2 — The login prompt

Add this to `LibrarianApp.main`, after the bootstrap check and before the command loop:

```
Scanner in = new Scanner(System.in);
System.out.print("Username: ");
String username = in.nextLine();
System.out.print("Password: ");
String password = in.nextLine(); // a production CLI would use System.console
    ().readPassword()

Document u = users.find(Filters.eq("username", username)).first();
if (u == null || !BCrypt.checkpw(password, u.getString("passwordHash"))) {
    System.out.println("Authentication failed.");
    return;
}
if (!"librarian".equals(u.getString("role"))) {
    System.out.println("Access denied: this account does not hold the
        librarian role.");
    return;
}
System.out.println("Welcome, " + username + ".");
```

Never Store or Compare Plaintext Passwords

`BCrypt.hashpw` applies a slow, salted, one-way hash — the stored value in `passwordHash` cannot be reversed back into the password, and two users with the same password get different stored hashes because of the random salt. `BCrypt.checkpw` recomputes the hash from the submitted password and the stored salt and compares the results; the plaintext password submitted at login is never written anywhere, including logs. Never replace this with a plain equality check or a fast general-purpose hash like MD5 or SHA-256 — those are built for speed, which is exactly the wrong property for password storage.

Step 5.3 — CustomerApp gets the same treatment

Customers self-register instead of being bootstrapped, since there is no fixed initial account. Add a `register` command alongside `search` and `checkout`:

```
case "register" -> {
    System.out.print("Choose a username: ");
    String u2 = in.nextLine();
    if (users.find(Filters.eq("username", u2)).first() != null) {
        System.out.println("That username is taken.");
        break;
    }
    System.out.print("Choose a password: ");
    String p2 = in.nextLine();
    users.insertOne(new Document("username", u2)
            .append("passwordHash", BCrypt.hashpw(p2, BCrypt.gensalt(12)))
            .append("role", "customer"));
    System.out.println("Registered. Please restart and log in.");
}
```

The login block itself is identical to Step 5.2, just checking for `"customer"` instead of `"librarian"`.

What about checkin-service? It has no human operator — it's an unattended kiosk that a customer walks up to and returns a book at. Forcing it through a username/password login would just mean hardcoding one set of credentials into an unattended machine, which is worse than no login at all. Its authentication is exactly the mTLS identity from Part 4: only a process holding `checkin-service`'s private key can act as `checkin-service`, which is the right notion of identity for a machine, the same way a human username/password is the right notion of identity for a person. Different actors, different credentials.

Checkpoint: Part 5

Run `docker compose run -rm librarian-service`, note the bootstrap message, and log in as `librarian1` using the password from `secrets/librarian_bootstrap_password.txt`. Then try an intentionally wrong password and confirm you get `Authentication failed.` rather than any hint about whether the username existed.

7 Part 6 — Two Independent Layers of Authorization

A logged-in librarian should be able to add books. A logged-in customer should not. We enforce this twice, on purpose: once in the application (cheap to check, gives a friendly error message) and once in the database (Part 3's roles, which hold even if the application check has a bug).

Step 6.1 — Gate commands by role in the application

In `LibrarianApp`'s command loop, guard the `add` command (`status` is read-only and fine for any authenticated librarian to run, since we already checked the role at login in Step 5.2 — but if you later add a second role that can log into `librarian-service`, e.g. an "assistant" role, you would gate individual commands like this):

```
case "add" -> {
    if (!"librarian".equals(u.getString("role"))) {
        System.out.println("Access denied: 'add' requires the librarian role.
            ");
        break;
    }
    System.out.print("Title: ");
    String title = in.nextLine();
    System.out.print("Author: ");
    String author = in.nextLine();
    books.insertOne(new Document("title", title)
            .append("author", author)
            .append("status", "AVAILABLE"));
    System.out.println("Added.");
}
```

Step 6.2 — The database enforces the same boundary independently

Prove it by trying to violate it directly, bypassing the application entirely. From a throwaway debug container reusing `librarian-service`'s own certificate:

```
docker compose run --rm --entrypoint sh librarian-service -c \
  'echo "db.getSiblingDB(\"librarydb\").books.deleteMany({})" | \
   mongosh --tls --tlsCertificateKeyFile "$MONGO_CLIENT_KEYSTORE_FILE" 2>/dev
       /null || true'
```

> This particular command will actually fail differently than you might expect: our container images don't ship a `mongosh` binary at all (Part 7 explains why that's deliberate). The point of this step is conceptual — even if an attacker found a way to run arbitrary MongoDB driver calls *as* `librarian-service`, the role we created in Part 3 only grants `find`, `insert`, and `update` on `books`, never `remove`. A `deleteMany` call would be rejected by MongoDB itself with an `Unauthorized` error, regardless of what the application code does or fails to check.

