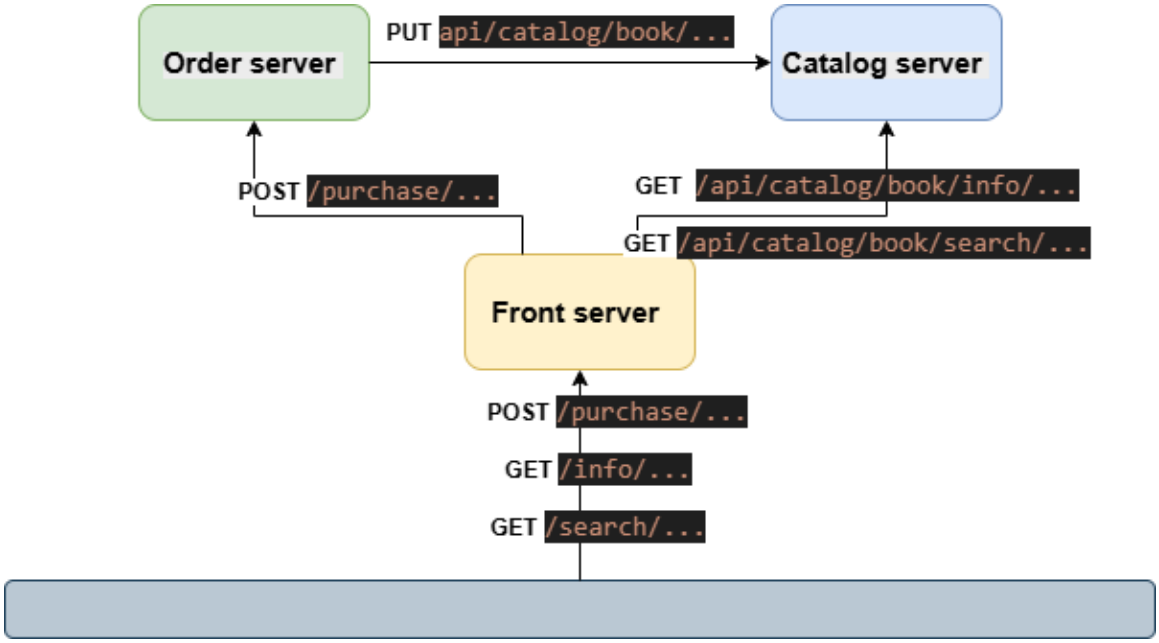
Bazar.com — Project Documentation

Contents

1. Project overview and design



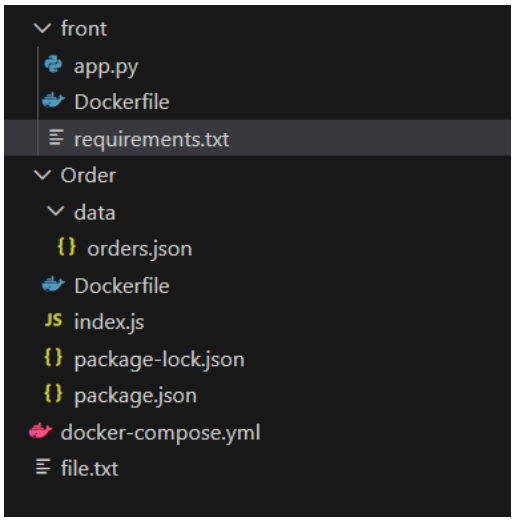
The system is built as three small, focused services that work together to provide search, book information and purchases. One service holds and manages the book catalog (titles, topics, quantities and IDs) and exposes REST endpoints to look up and update items; its data is stored in a local SQLite file so state survives restarts. Another service receives purchase requests, asks the catalog to decrement stock, records each successful order in a small orders. Json file and prints a clear confirmation message when a book is bought. A lightweight gateway sits in front and forwards client requests — search and info calls go straight to the catalog, while purchase calls are routed to the order service. Watching the logs while exercising the API shows the full flow: a client request → gateway → backend → persistent change → "bought book <title>" in the order logs.

