I have a WPPA .NET application and I have a ACCA .NET cluster and my API application is one of the nodes in the ACCA cluster and what happens is whenever an API comes in and it acts as a proxy layer, the node in the API is acting as a proxy layer, whenever the request comes in, we just forward it to the other nodes in the cluster. So now my question is, I want to implement a scheduler system in my application and so I am planning that I will be installing Quartz.NET as a IPoster surface in my API and I have to implement a dynamically growing scheduler and that schedule I can add into the table and that schedule needs to be added to the scheduler so the Quartz.NET has the facility to store it in the DB by its own and it can store it and then this schedule will be taking it up, it will trigger and it will just send the message as it does when the API comes to the forwarder and then the actor will take it up. So this is what I am thinking of, is it a good architectural decision and I do not want to do the job because the job will be done by the ACCA.NET, I only need to do the triggering part, scheduling part, the processing part. So what are the advantages if I do this because my schedules will be like it should happen on every Monday at 8pm and it should happen on every Tuesday at 10pm except the market holidays. So this is the kind of schedules I will be having. So give me the implementation for this case and the user can anytime come in and apply some kind of cases like they can edit the schedule, they can add new schedule. So add new schedule means that schedule will be, user has to define that schedule, how that schedule will be attached to the background process of ACCA, that is the second part. So having said that, this is my thought of implementing. So when I have gone across, I have seen there is an iTimer implementation in ACCA.NET. So rather than using the Quartz.NET and this above mentioned implementation, is there any advantage or will it be better to use ACCA.NET iTimer. Okay, give me the complete comparison and what is the strength of using Quartz.NET and what is the strength of using iTimer and what are the cons also in both cases.

You're thinking in the **right architectural direction**, and your mental model is already close to what most large-scale .NET + Akka.NET (ACCA) systems do in production. I'll answer this in **four clear layers**, keeping your exact scenario in mind (API as proxy node, Akka cluster doing the work, scheduler only triggering).

1️⃣ Restating *your* problem (to align context)

You have:

- **WPPA .NET API**
  - Part of an **Akka.NET (ACCA) cluster**
  - Acts mainly as a **proxy / ingress layer**
  - Forwards requests to actors running anywhere in the cluster

You want:

- A **dynamic scheduler**
- Schedules defined by users and stored in DB
- Examples:
  - "Every Monday at 8 PM"
  - "Every Tuesday at 10 PM"
  - "Exclude market holidays"
- Scheduler should:
  - **Only trigger**
  - **NOT do the job**
  - Forward a message → Akka.NET actor
- Must support:
  - Add / edit / delete schedules at runtime

- Cluster-safe execution (only one trigger fires)
- Crash recovery
- No lost triggers

You're evaluating:

- **Quartz.NET (DB-backed scheduler)**
- **Akka.NET ITimer / Scheduler**

2️⃣ Is Quartz.NET as a trigger + Akka.NET as processor a good architecture?

✅ Yes — this is a very strong, production-grade architecture

This is a **separation-of-concerns** design:

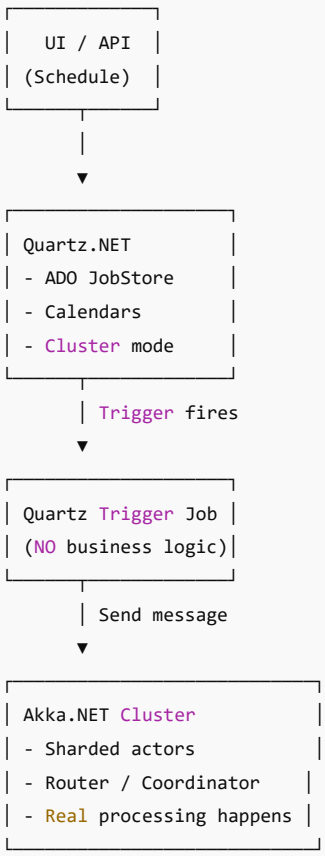
Responsibility	Technology
Scheduling / calendars / persistence	Quartz.NET
Trigger reliability & clustering	Quartz.NET
Business execution	Akka.NET
Scalability & backpressure	Akka.NET
Failover	Both

> **Quartz decides *when***
> **Akka decides *how***

This is exactly how **banking, trading, ETL, and data-ingestion platforms** are built.

3️⃣ Recommended architecture (high level)

pgsql



Copy code

4️⃣ How Quartz.NET fits your exact scheduling needs

✔ Cron schedules (Mon/Tue at specific times)

Quartz supports **Cron expressions** natively.