Actor	App-layer gate (Part 5–6)	DB-layer grant (Part 3)
Human: customer	role == "customer"	find, update on `books`
Human: librarian	role == "librarian"	find, insert, update on `books`
Machine: checkin-service	mTLS identity only	find, update on `books` (no insert)

Table 1: Every actor is checked twice, by two independent mechanisms with different failure modes. A bug in one does not automatically defeat the other.

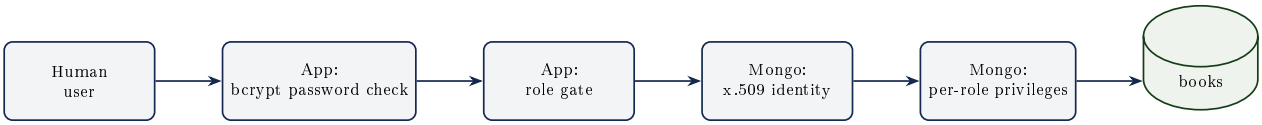


Figure 3: Every request from a human user passes through four independent checks before it touches data: two in the application (who are you, what does your role allow) and two at the database (which certificate is this connection, what does that identity's role allow). A machine actor like `checkin-service` skips the first two and starts at the mTLS identity check.

Checkpoint: Part 6

Log into `customer-service` and confirm there is no `add` command available to a customer account at all (it should never have been offered in the first place — customers and librarians are different applications). Then, as an exercise in reading role definitions rather than running code, look back at `customerServiceRole` in Part 3 and identify: could a compromised `customer-service` process insert a brand-new book into the catalog? Could it delete one? Justify both answers from the privilege list alone.

8 Part 7 — Hardening the Containers Themselves

Secrets and RBAC protect the *data*. These last few settings shrink what a compromised container can do to its *host* and to *itself*, on the assumption that any given service might eventually have a bug.

Step 7.1 — Run as a non-root user inside the image

Add this near the end of each service's Dockerfile, in the runtime stage:

```
RUN addgroup -S app && adduser -S app -G app
USER app
```

If the JVM process is ever tricked into writing or reading a file it shouldn't, it does so as an unprivileged user, not as `root` inside the container.

Step 7.2 — Lock the filesystem and drop Linux capabilities in Compose

```
  customer-service:
    build: ./customer-service
    read_only: true
    tmpfs:
      - /tmp
    cap_drop:
      - ALL
    security_opt:
      - no-new-privileges:true
```

`read_only: true` makes the container's own filesystem immutable at runtime — our app never needs to write anywhere except a scratch `/tmp`, which we grant explicitly as a tmpfs. `cap_drop: [ALL]` removes every Linux capability the process doesn't need (raw sockets, changing file ownership, and so on); a JVM running a simple console app needs none of them. `no-new-privileges` stops the process from gaining more privileges than it started with, even via a setuid binary. Apply the same three settings to `librarian-service` and `checkin-service`.

This is also why Step 6.2's debug command didn't work: our runtime images are deliberately minimal (`eclipse-temurin:17-jre-alpine` plus our own jar and `libs/`) and never include a `mongosh` binary, a shell history, or general-purpose tools. A smaller image is both a smaller download and a smaller attack surface — there is simply less installed for an attacker with a foothold to abuse.

Try It Yourself

Apply `read_only`, `cap_drop`, and `USER app` to `mongo` as well. You'll find it needs a small adjustment: the official Mongo image needs write access to `/data/db` (already a volume, so that's fine) and a writable `/tmp`. Figure out the minimal `tmpfs:` list that lets `mongod` start cleanly under `read_only: true`.