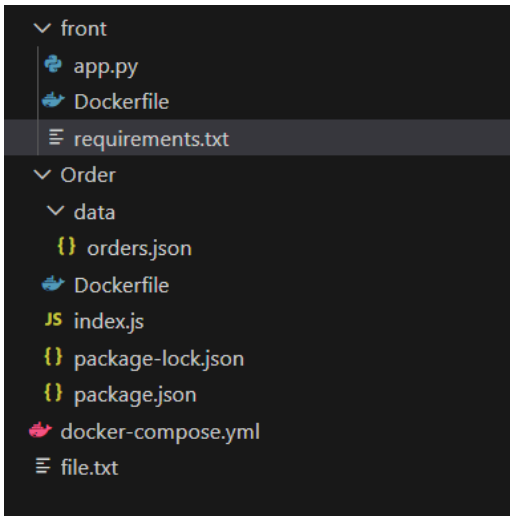
Containers and Docker Compose make the whole environment simple and repeatable: each service runs in its own container, connected by an internal network so services call each other by name, and volumes keep the SQLite database and orders file persistent between runs. This setup makes it straightforward to start the entire system with one command, run the three main user flows from your machine, and capture the output (including the purchase confirmation) for submission or testing.

The implementation intentionally uses different languages and runtimes to play to each component's strength and to illustrate a core microservice benefit: each service can be written in the language and framework best suited for its role. The catalog is implemented in ASP.NET for robust API and data access, the order service uses Node.js for quick JSON-forwarding and lightweight I/O, and the gateway uses Flask/Python for a minimal and readable proxy layer. That diversity keeps each codebase small and easy to understand while proving that independent services can interoperate smoothly.

Overall the design favors clarity, reproducibility and ease of testing. It provides the expected HTTP interfaces, persistent storage for both catalog and orders, and clear runtime output that demonstrates successful purchases