Example:

```text
0 0 20 ? * MON *
0 0 22 ? * TUE *
```

Copy code

✔ Market holidays (exception calendars)

Quartz has **Calendars**, which is a huge advantage.

```csharp
var holidayCalendar = new HolidayCalendar();
holidayCalendar.AddExcludedDate(new DateTime(2025, 1, 26)); // Example holiday

scheduler.AddCalendar(
    "MarketHolidays",
    holidayCalendar,
    replace: true,
    updateTriggers: true
);
```

Copy code

Attach it to trigger:

```csharp
.WithSchedule(
    CronScheduleBuilder
        .WeeklyOnDayAndHourAndMinute(DayOfWeek.Monday, 20, 0)
)
.ModifiedByCalendar("MarketHolidays")
```

Copy code

👉 **Akka.NET timers do NOT support calendars**

✔ Dynamic schedule updates

- User edits schedule
- Update DB
- Quartz:
  - `RescheduleJob`
  - `PauseTrigger`
  - `DeleteJob`
- No restart required

✔ Cluster-safe execution

With:

```csharp
.UsePersistentStore(store =>
{
    store.UseSqlServer(connectionString);
    store.UseClustering();
})
```

Copy code

Guarantee:

- Only **one node fires the trigger**

- If node dies → another node takes over
- Trigger is **never lost**

5️⃣ Minimal implementation (Quartz → Akka)

Quartz Job (Trigger-only job)

```csharp
public class AkkaTriggerJob : IJob
{
    private readonly ActorSystem _actorSystem;

    public AkkaTriggerJob(ActorSystem actorSystem)
    {
        _actorSystem = actorSystem;
    }

    public Task Execute(IJobExecutionContext context)
    {
        var scheduleId = context.JobDetail.JobDataMap.GetString("ScheduleId");

        var coordinator = _actorSystem.ActorSelection("/user/jobCoordinator");
        coordinator.Tell(new RunScheduledJob(scheduleId));

        return Task.CompletedTask;
    }
}
```

Copy code

👉 **No processing here**

Scheduling dynamically (API)

```csharp
public async Task AddSchedule(ScheduleDto dto)
{
    var job = JobBuilder.Create<AkkaTriggerJob>()
        .WithIdentity(dto.Id)
        .UsingJobData("ScheduleId", dto.Id)
        .Build();

    var trigger = TriggerBuilder.Create()
        .WithIdentity($"{dto.Id}-trigger")
        .WithCronSchedule(dto.Cron)
        .ModifiedByCalendar("MarketHolidays")
        .Build();

    await scheduler.ScheduleJob(job, trigger);
}
```

Copy code

6️⃣ How schedule attaches to Akka background processing

Inside Akka:

```csharp
public class JobCoordinatorActor : ReceiveActor
{
    public JobCoordinatorActor()
    {
        Receive<RunScheduledJob>(msg =>
        {
```

```
            // Route to shard / worker
            Context.ActorSelection("/user/jobRouter")
                   .Tell(new ExecuteJob(msg.ScheduleId));
        });
    }
}
```

Copy code

You can:

- Use **Cluster Sharding**
- Route based on `ScheduleId`
- Ensure **idempotency**

7️⃣ Now the big question: Quartz.NET vs Akka.NET ITimer

🪀 Feature-by-feature comparison

Feature	Quartz.NET	Akka.NET ITimer
Persistence	✅ DB-backed	❌ In-memory only
Crash recovery	✅ Yes	❌ Lost timers
Cron support	✅ Full	❌ Manual
Holiday calendars	✅ Built-in	❌ Custom code
Dynamic updates	✅ Native	⚠️ Restart actor
Cluster-safe	✅ Yes	❌ Per-node
Long intervals (days/weeks)	✅ Reliable	❌ Risky
Exactly-once trigger	✅ (clustered)	❌ No
Operational tooling	✅ Mature	❌ Minimal
Best use case	Scheduling	Short internal timers

8️⃣ When Akka.NET ITimer *is* the right choice

Use ITimer when:

- Timer is **short-lived** (seconds/minutes)
- Timer is **actor-local**
- Timer does NOT need persistence
- Example:
  - Retry in 10 seconds
  - Timeout after 30 seconds
  - Heartbeat pings

Example:

```csharp
Timers.StartSingleTimer(
    "retry",
    new RetryMessage(),
    TimeSpan.FromSeconds(10)
);
```

Copy code

9️⃣ Why ITimer is a bad fit for *your* case

Your requirements include:

❌ Long-running schedules (weekly, recurring)
❌ Business calendars
❌ Cluster failover
❌ DB reload after restart
❌ User-driven schedule changes

Trying to build this with ITimer means:

- Writing your **own scheduler**
- Writing **your own persistence**
- Writing **your own leader election**
- Writing **your own calendar logic**

👉 You'd end up re-implementing **Quartz poorly**

🔟 Final recommendation (clear and firm)

✅ Best Architecture for Your System

> **Quartz.NET for scheduling + Akka.NET for execution**

Why this wins:

- Clean separation of concerns
- Proven reliability
- Perfect match for cron + holidays
- Cluster-safe
- Operationally mature
- No trigger loss
- Akka focuses only on what it does best: **processing**

❌ Do NOT use Akka.NET ITimer for this use case

ITimer is **not a scheduler**, it is an **actor utility**.

