# Recitation 6: Containerizing a Distributed Library System

Dockerfiles, Compose Networks, Volumes, and the Docker Shell

Michael J. May

1 December 2026

**Introduction** We're going to build a small distributed library system made of three independent Java applications and one shared MongoDB database:

**customer-service** lets a customer search for books and request/check out a book.

**librarian-service** lets a librarian add new books and check their status

**checkin-service** an automated kiosk that marks a book as returned when a customer brings it back

**mongo** the single MongoDB instance holding all customer and book data, shared by all three services.

By the end of this recitation, all four containers will run together on a private Docker network, the database will persist its data on disk using a volume, and you will know how to open a shell inside any running container to inspect or fix things manually.

## 1 Learning Objectives

By the end of this lab you will be able to:

- Write a `Dockerfile` that packages a Java application into a runnable image.
- Build, tag, run, and inspect Docker images and containers from the command line.
- Run a MongoDB container and connect to it using `docker exec`.
- Create a custom Docker network so multiple containers can find each other by name.
- Write a `docker-compose.yml` file that starts an entire multi-service system with one command.
- Use **volumes** to persist MongoDB's data across container restarts and to share files between host and container.
- Open a shell inside a running container (`docker exec -it ... sh`) to debug and configure things manually.

**Prerequisites.** Docker Desktop (or Docker Engine + Compose plugin) installed and running. Verify with:

```
docker --version
docker compose version
```

You do *not* need Java or Gradle installed on your host machine. Everything will be compiled *inside* the containers. This is one of the advantages of containerization that we will highlight along the way.

## 2 Part 0: Project Layout

We're going to create the following folder structure in the recitation. We'll fill in each file step by step over the course, so you don't need to make everything at the start. This structure is for reference and to point you how to organize the IntelliJ projects you're going to create in the next sections.

```
library-system/
  customer-service/
    src/main/java/lib/CustomerApp.java
    build.gradle.kts
    settings.gradle.kts
    Dockerfile
  librarian-service/
    src/main/java/lib/LibrarianApp.java
    build.gradle.kts
    settings.gradle.kts
    Dockerfile
  checkin-service/
    src/main/java/lib/CheckinApp.java
    build.gradle.kts
    settings.gradle.kts
    Dockerfile
  mongo-init/
    init.js
  docker-compose.yml
```

# 3 Part 1: One Simple Java App, No Docker Yet

Before we containerize anything, let's write a small Java program so we understand what we are packaging. The first version does not talk to MongoDB at all; it just prints a menu. We'll grow it later.

### Step 1.1: A trivial CustomerApp

Create `customer-service/src/main/java/lib/CustomerApp.java`:

```
package lib;

public class CustomerApp {
    public static void main(String[] args) {
        System.out.println("Customer service starting up...");
        System.out.println("(No database connection yet.  That comes in Part
            4.)");
    }
}
```

### Step 1.2: A Gradle build written in the Kotlin DSL

We'll use Gradle so dependencies like the MongoDB driver are managed automatically. Gradle build files can be written in Groovy (`build.gradle`) or in the **Kotlin DSL** (`build.gradle.kts`). We'll use Kotlin so we get autocomplete inside the build file. It's fine to have the build script written in Kotlin even if the application code is in Java.
Use IntelliJ to make a new Gradle project with Kotlin DSL called "customer-service." If you open the `settings.gradle.kts` file you'll see it's all ready:

```
rootProject.name = "customer-service"
```

Modify the `customer-service/build.gradle.kts` to have the following elements:

```
plugins {
    id("java")
    application
}

group = "il.ac.kinneret.docker.customer-service"
version = "1.0"

repositories {
    mavenCentral()
}

dependencies {
    testImplementation(platform("org.junit:junit-bom:6.0.0"))
    testImplementation("org.junit.jupiter:junit-jupiter")
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")
    implementation("org.mongodb:mongodb-driver-sync:5.15.0")
    runtimeOnly("org.slf4j:slf4j-nop:2.0.16")
}

java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(25))
    }
}

application {
    mainClass.set("lib.CustomerApp")
}
```

```
30
31 tasks.test {
32     useJUnitPlatform()
33 }
34
35 // Copy every runtime dependency jar into build/libs/libs, right next to
36 // the application jar that the `jar` task produces in build/libs.
37 tasks.register<Copy>("copyDependencies") {
38     from(configurations.runtimeClasspath)
39     into(layout.buildDirectory.dir("libs/libs"))
40 }
41
42 tasks.jar {
43     archiveBaseName.set("customer-service")
44     archiveClassifier.set("")
45     archiveVersion.set("")
46     dependsOn("copyDependencies")
47     manifest {
48         attributes["Main-Class"] = "lib.CustomerApp"
49         attributes["Class-Path"] = configurations.runtimeClasspath.get().
               files
50             .joinToString(" ") { "libs/${it.name}" }
51     }
52 }
```