2. Repository layout





3. Catalog service (ASP.NET)

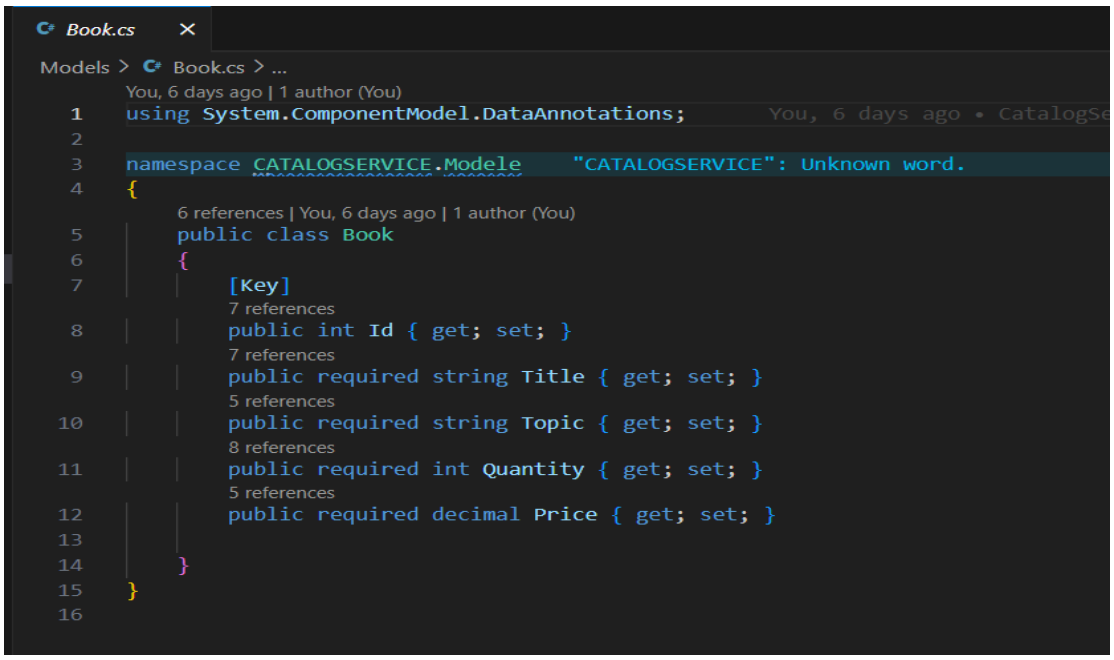
1)Files and purpose

- Docerfile

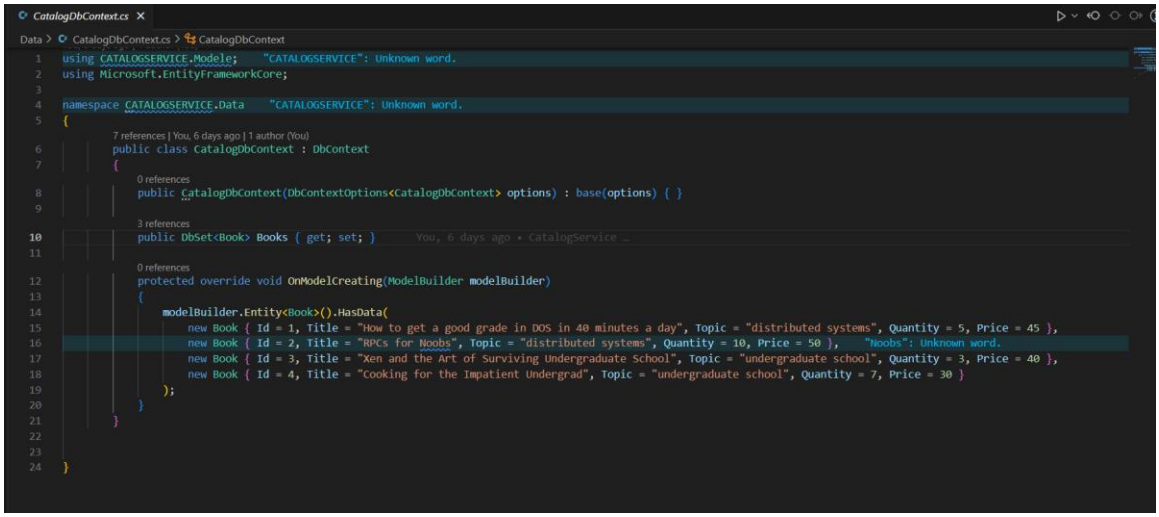
Code | Blame | 14 lines (14 loc) · 412 Bytes

```
1   FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
2   WORKDIR /app
3   COPY *.csproj ./
4   RUN dotnet restore
5   COPY . ./
6   RUN dotnet publish -c Release -o /app/publish
7   FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS runtime
8   WORKDIR /app
9   COPY --from=build /app/publish .
10  ENV ASPNETCORE_URLS=http://+:5142
11  EXPOSE 5142
12  ENV DOTNET_RUNNING_IN_CONTAINER=true
13  ENV SQLITE_DB_PATH=/app/catalog.db
14  ENTRYPOINT ["dotnet", "Catalog_Service.dll"]
```

- Model(Book.cs)



- Data (we use SQLite as a Database)



```
CatalogDbContext.cs
Data > CatalogDbContext.cs > CatalogDbContext
using CATALOGSERVICE.Modele;    "CATALOGSERVICE": Unknown word.
using Microsoft.EntityFrameworkCore;

namespace CATALOGSERVICE.Data    "CATALOGSERVICE": Unknown word.
{
        7 references | You, 6 days ago | 1 author (You)
        public class CatalogDbContext : DbContext
        {
            0 references
            public CatalogDbContext(DbContextOptions<CatalogDbContext> options) : base(options) { }

            3 references
            public DbSet<Book> Books { get; set; }      You, 6 days ago • CatalogService …

            0 references
            protected override void OnModelCreating(ModelBuilder modelBuilder)
            {
                modelBuilder.Entity<Book>().HasData(
                    new Book { Id = 1, Title = "How to get a good grade in DOS in 40 minutes a day", Topic = "distributed systems", Quantity = 5, Price = 45 },
                    new Book { Id = 2, Title = "RPCs for Noobs", Topic = "distributed systems", Quantity = 10, Price = 50 },    "Noobs": Unknown word.
                    new Book { Id = 3, Title = "Xen and the Art of Surviving Undergraduate School", Topic = "undergraduate school", Quantity = 3, Price = 40 },
                    new Book { Id = 4, Title = "Cooking for the Impatient Undergrad", Topic = "undergraduate school", Quantity = 7, Price = 30 }
                );
            }
        }


}
```