1️⃣1️⃣ One-line architectural summary (you can reuse this)

> *"We use Quartz.NET as a clustered, persistent scheduling engine to reliably trigger time-based events, while Akka.NET handles the distributed, scalable execution of business workloads across the cluster."*

If you want next:

- DB schema design for schedules
- Quartz clustering tuning
- Idempotency patterns in Akka
- Leader-only scheduling vs all-node scheduling

Tell me 👍

So when I am horizontally scaling the API project, like API instance, like for example, the API instance, I have kept a background scheduling, right? So if I have three instances, how the behavior would be? Like how, when it is horizontally scaled, what is the right time to tell that? When multiple instances of API happens and each API has the hosted service up and running, now how the interaction will be? Like three will start triggering and how it will act?

This is a **very important question**, and you're asking exactly the right thing at the right time 👍
Let's go step-by-step and be *very precise*, because this is where many distributed systems go wrong.

I'll explain:

1. **What actually happens when you horizontally scale**
2. **What goes wrong if Quartz is not configured correctly**
3. **How Quartz behaves in clustered mode (the correct setup)**
4. **What exactly runs on each API instance**
5. **The "right time" and lifecycle for starting the scheduler**
6. **How this interacts with Akka.NET**

1️⃣ What happens when you scale your API horizontally?

Assume:

- You have **3 API instances**
- Each API instance has:
  - ASP.NET Core
  - `IHostedService`
  - Quartz.NET configured

Without any coordination

If you **naively** do this:

```csharp
services.AddQuartz();
services.AddQuartzHostedService();
```

👉 Then **each instance runs its own independent scheduler**.

So at **8 PM Monday**:

- Instance A triggers
- Instance B triggers
- Instance C triggers

❌ **Same schedule fires 3 times**
❌ **3 messages sent to Akka**
❌ **Duplicate processing**

This is the *default* behavior.

2️⃣ Why this happens (important mental model)

> Quartz by default is **process-local**, not cluster-aware.

Each API instance:

- Has its own memory
- Has its own scheduler thread
- Has no idea about other instances

So horizontal scaling **multiplies triggers** unless you stop it.

3️⃣ Correct behavior with Quartz clustering (THIS is what you want)

The key concept: Quartz Cluster Mode

When Quartz is configured with:

- **ADO JobStore**
- **Same database**
- **Clustering enabled**

Then:

> 💡 **All API instances share the same scheduler state**

What actually happens at runtime

Let's say:

- 3 API instances
- One Quartz trigger due at 8 PM

At 8 PM:

1. All 3 instances **wake up**
2. They all try to acquire the trigger
3. Quartz uses **DB row locking**
4. **Only ONE instance wins**
5. That instance executes the job
6. Others back off

✔ Exactly-once trigger
✔ No duplicates
✔ Automatic failover

4️⃣ How Quartz decides "which instance runs it?"

Internally Quartz uses:

- `QRTZ_TRIGGERS`
- `QRTZ_FIRED_TRIGGERS`
- Database locks (`SELECT FOR UPDATE` / row locks)

Each node has a unique:

```text
instanceId = AUTO
```

So:

- Instance A → wins lock → executes
- Instance B/C → see lock → skip

If A crashes mid-flight:

- DB heartbeat expires
- Another instance takes over

5️⃣ What exactly runs on each API instance?

Each API instance runs:

Component	Behavior
Quartz Scheduler	✅ Running
Quartz Job polling	✅ Running
Trigger execution	⚠️ Only ONE instance
Akka message send	✅ Only from winner
Akka cluster	✅ Independent

So yes:

- **All 3 schedulers are "active"**
- But **only one actually fires per trigger**

This is the correct and expected design.

6️⃣ Correct Quartz configuration for horizontal scaling

🔑 Mandatory settings

csharp Copy code

```csharp
services.AddQuartz(q =>
{
    q.UseMicrosoftDependencyInjectionJobFactory();

    q.UsePersistentStore(store =>
    {
        store.UseSqlServer(connectionString);
        store.UseClustering();

        store.UseJsonSerializer();
    });
});

services.AddQuartzHostedService(options =>
{
    options.WaitForJobsToComplete = true;
});
```

Appsettings (very important)

json Copy code

```json
"Quartz": {
  "quartz.scheduler.instanceName": "ClusteredScheduler",
  "quartz.scheduler.instanceId": "AUTO",
  "quartz.jobStore.type": "Quartz.Impl.AdoJobStore.JobStoreTX, Quartz",
  "quartz.jobStore.useProperties": false,
  "quartz.jobStore.dataSource": "default",
  "quartz.jobStore.clustered": true,
  "quartz.jobStore.clusterCheckinInterval": 15000
}
```

7️⃣ When should the scheduler start?