The MongoDB driver looks for an SLF4J logging backend at startup; without one it prints a harmless but noisy `SLF4J not found on the classpath` warning. `slf4j-nop` is a no-op binding that satisfies that lookup and quietly discards the log messages, so no extra configuration is needed.

After running `gradle jar`, you should see:

```
build/libs/
  customer-service.jar
  libs/
    mongodb-driver-sync-5.15.0.jar
    bson-5.15.0.jar
    mongodb-driver-core-5.15.0.jar
    slf4j-api-2.0.16.jar
    slf4j-nop-2.0.16.jar
    ...
```

# 4 Part 2: A First Dockerfile

Now we'll package `customer-service` into an image using a **multi-stage build**. One stage compiles the code with Gradle. A second, much smaller stage, runs the compiled jar. This keeps the final image small because it does not carry the entire Gradle toolchain.

### Step 2.1: Write the Dockerfile

Create `customer-service/Dockerfile`:

```
# Stage 1: build
FROM gradle:9-jdk25 AS build
WORKDIR /build
COPY build.gradle.kts settings.gradle.kts ./
COPY src ./src
RUN gradle jar --no-daemon

# Stage 2: run
FROM eclipse-temurin:25-jre-alpine
WORKDIR /app
COPY --from=build /build/build/libs/customer-service.jar app.jar
COPY --from=build /build/build/libs/libs ./libs
CMD ["java", "-jar", "app.jar"]
```

`--no-daemon` tells Gradle not to keep a background process running after the build finishes. Since we're making a throwaway build stage there is no benefit to the daemon (which exists to speed up repeated local builds).
The application jar ends up in `build/libs/`, Gradle's default output directory, with its dependency jars right beside it in `build/libs/libs/` thanks to the `copyDependencies` task. The second `COPY` line brings that whole `libs/` folder into the runtime image *as* `/app/libs`, so the `Class-Path` entries baked into the jar's manifest (each one a relative path like `libs/mongodb-driver-sync-5.15.0.jar`) resolve correctly when the container runs `java -jar app.jar`.

### Step 2.2: Build the image

From inside `customer-service/`, run:

```
docker build -t library/customer-service:1.0 .
```

`-t` gives the image a human-readable name and tag. The trailing `.` tells Docker the *build context* (the files it is allowed to `COPY`) is the current directory.

### Step 2.3: Run the container

```
docker run --rm library/customer-service:1.0
```

You should see the two lines it prints, and then the container exits (because `main` returns). `--rm` tells Docker to delete the stopped container automatically so it doesn't clutter your machine.

### Checkpoint: Part 2

Run `docker images` and confirm `library/customer-service` appears in the list. Run `docker ps -a` right after the container exits and confirm it is *not* there (because of `--rm`). Then run it again *without* `--rm` and confirm it now shows up in `docker ps -a` with status "Exited".

**Try It Yourself**

Repeat Steps 2.1 and 2.2 for `librarian-service` and `checkin-service`, creating IntelliJ projects for `LibrarianApp` and `CheckinApp` (just a print statement for now, same as `CustomerApp`). Be sure to change the main class names in the build.gradle.kts and settings.gradle files. You should end up with three images: `library/customer-service:1.0`, `library/librarian-service:1.0`, and `library/checkin-service:1.0`.

# 5 Part 3: Running MongoDB

Before wiring MongoDB into the system, let's run it alone to get comfortable.

### Step 3.1: Start a MongoDB container

```
docker run -d --name mongo-test -p 27017:27017 mongo:9
```

- `-d` runs the container in the background ("detached").
- `--name mongo-test` gives it a name we can refer to later.
- `-p 27017:27017` maps port 27017 on your host to port 27017 inside the container, so tools on your laptop (like MongoDB Compass) could connect to it.
- `mongo:9` is the official MongoDB image, pulled automatically from Docker Hub the first time you use it.