- Controller(BookController.cs)

- ## Search Api

```
[HttpGet("search/{topic}")]
0 references
public async Task<IActionResult> Search(string topic)
{
    Console.WriteLine("\n==============================");
    Console.WriteLine(" SEARCH REQUEST");
    Console.WriteLine($"Topic: {topic}");
    Console.WriteLine("------------------------------");

    var response = await _context.Books
    .Where(b => b.Topic.ToLower() == topic.Trim().ToLower())
    .Select(b => new SearchResponseDto
    {
        Id = b.Id,
        Title = b.Title

    }).ToListAsync();

    if (response.Count() == 0)
    {
        Console.WriteLine($" No books found for topic: {topic}");
        Console.WriteLine("==============================\n");
        return NotFound(new { message = $"No books found for this topic '{topic}'" });

    }

    Console.WriteLine($"Found {response.Count} book(s):");
    foreach (var r in response)
        Console.WriteLine($"    - [{r.Id}] {r.Title}");

    Console.WriteLine("==============================\n");
    return Ok(response);       You, 6 days ago • CatalogService …
}
```

Search – Find books by topic

- Accepts a **topic** and searches for all books that match it.
- Returns a list of books with their **Id** and **Title**.
- If no books are found, it returns **NotFound** with a message.
- Prints detailed logs about the search to the console.

- Info Api

```
[HttpGet("info/{id}")]
0 references
public async Task<IActionResult> GetInformation(int id)
{
    Console.WriteLine("\n==============================");
    Console.WriteLine("INFO REQUEST");
    Console.WriteLine($"Book ID: {id}");
    Console.WriteLine("------------------------------");

    var book = await _context.Books.FindAsync(id);
    if (book == null)
    {
        Console.WriteLine($"Book with ID {id} not found.");
        Console.WriteLine("==============================\n");
        return NotFound(new { message = "Book not found" });

    }
    var response = new InfoResponseDto
    {
        Title = book.Title,
        Price = book.Price,
        Quantity =book.Quantity
    };

    Console.WriteLine($" Book found:");
    Console.WriteLine($"    - Title: {response.Title}");
    Console.WriteLine($"    - Quantity: {response.Quantity}");
    Console.WriteLine($"    - Price: {response.Price}");
    Console.WriteLine("==============================\n");       You, 6 days ago
    return Ok(response);
}
```

Get Information – Get book details by ID

- Takes a **book ID** and retrieves the book from the database.
- If the book doesn't exist, it returns **NotFound**.

- If found, it returns the book's **Title, Price, and Quantity**.
- Also prints detailed logs to the console.

- ## Update Quantity Api

```
[HttpPut("{id}")]
0 references
public async Task<IActionResult> UpdateQuantity(int id, UpdateQuantityOfBook dto)
{
    for (int i = 1; i <= MaxRetries; i++)
    {
        using var TX = await _context.Database.BeginTransactionAsync(System.Data.IsolationLevel.Serializable);
        var book = await _context.Books
        .Where(b => b.Id == id)
        .FirstOrDefaultAsync();

        if (book == null)
            return NotFound(new { message = "Book not found" });

        int newQuantity = book.Quantity + dto.QuantityDelta;
        if (newQuantity < 0)
            return BadRequest(new { message = "Available quantity is zero" });
        book.Quantity = newQuantity;

        try
        {
            await _context.SaveChangesAsync();
            await TX.CommitAsync();

            return Ok(new
            {
                message = "Quantity updated successfully",
                id = book.Id,
                title = book.Title,
                quantity = book.Quantity
            });
        catch (DbUpdateConcurrencyException)
        {
            await TX.RollbackAsync();

            if (i == MaxRetries)
                return Conflict(new { message = "Conflict occurred, please try again later" });

            await Task.Delay(50);
        }
        catch
        {
            await TX.RollbackAsync();
            return StatusCode(500, new { message = "Failed to update Quantity" });
        }
    }

    return StatusCode(500, new { message = "error" });
}
```