9 Part 8 — Shipping to a Customer Without Shipping Your Secrets

Suppose this library system is a product you sell, and a customer will run it on their own infrastructure. The image you hand them must be fully deployable by their own operators — and must not hand them (or anyone who later obtains a copy of that image) anything that helps attack *you*, *other customers*, or even that same customer's own deployment.

Safe to bake into the shipped image	**Must be supplied by the customer at deploy time**
Application code and compiled jars	That customer's own root CA and all certificates
The `libs/` dependency jars	That customer's own `mongo_root_password`
Default, non-secret `mongod.conf` structure	That customer's own `librarian_bootstrap_password`
The `docker-compose.yml` *template* (with `secrets:` pointing at files the customer provides)	Any of *your* internal certificates, staging credentials, or test accounts

Table 2: The dividing line is not "is this useful to the customer" but "does this value need to be secret, and if so, whose secret is it." A customer's own generated certificates are config *from their perspective* but must never originate from you, because a value you generated and shipped to many customers is really a single shared secret with many holders.

The Single Most Important Rule In This Table

Never generate the customer's CA, root password, or bootstrap password for them and bake it into an image or a shared installer. If you do, every customer who receives that image shares the same secret, and a leak at any one customer's site compromises all of them. Ship `generate-certs.sh` (Part 1) itself, or an equivalent onboarding script, so each customer generates their *own* CA and secrets locally, the same way you did in this recitation.

Step 8.1 — The pre-shipment check

Run this against every image before it leaves your build pipeline:

```
docker build -t library/customer-service:1.0 customer-service/
docker history --no-trunc library/customer-service:1.0 | \
  grep -Ei '\.key|\.p12|\.pem|password|secret' && \
  echo "STOP: possible secret material found in an image layer" || \
  echo "OK: no obvious secret material in any layer"
```

This is the same check from the Part 2 checkpoint, now framed as a release gate rather than a one-off exercise. A real CI pipeline would fail the build on the first branch of that `&&`.

This check is a pattern match, not a proof — it catches the mistake from Step 2.1, not every possible way a secret could leak. Treat it as one layer of defense alongside the discipline of never writing a `COPY` instruction that touches `secrets/` in the first place (Part 2), which `.dockerignore` makes structurally hard to do by accident.

Step 8.2 — Generalizing beyond file-based Compose secrets

Everything in this recitation reads a secret's value from a file path named by an environment variable. That is deliberate: it is exactly the interface Docker Swarm secrets, Kubernetes Secrets mounted as volumes, and most cloud secret managers (via a sidecar or CSI driver) all present to a container. `SecretsLoader` would not need to change at all to move from Compose file secrets to any of those — only `docker-compose.yml`'s `secrets:` section changes, in a real deployment, to reference the orchestrator's secret store instead of a local file.