### Step 3.2: Open a shell *inside* the running container

This is a first use of the **Docker shell**, one of the techniques we'll see here.

```
docker exec -it mongo-test bash
```

`-it` means *interactive* + *allocate a terminal*, which is what makes the shell act like a normal terminal session. You are now inside the container's filesystem. Try:

```
ls /
whoami
exit
```

### Step 3.3: Talk to MongoDB using its own shell, mongosh

Instead of a generic `bash` shell, we can jump straight into MongoDB's own query shell, which is also just a program installed inside the container:

```
docker exec -it mongo-test mongosh
```

Inside `mongosh`, try:

```
use librarydb
db.books.insertOne({ title: "Distributed Systems", author: "Tanenbaum",
    status: "AVAILABLE" })
db.books.find()
exit
```

Notice that `docker exec` runs a new process inside a running container. It is different from `docker run`, which creates a new one. You will use `exec` regularly for debugging: checking logs, inspecting files, or, manually querying a database.

### Step 3.4: Clean up

```
docker stop mongo-test
docker rm mongo-test
```

### Checkpoint: Part 3

Stop and remove `mongo-test`, then start it again with the same command from Step 3.1. Run `mongosh` and check whether the book you inserted is still there. It should *not* be there, because we haven't given this container anywhere durable to store its data yet. Keep that observation

in mind; we fix it with a volume in Part 6.

# 6 Part 4: Connecting Two Containers Manually (Before Compose)

We could keep running `docker run` commands by hand and connect containers using a Docker *network*. Seeing the manual way first makes it obvious why Docker Compose (Part 5) is useful.

### Step 4.1: Create a user-defined network

```
docker network create library-net
```

Containers on the default Docker network cannot resolve each other by name. Containers on a *user-defined* network like `library-net` can. Docker runs an internal DNS server that resolves a container's name to its IP address automatically.

### Step 4.2: Start MongoDB on that network

```
docker run -d --name mongo --network library-net mongo:9
```

Note we no longer need `-p`: only `customer-service`, `librarian-service`, and `checkin-service` need to reach Mongo, and they can all do so over the private network. Nothing outside Docker needs direct access to the database.

### Step 4.3: Update CustomerApp to connect to Mongo

Replace the contents of `CustomerApp.java`:

```
package lib;

import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;

public class CustomerApp {
    public static void main(String[] args) {
        // "mongo" here is resolved by Docker's built-in DNS. It is the
        // *container name* of the MongoDB service on our network.
        String uri = "mongodb://mongo:27017";

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

            System.out.println("Connected. Books currently in the catalog:");
            for (Document book : books.find()) {
                System.out.println(" - " + book.getString("title")
                        + " (" + book.getString("status") + ")");
            }
        }
    }
}
```

### Step 4.4: Rebuild and run on the network

Now let's rebuild the customer service module. Remember that you need to run the command from within the customer-service directory.

```
docker build -t library/customer-service:1.0 .
docker run --rm --network library-net library/customer-service:1.0
```

If Part 3's data didn't persist, the list will be empty. You can seed a book first with `docker`

`exec -it mongo mongosh` as in Step 3.3.

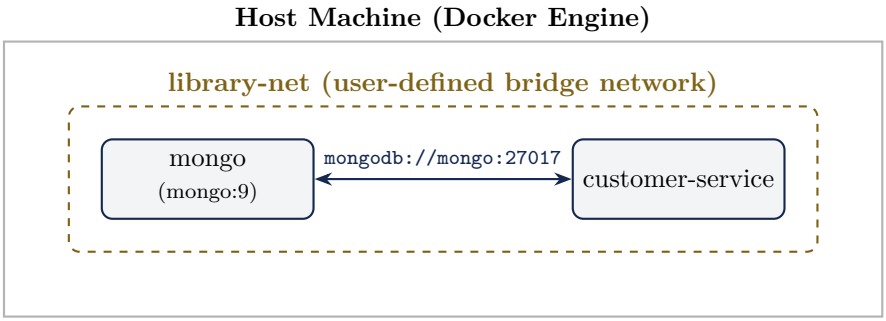



Figure 1: The topology after Part 4: two containers manually attached to a user-defined bridge network with `docker network create`. Docker's built-in DNS resolves the container name `mongo` to its current IP address, so `customer-service` can reach it without any `-p` port mapping to the host.