This API is responsible for modifying the quantity of any book (increasing or decreasing it) while ensuring the process is secure and preventing conflicts if multiple users attempt to update the same book simultaneously.

Mechanism:

- It opens a transaction with the highest level of Serializable isolation to prevent cross-reading/modification.
- It retrieves the book; if it's not found, it returns NotFound.

- It calculates the new quantity; if it's less than zero, it returns BadRequest.
- It attempts to save the change.
- If a DbUpdateConcurrencyException (conflict) occurs, it rolls back and re-attempts up to MaxRetries.
- If it still doesn't work after all attempts, it returns Conflict.
- If an unexpected error occurs, it returns 500.

4. Order service (Node.js)

1) Files and purpose

- **Dockerfile**

```
Order > Dockerfile
1  FROM node:20
2
3  WORKDIR /app
4
5  COPY package*.json ./
6  RUN npm install
7
8  COPY . .
9
10 EXPOSE 3002
11 CMD ["node", "index.js"]
12
```

  - Uses node:**20** base image
  - Copies package*.json, runs npm install, copies source and exposes port **3002**.

- **index.js**
  - Express server that listens on port **3002**.
  - Reads CATALOG_URL from the environment (default **http://localhost:5142**).
  - Uses an orders file **data/orders.json** to persist orders locally.
  - Endpoint implemented: **POST /purchase/:id.**

- **package.json**

```
Order > {} package.json > ...
1  {
2    "name": "order",
3    "version": "1.0.0",
4    "main": "index.js",
     ▷Debug
5    "scripts": {
6      "start": "node index.js"
7    },
8    "dependencies": {
9      "express": "^4.19.2",
10     "axios": "^1.5.0"
11   }
12 }
13
```

  - Metadata and dependencies: **express**, **axios**.
  - Start script: **node index.js**.

- **data/orders.json**
    - ➢ Persistent JSON array of orders. Initially: [].

2) How it works

1. Client sends **POST /purchase/:id** to the Order service.
2. Order service sends a **PUT** request to the Catalog service at **/api/catalog/book/:id** with a body { quantityDelta: -1 } to decrement stock.

```
app.post('/purchase/:id', async (req, res) => {
  const id = req.params.id;
  try {
    const body = { quantityDelta: -1, QuantityDelta: -1 };

    const resp = await axios.put(`${CATALOG_URL}/api/catalog/book/${id}`, body, {
      headers: { 'Content-Type': 'application/json' },
      timeout: 8000
    });
```

3. If Catalog returns success (**HTTP 200**), Order appends a new order object to **data/orders.json** with fields { id, itemId, title, time } and logs bought book <title>.

```
async function readOrders() {
  try {
    const txt = await fs.readFile(ORDERS_FILE, 'utf8');
    return JSON.parse(txt);
  } catch {
    return [];
  }
}
async function writeOrders(arr) {
  await fs.mkdir(path.join(__dirname, 'data'), { recursive: true });
  await fs.writeFile(ORDERS_FILE, JSON.stringify(arr, null, 2));
}
```

```
if (resp.status === 200 && resp.data) {
  const title = resp.data.title || resp.data.Title || `item-${id}`;


  const orders = await readOrders();
  const newOrder = { id: orders.length + 1, itemId: Number(id), title, time: new Date().toISOString() };
  orders.push(newOrder);
  await writeOrders(orders);

  console.log(`bought book ${title}`);
  return res.status(200).json({ ok: true, order: newOrder });
} else {
  return res.status(502).json({ ok: false, error: 'catalog_error' });
}
```

