Migrate a legacy Java service to modern .NET

Preserve behavior one vertical slice at a time

Session M2-07 | Copilot in Practice | Advanced | 3 hours

Agenda

Segment	Time
Define the migration boundary	8 min
Read the Java request path	10 min
Extract behavior and contract cases	12 min
Map the slice to .NET 10	12 min
Use Copilot with review gates	10 min
Prepared demo and lab handoff	8 min

Start with one route

```
POST /orders/price
```

The slice includes:

- request parsing and validation;
- pricing rules and rounding;
- response fields and errors;
- repeatable checks.

The health and sample-order routes stay in Java.

A rewrite hides the unknowns

Whole-service rewrite	Vertical-slice migration
Scope grows before behavior is known	One route defines the boundary
New code becomes the assumed truth	Existing behavior is captured first
Cutover checks arrive late	Each slice is checked before replacement
Java structure leaks into C#	Target design follows the contract

Record unknown behavior before expanding the slice.

Run the legacy service first

```
cd lab/starter/legacy-java-order-service
./mvnw test
./mvnw package
java -jar target/legacy-order-service-1.0.0.jar
```

```
Contract fixture tests passed: 5
Legacy order service listening on http://localhost:8080
```

Check how the service behaves before reading every file.

Trace the request path

```
App.java
  → JsonCodec.readOrderRequest
  → OrderPricingService.price
  → JsonCodec.writeQuote
  → HTTP 200
```

Errors from validation become HTTP 400 with:

```
{"error":"..."}
```

Classify what you find

Finding	Type	Action
`GOLD` means 10% discount	External behavior	Preserve
Coupon runs after tier discount	Calculation rule	Preserve and test
JDK `HttpServer`	Transport choice	Replace
Java getters	Language structure	Redesign
Unknown throughput	Open question	Record

Do not translate incidental structure into a requirement.

Pricing order is part of the contract

```
subtotal
  → tier discount
  → SAVE10 coupon
  → shipping decision
  → tax
  → total
```

Moving one step changes the result.

Money stays in integer cents

```
rounded cents =
(amount in cents × basis points + 5000) / 10000
```

Examples:

- `SILVER` = 500 basis points
- `GOLD` = 1000 basis points
- tax = 825 basis points

No binary floating point.

Shipping uses discounted merchandise

Destination	Rule
Domestic	795 cents
Domestic at 5000+ discounted cents	Free
International	1995 cents

Tax applies to discounted merchandise. Shipping is not taxed.

Shared fixtures hold accepted behavior

```
{"name":"round-half-up",
 "request":{
   "customerTier":"SILVER",
   "destination":"DOMESTIC",
   "couponCode":"SAVE10",
   "items":[{"sku":"SKU-ROUND","quantity":1,"unitPriceCents":101}]
 },
 "expected":{"totalCents":888}}
```

The Java and .NET tests read the same file.

What the five cases protect

Case	Risk
`standard-domestic`	baseline shipping and tax
`silver-free-domestic-shipping`	threshold after discount
`gold-save10-domestic`	discount order
`gold-international`	destination rule
`round-half-up`	one-cent drift

Add cases when you discover behavior. Do not invent them after migration.

Contract tests cover selected behavior

Shared cases check only the selected successful calculations.

Also review:

- invalid tiers and destinations;
- empty orders and bad quantities;
- unsupported coupons;
- overflow behavior;
- method, path, status, and error JSON.

Map responsibilities, not classes

Java	.NET 10
`HttpHandler`	Minimal API route
`OrderRequest`	request records
`OrderPricingService`	`PricingEngine`
`PriceQuote`	response record
fixture test main	contract test executable

The target can be smaller than the source.

The Minimal API boundary

```
app.MapPost("/orders/price",
    (OrderRequest request, PricingEngine pricing) =>
{
    try
    {
        return Results.Ok(pricing.Price(request));
    }
    catch (ArgumentException exception)
    {
        return Results.BadRequest(new { error = exception.Message });
    }
});
```

Keep transport code thin.

Platform APIs are enough

The target uses:

- ASP.NET Core Minimal APIs;
- built-in dependency injection;
- `System.Text.Json`;
- C# records;
- a console test executable.

Do not add a database, mapper, mediator, or test package.

Copilot starts with inspection

```
Trace POST /orders/price through the Java starter.
Return entry point, parsing, validation, pricing,
response mapping, and tests with file and symbol names.
Do not propose .NET code yet.
```

The first prompt builds a map. It does not ask for a rewrite.

Ask for the behavior rules

```
Compare the Java pricing code with the shared JSON cases.
List operation order, rounding, shipping threshold,
accepted values, and uncovered behavior.
Do not change files.
```

Review the answer against code and test output.

Bound the implementation prompt

```
Implement PricingEngine.Price in the .NET 10 scaffold.
Preserve the shared JSON contract.
Use integer cents, checked arithmetic, and platform APIs.
Do not add packages or change the endpoint shape.
```

Name the limits so you can check whether Copilot stayed within them.

Review generated code before running it

Check:

- JSON names;
- calculation order;
- integer arithmetic;
- overflow handling;
- shipping threshold;
- tax base;
- added dependencies.

Generated code is a draft.

A clean migration can still be wrong

This looks reasonable:

```
var couponDiscount = Percentage(subtotal, 1000);
```

The accepted behavior is:

```
var couponDiscount = Percentage(afterTier, 1000);
```

One line breaks `gold-save10-domestic`.

Verification has layers

```
Java fixture tests
        +
.NET fixture tests
        +
same HTTP request to both services
        +
human review of uncovered edges
```

No single layer proves the whole service.

Prepared demo

1. Run the Java contract cases.
2. Send the gold plus coupon request.
3. Show the Java request path.
4. Run a deliberately wrong .NET discount order.
5. Watch the shared case fail.
6. restore the accepted code and rerun.

Use the failing case to show why the calculation order matters.

Review point: architecture notes

Learners should name:

- the HTTP entry point;
- the parsing boundary;
- the domain calculation;
- the response mapping;
- the existing checks.

Class names alone are not an architecture explanation.

Review point: compare test and HTTP results

Save these results:

```
Java:  Contract fixture tests passed: 5
.NET:  Contract fixture tests passed: 5
HTTP:  matching field values for one shared request
```

Also record checks that were not run.

Write the next backlog as slices

Priority	Slice	Checks before removal
P0	Order pricing	Contract and HTTP parity
P1	Sample orders	Response contract
P1	Health	Operational contract
P2	Error matrix	Status and error cases
P2	Concurrency	Measured target

Keep the Java behavior until tests and review support its replacement.

Environment boundary

Primary: repository Dev Container or Codespaces

- Java 21 compiling Java 8-compatible source
- Maven 3.9.9 through `mvnw`
- .NET 10 SDK

Native setup alternative: JDK 17 or 21, .NET 10 SDK, Git, and `curl`.

Access stop

Before the lab, check that GitHub Copilot works in the training environment.

If access is missing:

1. stop;
2. preserve the starter files;
3. resolve account, extension, or policy access;
4. restart from preflight.

Do not switch to an unapproved tool.

Lab handoff

Inspect the Java service, complete architecture notes, implement the .NET pricing slice, run shared contract cases, compare HTTP responses, and write the migration backlog.

Complete one route and show its test results.