**Checkpoint: Part 4**

Explain in one sentence why `customer-service` can reach MongoDB using the hostname `mongo` even though no such hostname exists anywhere on the public internet. (Hint: think about what `docker network create` actually set up.)

# 7 Part 5: Docker Compose: One File, One Command

Typing four separate `docker run` commands with matching flags every time is error-prone. **Docker Compose** lets us describe the whole system (all four containers, the network that connects them, and any volumes) in a single declarative file.

### Step 5.1: Clean up the manual setup from Part 4

Compose is about to create its own container named `mongo`, and Docker doesn't allow two containers with the same name. Stop and remove the one we started by hand, along with the manual network (Compose will create its own):

```
docker rm -f mongo
docker network rm library-net
```

`docker rm -f` stops the container if it is running and removes it in one step. If either command reports that something doesn't exist, that's fine. It just means it was already cleaned up.

### Step 5.2: First pass: just Mongo and customer-service

Create `library-system/docker-compose.yml`: (Recent versions of Compose don't use a top-level `version:` field, so we leave it out.)

```
services:
  mongo:
    image: mongo:9
    container_name: mongo
    networks:
      - library-net

  customer-service:
    build: ./customer-service
    container_name: customer-service
    depends_on:
      - mongo
    networks:
      - library-net

networks:
  library-net:
    driver: bridge
```

### Step 5.3: Run it

From `library-system/`:

```
docker compose up --build
```

`--build` forces Compose to (re)build any service that has a `build:` entry. Notice: we never typed `--network` anywhere. Compose automatically creates the network and attaches every service in the file to it, and *every service can already reach every other service by its service name*. That name is the hostname, exactly like `mongo` was in Part 4.

Because `customer-service` exits immediately after printing (its `main` method returns), Compose will report it as `Exited (0)`. That's expected for now. The same goes for `librarian-service` and `checkin-service` once we add them. In Part 8 we will make the apps stay alive with a menu loop.

### Step 5.4: Add the other two services

Extend `docker-compose.yml`:

```
  librarian-service:
    build: ./librarian-service
    container_name: librarian-service
    depends_on:
      - mongo
    networks:
      - library-net

  checkin-service:
    build: ./checkin-service
    container_name: checkin-service
    depends_on:
      - mongo
    networks:
      - library-net
```

(Add this under the existing `services:` key, alongside `mongo` and `customer-service`.)

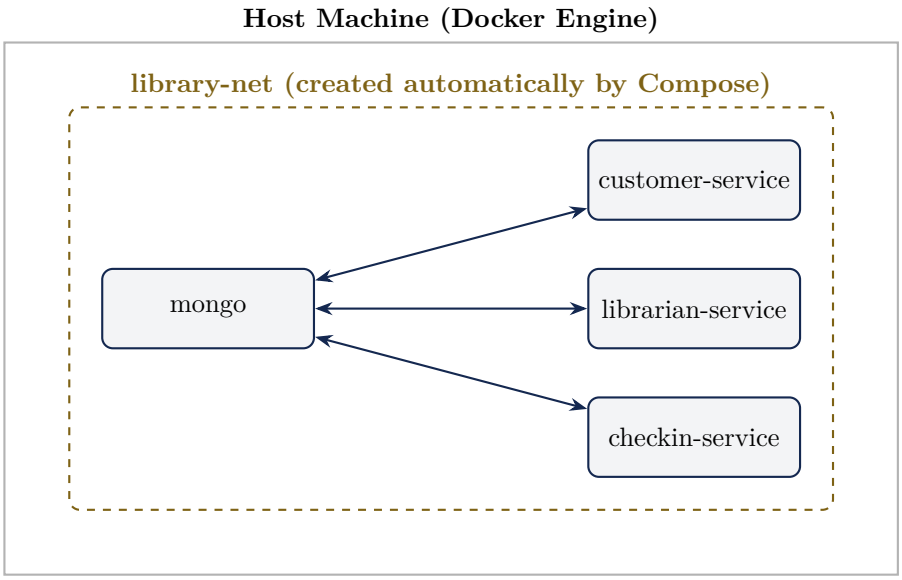



Figure 2: The topology after Part 5: Compose reads `docker-compose.yml` and automatically creates `library-net`, attaching all four services to it. Every service can already reach every other service by its *service name*, with no manual `docker network` commands required and no volumes attached yet.