10 Part 9 — The Complete, Hardened System

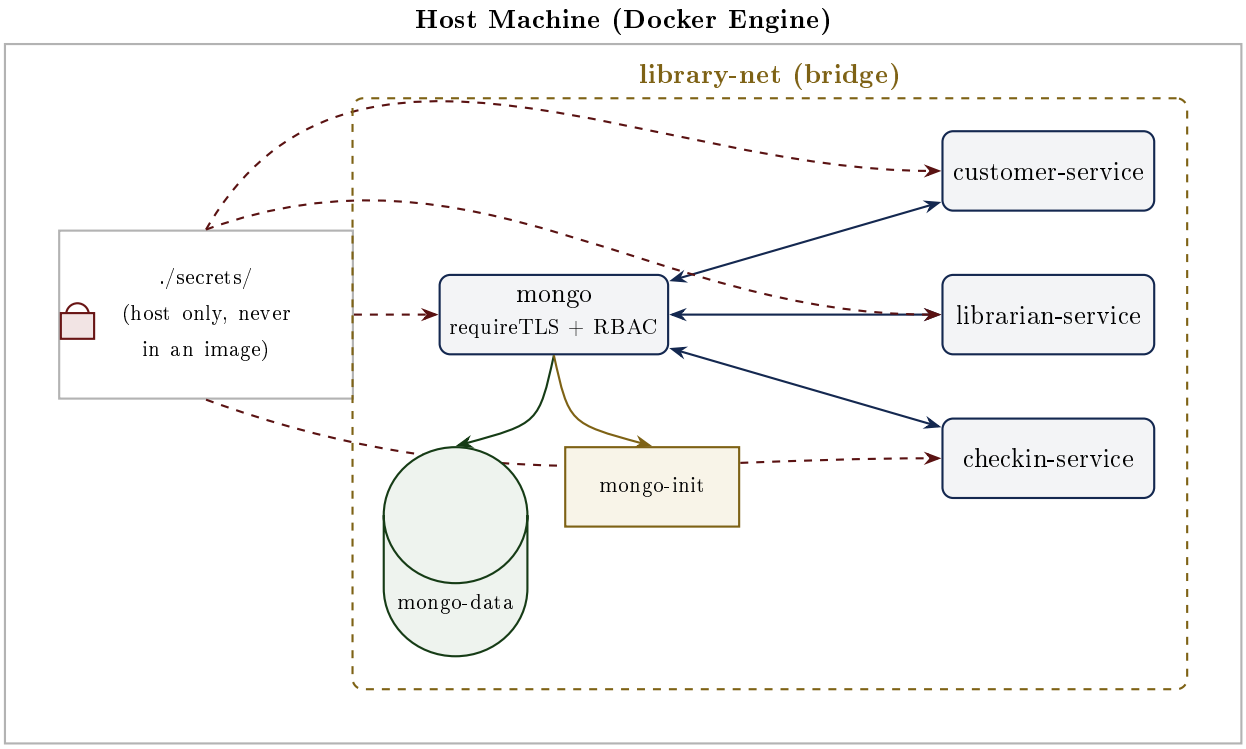


Figure 4: The complete, hardened system. Certificates and passwords live only on the host, outside the network boundary, and are mounted (dashed red arrows) into containers at runtime by Compose — never baked into an image layer. Every solid arrow inside `library-net` is mutual TLS, authenticated by certificate and authorized by a least-privilege MongoDB role. Human users additionally authenticate to `customer-service` and `librarian-service` with a bcrypt-checked password before any command is accepted.

Compare this figure to the previous recitation's final topology (Figure 4 there). The containers, the network, and the volumes are identical — we have not changed the system's shape. What changed is every arrow's *meaning*: each one now carries an authenticated, encrypted, and authorized connection instead of a bare, trusting one.

11 Exercises

Exercise 1. Certificate rotation. `customer-service`'s laptop is stolen. Write out, in order, the exact commands and Compose changes you would run to revoke that certificate's access without taking down `librarian-service` or `checkin-service`. (Hint: you do not need to touch the CA's private key at all.)

Suggested Answer

Revocation here means removing the identity's ability to *authenticate*, not touching the CA:

(a) Connect to `mongo` as the root SCRAM user and drop the compromised identity: `db.getSiblingDB("$external").dropUser("CN=customer-service,OU=services,O=SE424-Library")`. The stolen certificate can still complete a TLS handshake (it's still CA-signed), but MongoDB will refuse to authenticate it to anything, since the user backing that CN no longer exists.

(b) Remove `customer_client_p12` and `customer_keystore_password` from `docker-compose.yml`'s `secrets:` block so no container can accidentally be started with the compromised files again, and delete those two files from `secrets/customer-service/` on disk.

(c) Generate a brand-new key pair and certificate for `customer-service` (rerun the relevant loop body of `generate-certs.sh`) — never reissue a certificate from an already-compromised private key.

(d) Re-add the new `client.p12` and its keystore password as (new) Compose secrets, and create a matching x.509 user for the *new* certificate's Subject DN with `customerServiceRole`, the same role the old identity had.

(e) Rebuild and restart only `customer-service`: `docker compose up -d --build customer-service`. `librarian-service` and `checkin-service` never appear in any of these steps, because their certificates, Mongo users, and roles are entirely separate — this is exactly the point of giving every service its own identity instead of sharing one certificate across all three.

A production system would additionally maintain a Certificate Revocation List (CRL) or use OCSP so the stolen certificate is rejected at the TLS handshake itself, not just at the MongoDB authentication step; that infrastructure is outside this recitation's scope.