4. Order returns { ok: true, order: ... } to the caller. If Catalog returns errors (400, 404, 409) Order maps these to appropriate responses and forwards the error reason.

```
} catch (e) {
  if (e.response) {
    const status = e.response.status;
    const data = e.response.data || {};

    if (status === 400) {
      const msg = data.message || data.error || 'out_of_stock';
      return res.status(400).json({ ok: false, error: msg });
    }
    if (status === 404) {
      const msg = data.message || 'not_found';
      return res.status(404).json({ ok: false, error: msg });
    }
    if (status === 409) {
      const msg = data.message || 'conflict';
      return res.status(409).json({ ok: false, error: msg });
    }
    return res.status(502).json({ ok: false, error: 'catalog_error', details: data });
  }

  console.error('purchase error', e.message || e);
  return res.status(500).json({ ok: false, error: 'purchase_failed' });
```

3) How to run

Docker

- Build and run via docker-compose (project root):

docker compose up -d --build order

Locally

- Install dependencies and run:

cd order

npm install

node index.js

4) Endpoints

```
app.post('/purchase/:id', async (req, res) => {
  const id = req.params.id;
```

- POST /purchase/:id
  - Description: Attempt to purchase book with id :id.
  - Behavior:
    - On success: returns HTTP 200 with JSON { ok: true, order: { id, itemId, title, time } }.
    - If out of stock: HTTP 400 { ok: false, error: "Available quantity is zero" }.
    - If catalog not reachable or other error: 502 or 500 with error field.

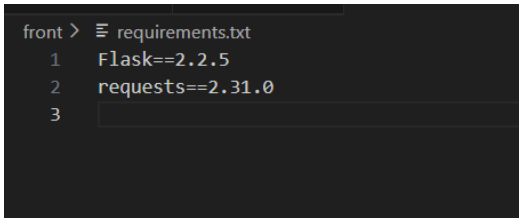
5) Data and logs

- **data/orders.json**: all successful purchases are appended here.
- Console log message on successful purchase: **bought book <title>**.

5. Front service (Flask)

1. Files and purpose

- **requirements.txt**



  - Lists Python dependencies: Flask==**2.2.5**, requests==**2.31.0**.

- **Dockerfile**

  - Uses python:**3.11-slim**, installs dependencies from requirements.txt, copies source and exposes port **3000**.

- **app.py**
  - Flask application that listens on port **3000**.
  - Reads CATALOG_URL (**http://localhost:5142**) and ORDER_URL (**http://localhost:3002**).
  - Implements three routes:
    - **GET /search?topic**=... → calls Catalog **GET /api/catalog/book/search/{topic}**

```python
@app.route('/search', methods=['GET'])
def search():
    topic = request.args.get('topic', '')
    logging.info(f"Front: search topic='{topic}'")
    if not topic:
        return jsonify({"error": "missing topic parameter"}), 400
    try:
        encoded = requests.utils.requote_uri(topic)
        resp = forward_get(f"{CATALOG_URL}/api/catalog/book/search/{encoded}")
        return (resp.content, resp.status_code, {'Content-Type': 'application/json'})
    except requests.RequestException as e:
        logging.error("Catalog unreachable: %s", e)
        return jsonify({"error": "catalog_unreachable"}), 502
```

    - **GET /info/<id>** → calls Catalog GET /api/catalog/book/info/{id}
    - **POST /purchase/<id>** → calls Order **POST /purchase/{id}**

```python
@app.route('/info/<int:item_id>', methods=['GET'])
def info(item_id):
    logging.info(f"Front: info id={item_id}")
    try:
        resp = forward_get(f"{CATALOG_URL}/api/catalog/book/info/{item_id}")
        return (resp.content, resp.status_code, {'Content-Type': 'application/json'})
    except requests.RequestException as e:
        logging.error("Catalog unreachable: %s", e)
        return jsonify({"error": "catalog_unreachable"}), 502

@app.route('/purchase/<int:item_id>', methods=['POST'])
def purchase(item_id):
    logging.info(f"Front: purchase id={item_id}")
    try:
        resp = forward_post(f"{ORDER_URL}/purchase/{item_id}")
        return (resp.content, resp.status_code, {'Content-Type': 'application/json'})
    except requests.RequestException as e:
        logging.error("Order unreachable: %s", e)
        return jsonify({"error": "order_unreachable"}), 502
```