### Step 5.5: Useful Compose commands

```
docker compose up -d        # start everything in the background
docker compose ps           # see the services that are currently running
docker compose ps -a        # see all services, including ones that exited
docker compose logs mongo   # view logs for just one service
docker compose logs -f      # follow logs for all services
docker compose down         # stop and remove everything Compose created
```

**Checkpoint: Part 5**

Run `docker compose up -d`, then `docker compose ps`. At first you may see `customer-service` listed briefly, but soon only `mongo` remains: plain `docker compose ps` shows only *running* containers, and our three Java apps exit as soon as `main` returns.
Next, run `docker compose ps -a` and confirm all four containers appear: `mongo` as `Up`, and the other three as `Exited (0)`. Run `docker network ls` and identify the network Compose created. Its name is usually the folder name followed by `_library-net`.

# 8 Part 6: Volumes: Making MongoDB's Data Survive

Right now, every time the `mongo` container is recreated, its data disappears. You saw this in the Part 3 checkpoint. A **volume** is storage that lives outside the container's writable layer, managed by Docker, so it survives container removal.

### Step 6.1: Add a named volume for Mongo's data directory

```
services:
  mongo:
    image: mongo:9
    container_name: mongo
    volumes:
      - mongo-data:/data/db
    networks:
      - library-net
```

`/data/db` is the directory the official Mongo image already uses internally for its database files. We're telling Docker to back that specific directory with a durable volume named `mongo-data` instead of the container's disposable layer.
Then declare the volume at the bottom of the file:

```
volumes:
  mongo-data:
```

### Step 6.2: Prove that data now survives

```
docker compose up -d
docker exec -it mongo mongosh librarydb
```

Inside `mongosh`, insert a book and leave:

```
db.books.insertOne({ title: "Elements of Distributed Computing", status: "
    AVAILABLE" })
exit
```

Now destroy and recreate the containers and look for the book:

```
docker compose down          # removes the CONTAINERS, but not named volumes
docker compose up -d
docker exec -it mongo mongosh librarydb
```

```
db.books.find()
exit
```

The book should still be there, even though the `mongo` container was destroyed and recreated.

**Two kinds of volumes you should know:**

**Named volume** (what we just used, e.g. `mongo-data`). Docker manages where it lives on disk. Best for data a container owns, like a database's files.

**Bind mount** You specify an exact folder on your host machine, and it is mirrored into the container. Best when *you* want direct, easy access to the files (e.g., to inspect an exported CSV, or to live-edit config).

### Step 6.3: A bind mount: seed script + host-visible exports folder

Create `mongo-init/init.js`:

```
db = db.getSiblingDB("librarydb");
db.books.insertMany([
  { title: "Distributed Systems", author: "Tanenbaum", status: "AVAILABLE" },
  { title: "Designing Data-Intensive Applications", author: "Kleppmann",
    status: "AVAILABLE" }
]);
db.customers.insertMany([
  { name: "Dana Levi", checkedOut: [] },
  { name: "Omer Cohen", checkedOut: [] }
]);
```

The official Mongo image automatically runs any `.js` file it finds in `/docker-entrypoint-initdb.d/` the *first* time it starts with an empty data directory. Let's bind-mount the local script into that folder:

```
services:
  mongo:
    image: mongo:9
    container_name: mongo
    volumes:
      - mongo-data:/data/db
      - ./mongo-init:/docker-entrypoint-initdb.d:ro
    networks:
      - library-net
```

`:ro` mounts it read-only, since the container should never need to modify our seed script. Because this only runs on a *fresh* data directory, remove the volume first to see it in action:

```
docker compose down -v      # -v also deletes named volumes; use with care!
docker compose up -d
```

The seed script runs only during Mongo's very first startup, inside a temporary server that shuts down and restarts before the real one is ready. Then connect and look:

```
docker exec -it mongo mongosh librarydb
```

```
db.books.find()
exit
```

You should see the two seeded books.