Exercise 2. A fourth role. Add a `reportsServiceRole` that can `find` on `books` but nothing else, and a matching x.509 user, without adding a fourth container — just add the role and user to `01-create-app-roles.js` and note in one sentence why an unused role sitting in the database is harmless as long as no certificate is issued for it.

Suggested Answer

Add to `01-create-app-roles.js`:

```
db.createRole({ role: "reportsServiceRole", privileges: [
  { resource: { db: "librarydb", collection: "books" },
    actions: ["find"] } ], roles: [] });
external.createUser({
  user: "CN=reports-service,OU=services,O=SE424-Library",
  roles: [{ role: "reportsServiceRole", db: "librarydb" }] });
```

Why it's harmless unused: a MongoDB role is only reachable through an authenticated identity that has been explicitly granted it; since no certificate with CN=`reports-service` exists and no container presents that identity, there is no connection anywhere in the system

that could ever exercise `reportsServiceRole`'s privileges, so defining it changes nothing about what any real, running client can do.

Exercise 3. Break the app-layer gate on purpose. Comment out the role check in Step 6.1's `add` command (so any logged-in user, including a customer, could theoretically call it if the command were reachable) and confirm via Part 3's role definitions that `customerServiceRole` would still block the underlying `insert` at the database. Write two sentences on why this exercise is only meaningful because of Part 3, not despite it.

Suggested Answer

Removing the role check in `LibrarianApp` does nothing to `customer-service` at all — the two are separate applications — so this exercise is really about confirming that *if* a customer somehow reached an `insert`-shaped code path, `customerServiceRole` (Part 3) grants only `find` and `update` on `books`, so MongoDB itself would reject the `insert` with an `Unauthorized` error regardless of what the application allowed. The exercise is meaningful only because Part 3 independently defined a role without `insert`: if that role had mistakenly been granted `insert` (or a broader role like `readWrite` had been used instead of a purpose-built one), removing the application check would have left nothing standing between a bug and a real unauthorized write — the app-layer gate and the database-layer grant have to *both* be correct, and this exercise only demonstrates that because Part 3 happened to get its half right.

Exercise 4. Password reset flow. `librarian1` was created with `mustChangePassword: true` in Step 5.1 but nothing in Part 5 ever checks that field. Add the check: on successful login, if it is `true`, require the user to set a new password (hashed the same way) before reaching the command loop, and flip the field to `false`.

Suggested Answer

Insert this immediately after the successful role check in Step 5.2, before the command loop begins:

```
if (Boolean.TRUE.equals(u.getBoolean("mustChangePassword"))) {
  System.out.println("You must set a new password.");
  System.out.print("New password:  ");
  String newPw = in.nextLine();
  String newHash = BCrypt.hashpw(newPw, BCrypt.gensalt(12));
  users.updateOne(Filters.eq("username", username),
    Updates.combine(
      Updates.set("passwordHash", newHash),
      Updates.set("mustChangePassword", false)));
  System.out.println("Password updated.");
}
```

Two details matter here: the new password is hashed exactly the same way the bootstrap password was (same `BCrypt.gensalt(12)` work factor), and both fields are updated in a single `updateOne` call so the account can never be left in a state where the hash changed but `mustChangePassword` did not (or vice versa).

Exercise 5. Prove the read-only filesystem works. After completing Part 7's `trybox` for `mongo`, deliberately try to `docker exec -it mongo sh -c "touch /etc/pwned"` and confirm it fails. Explain in one sentence what class of attack this specific hardening step is aimed at, distinct from anything TLS or RBAC protects against.

Suggested Answer

The command fails with a message like `touch: /etc/pwned: Read-only file system`, because `read_only: true` makes every path outside the explicitly declared writable volumes/tmpfs mounts (`/data/db` and whatever `tmpfs:` entries you added) immutable at the kernel level. This protects against *post-compromise persistence and tampering*: even if an attacker found a way to execute code inside the `mongo` container (via an application bug, a vulnerable dependency, and so on), they could not plant a malicious binary, modify `mongod` itself, add a cron job, or alter any configuration file on disk. This is a fundamentally different threat than TLS (which protects data *in transit* between containers) or RBAC (which limits what an *authenticated database client* can do to *data*) — read-only filesystems limit what a process that already has arbitrary code execution inside the container can do to the *container itself*.