2. How it works

- The Front acts like a layer that connects the client and the required server:
    - It forwards **search** and **info** GET requests to Catalog and returns Catalog's response transparently.
    - It forwards **purchase** POST requests to Order and returns Order's response transparently.

```python
def forward_get(url, timeout=5):
    return requests.get(url, timeout=timeout)

def forward_post(url, json=None, timeout=8):
    return requests.post(url, json=json, timeout=timeout)

def forward_put(url, json=None, timeout=8):
    return requests.put(url, json=json, timeout=timeout)
```

- The Front provides simple logging using Python's logging module.

3. How to run

Docker (recommended)

```
docker compose up -d --build front
```

Locally (development)

```
cd front

python -m venv venv

source venv/bin/activate  # or venv\Scripts\activate on Windows

pip install -r requirements.txt

python app.py
```

4. Endpoints

- **GET /search?topic=<topic>**
    - "http://localhost:3000/search?topic=Science"
- **GET /info/<id>**
    - "http://localhost:3000/info/1"
- **POST /purchase/<id>**
    - "http://localhost:3000/purchase/1”

6. Docker compose

The **docker-compose.yml** file starts and connects the three services used in this project: **Catalog**, **Order**, and **Front**. Run everything with one command from the project root:

docker compose up --build

Catalog

```yaml
services:
  catalog:
    build: ./Catalog_Service
    container_name: bazar_catalog
    ports:
      - "5142:5142"
    environment:
      - ASPNETCORE_URLS=http://+:5142
      - DOTNET_RUNNING_IN_CONTAINER=true
      - SQLITE_DB_PATH=/app/Data/bazar.db
    volumes:
      - ./Catalog_Service/Data:/app/Data
    restart: unless-stopped
```

- Path: **./Catalog_Service**
- Container name: **bazar_catalog**
- Ports: **5142** (host) → **5142** (container)
- .NET API that stores book data in an SQLite database and exposes **search/info/update** endpoints.
- the local database folder ./**Catalog_Service/Data** is mounted into the container so the **catalog.db** file persists across restarts. The service listens on **5142**.

Order

```yaml
order:
  build: ./Order
  container_name: bazar_order
  ports:
    - "3002:3002"
  environment:
    - CATALOG_URL=http://catalog:5142
  depends_on:
    - catalog
  volumes:
    - ./Order/data:/app/data
  restart: unless-stopped
```

- Path: **./Order**
- Container name: **bazar_order**
- Ports: **3002** (host) → **3002** (container)
- Node.js service that handles purchases (**POST /purchase/:id**). It calls the Catalog to decrement stock, logs bought book <title>, and saves orders to Order/data/orders.json.
- it uses **CATALOG_URL=http://catalog:5142** to reach the Catalog inside the Docker network. The order data folder is mounted so orders are persistent.

Front

```yaml
front:
  build: ./front
  container_name: bazar_front
  ports:
    - "3000:3000"
  environment:
    - CATALOG_URL=http://catalog:5142
    - ORDER_URL=http://order:3002
  depends_on:
    - order
    - catalog
  restart: unless-stopped
```

- Path: **./front**
- Container name: **bazar_front**
- Ports: **3000** (host) → **3000** (container)
- Flask gateway that forwards client requests to the backend services: /**search** and /**info** go to Catalog, /**purchase** goes to Order.
- environment variables **CATALOG_URL=http://catalog:5142** and **ORDER_URL=http://order:3002** ensure internal communication by service name.