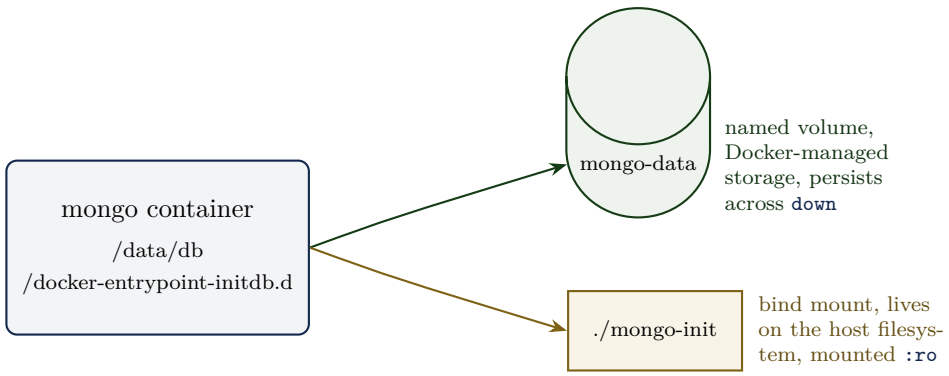



Figure 3: The two storage mechanisms attached to `mongo` by the end of Part 6. The named volume `mongo-data` backs `/data/db` with Docker-managed storage that survives `docker compose down`. The bind mount maps the host folder `./mongo-init` directly into `/docker-entrypoint-initdb.d`, so edits you make on your laptop are immediately visible inside the container.

**Checkpoint: Part 6**

In your own words, explain the difference between `docker compose down` and `docker compose down -v`. Which one would you use if you suspected your database was in a corrupted state? Which would you use for a normal shutdown?

# 9 Part 7: Building Out the Real Application Logic

Now that the plumbing works, let's give each service real behavior. This part is just copy-paste. We'll do thing in the next section.

### Step 7.1: LibrarianApp: add a book, check status

```java
package lib;

import com.mongodb.client.*;
import org.bson.Document;
import java.util.Scanner;

public class LibrarianApp {
    public static void main(String[] args) {
        try (MongoClient client = MongoClients.create("mongodb://mongo:27017"
            )) {
            MongoCollection<Document> books =
                client.getDatabase("librarydb").getCollection("books");
            Scanner in = new Scanner(System.in);

            System.out.println("Librarian console. Commands: add | status |
                quit");
            String cmd;
            while (!(cmd = in.nextLine()).equals("quit")) {
                switch (cmd) {
                    case "add" -> {
                        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.");
                    }
                    case "status" -> {
                        for (Document b : books.find()) {
                            System.out.println(b.getString("title")
                                    + " -> " + b.getString("status"));
                        }
                    }
                    default -> System.out.println("Unknown command.");
                }
            }
        }
    }
}
```

### Step 7.2: CustomerApp: search and check out

```java
package lib;

import com.mongodb.client.*;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Updates;
import org.bson.Document;
import java.util.Scanner;

public class CustomerApp {
    public static void main(String[] args) {
```

```
        try (MongoClient client = MongoClients.create("mongodb://mongo:27017")) {
            MongoCollection<Document> books =
                client.getDatabase("librarydb").getCollection("books");
            Scanner in = new Scanner(System.in);

            System.out.println("Customer console. Commands: search | checkout | quit");
            String cmd;
            while (!(cmd = in.nextLine()).equals("quit")) {
                switch (cmd) {
                    case "search" -> {
                        for (Document b : books.find(Filters.eq("status", "AVAILABLE"))) {
                            System.out.println(b.getString("title"));
                        }
                    }
                    case "checkout" -> {
                        System.out.print("Title to check out: ");
                        String title = in.nextLine();
                        var result = books.updateOne(
                            Filters.and(Filters.eq("title", title),
                                        Filters.eq("status", "AVAILABLE")),
                            Updates.set("status", "CHECKED_OUT"));
                        System.out.println(result.getModifiedCount() == 1
                                ? "Checked out!" : "Not available.");
                    }
                    default -> System.out.println("Unknown command.");
                }
            }
        }
    }
}
```

## Step 7.3: CheckinApp: the automated return kiosk

```
package lib;

import com.mongodb.client.*;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Updates;
import org.bson.Document;
import java.util.Scanner;

public class CheckinApp {
    public static void main(String[] args) {
        try (MongoClient client = MongoClients.create("mongodb://mongo:27017")) {
            MongoCollection<Document> books =
                client.getDatabase("librarydb").getCollection("books");
            Scanner in = new Scanner(System.in);

            System.out.println("Checkin kiosk ready. Scan a title (or type 'quit'):");
            String title;
            while (!(title = in.nextLine()).equals("quit")) {
                var result = books.updateOne(
                    Filters.and(Filters.eq("title", title),
                                Filters.eq("status", "CHECKED_OUT")),
                    Updates.set("status", "AVAILABLE"));
                System.out.println(result.getModifiedCount() == 1
```

```
                            ? "Thank you, book returned." : "This book was not
                                checked out.");
                }
            }
        }
    }
```

All three apps read from `stdin`, so to type commands into them once they run under Compose, you'll need an interactive terminal. We'll do that next.

### Step 7.4: Rebuild

Now that we've modified the code, we need to rebuild all of the container images. Run `docker compose build` to do that.

# 10 Part 8: Interacting with Running Services

### Step 8.1: Attach an interactive terminal to a Compose service

Because our apps read console input, run them with `docker compose run`, which starts a *new*, throwaway instance of a service with your terminal attached (rather than `docker compose up`, which is meant for long-running background services):

```
docker compose run --rm librarian-service
```

Type `add`, then a title and author, then `status` to confirm it was added, then `quit`.

### Step 8.2: Try the full workflow across services

```
docker compose run --rm librarian-service     # add a book
docker compose run --rm customer-service      # search, then checkout
docker compose run --rm checkin-service       # return it
```

Because all three connect to the *same* `mongo` container over `library-net`, each one sees the changes made by the others. This is the essence of a distributed system sharing state through a common data store.

### Step 8.3: Debugging with the Docker shell, revisited

Suppose `checkout` isn't working as expected. Rather than guessing, go straight into the database:

```
docker exec -it mongo mongosh librarydb
```

```
db.books.find().pretty()
db.books.updateOne({title: "Distributed Systems"}, {$set: {status: "AVAILABLE
    "}})
```

You can also shell into an *application* container to check that the jar and its dependencies are where you expect:

```
docker compose run --rm customer-service sh
ls /app
cat /app/app.jar | head -c 4    # just to prove it's a real jar (zip) file
exit
```

> Note we used `sh` here. The runtime image (`eclipse-temurin:25-jre-alpine`) is based on Alpine Linux, which ships the lightweight `sh` shell to keep the image small. Always check which shell an image provides. A `bash` on an Alpine-based image will simply fail with "executable file not found".

### Checkpoint: Part 8

Deliberately break something: use `docker exec` to open `mongosh` and delete all documents from `books` (`db.books.deleteMany({})`). Then run `customer-service` and confirm `search` correctly returns nothing. This is exactly the kind of manual state inspection you'll rely on constantly once you start debugging multi-node distributed systems in later labs.

## 11 Part 9: Final docker-compose.yml

For reference, here is the final version of the docker-compose file. Note that there are two small changes here.

```yaml
services:
  mongo:
    image: mongo:9
    container_name: mongo
    volumes:
      - mongo-data:/data/db
      - ./mongo-init:/docker-entrypoint-initdb.d:ro
    networks:
      - library-net

  customer-service:
    build: ./customer-service
    container_name: customer-service
    depends_on:
      - mongo
    networks:
      - library-net
    stdin_open: true
    tty: true

  librarian-service:
    build: ./librarian-service
    container_name: librarian-service
    depends_on:
      - mongo
    networks:
      - library-net
    stdin_open: true
    tty: true

  checkin-service:
    build: ./checkin-service
    container_name: checkin-service
    depends_on:
      - mongo
    networks:
      - library-net
    stdin_open: true
    tty: true

volumes:
  mongo-data:

networks:
  library-net:
    driver: bridge
```

`stdin_open: true` and `tty: true` are the Compose-file equivalents of the `-it` flags, letting you later run `docker compose up` and `docker attach customer-service` in a second terminal if you want an always-on interactive session instead of the throwaway `run --rm` pattern from Part 8.

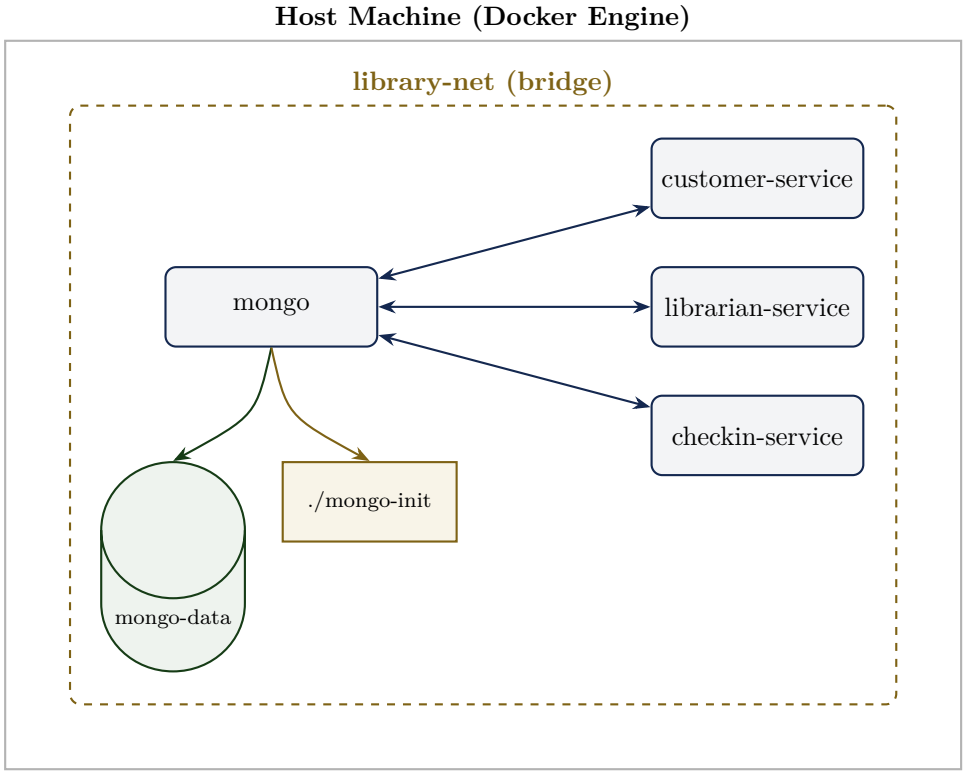



Figure 4: The complete system: four containers sharing `library-net`, with `mongo` as the hub every application service talks to, backed by a named volume for durable data and a read-only bind mount for the seed script. This is the full topology this lab has been building towards, one `docker` concept at a time.

## 12 Exercises

We've now put together a basic Docker network with four elements in a local distributed system. Here are some extensions you can try to make the example a bit more interesting.

**Exercise 1. Health checks.** Add a `healthcheck:` entry to the `mongo` service in Compose (research the `mongosh --eval "db.adminCommand('ping')"` command) and change `depends_on` on the other three services to the long form with `condition: service_healthy`, so the app services wait until Mongo is actually ready rather than merely started.

**Exercise 2. A fourth service.** Add a `reports-service` component that connects to `mongo` and prints how many books are currently `CHECKED_OUT` versus `AVAILABLE`. Wire it into `docker-compose.yml` exactly like the others.

**Exercise 3. Environment variables.** Instead of hardcoding `"mongodb://mongo:27017"` in every Java file, read it from an environment variable (`System.getenv("MONGO_URI")`) and set that variable per-service in `docker-compose.yml` under an `environment:` key. Explain in a sentence why this is better practice for a real system.

**Exercise 4. Broken volume, on purpose.** Remove the `mongo-data` volume mapping from `docker-compose.yml`, run the system, add some books, then run `docker compose down && docker compose up -d` again. Confirm the data is gone, then put the volume mapping back and repeat, confirming the data now survives. Write two sentences explaining, in your own words, exactly what changed between the two runs.

**Exercise 5. Manual recovery via the shell.** Simulate a scenario where a bug in `checkin-service` left a returned book stuck as `CHECKED_OUT` forever. Using only `docker exec` and `mongosh` (no code changes), find that book and fix its status by hand. This is exactly the kind of live operational skill you'll need when you start to run your own distributed system for the assignments.