✔ Correct timing

Quartz should start:

- When the **API instance starts**
- Inside `IHostedService`
- Automatically joins the cluster

You do **not** need:

- Leader election
- Master node logic
- Custom coordination

Quartz handles it.

8️⃣ Interaction with Akka.NET (very important detail)

At trigger time:

pgsql Copy code

```
Quartz (one instance)
        ↓
Send message to Akka
```

```
        ↓
Akka cluster routes work
```

Akka does NOT care:

- Which API node sent the message
- How many API nodes exist

As long as:

- You send **one message per trigger**
- Your Akka job is **idempotent**

9️⃣ What if all 3 API instances crash?

Scenario	Result
API down, DB up	Triggers wait
API comes back	Quartz resumes
Missed triggers	Fired immediately or misfire rules
No data loss	✅

You can control this using **misfire policies**.

🔟 Common mistakes (avoid these)

❌ Using `RAMJobStore`
❌ Different DB per instance
❌ Same `instanceId` hardcoded
❌ Doing business logic inside Quartz job
❌ Starting Quartz multiple times manually

1️⃣1️⃣ Short, clear answer to your core question

> **When you horizontally scale your API, all instances run Quartz, but only one instance will fire a scheduled trigger at a given time when Quartz clustering is enabled. Quartz coordinates this via the shared database, ensuring exactly-once execution.**

1️⃣2️⃣ Final mental model (remember this)

> **Horizontal scaling does NOT mean "only one scheduler runs"**
> **It means "many schedulers compete, one executes."**

That's the right and safe way.

If you want next, I can:

- Show **misfire handling**
- Show **safe Akka idempotency**
- Explain **why not to elect a leader**
- Draw a **sequence diagram**

Just tell me 👍

Can I keep Redistore?

Short answer first:

> ❌ **You should NOT use Redis as the Quartz.NET job store.**
> ✅ **You CAN use Redis alongside Quartz, but not *as* Quartz's store.**

Now let me explain **why**, and then show you the **correct ways Redis *can* fit** into your architecture.

1️⃣ What Quartz *requires* from a JobStore

Quartz clustering depends on very specific guarantees:

Requirement	Why it matters
Strong consistency	Only ONE node must fire a trigger
Row-level locking	Prevent duplicate execution
Transactional safety	Trigger state must be atomic
Crash recovery	Dead node's work must be taken over
Misfire handling	Correct behavior after downtime

Quartz achieves this using:

- Database **transactions**
- **Pessimistic locking**
- **Heartbeat tables**

👉 These are **not optional**

2️⃣ Why Redis is NOT suitable as Quartz JobStore

Redis is:

- In-memory (even with AOF/RDB)
- Eventually consistent (in clusters)
- No row-level locking
- No transactional scheduler semantics

Problems if Redis were used as JobStore

Issue	What goes wrong
No real locks	Multiple API nodes fire same trigger
Network partitions	Split-brain scheduling
Failover timing	Duplicate or missed triggers
Misfire logic	Extremely complex to implement
Quartz internals	Expect SQL semantics

> Quartz's internal design **assumes SQL-like behavior**
> Redis **cannot provide that safely**

That's why:

- Quartz **does not officially support Redis**
- Community Redis stores are **experimental / unsafe**

3️⃣ The CORRECT way to use Redis in *your* architecture

You **absolutely can keep Redis**, just not for Quartz persistence.

✅ Recommended usage of Redis

Use case	Redis role
Schedule definitions cache	✅ Yes
Holiday calendar cache	✅ Yes
API-side fast reads	✅ Yes
Deduplication keys	✅ Yes
Akka coordination	✅ Yes
Quartz JobStore	❌ No

---

pgsql Copy code

```
┌──────────────┐
│   UI/API     │
└──────┬───────┘
       │
       ▼
┌──────────────────────┐
│ Redis                │
│ - Cached schedules   │
│ - Calendars          │
│ - Read-optimized     │
└──────┬───────────────┘
       │ (write-through)
       ▼
┌──────────────────────┐
│ SQL Database         │
│ - Quartz tables      │
│ - Source of truth    │
└──────┬───────────────┘
       │
       ▼
┌──────────────────────┐
│ Quartz.NET           │
│ - Clustered          │
│ - Exactly-once       │
└──────┬───────────────┘
       │ trigger
       ▼
┌──────────────────────┐
│ Akka.NET Cluster     │
│ - Processing         │
└──────────────────────┘
```

5️⃣ Concrete Redis usage patterns (recommended)

🔹 Pattern 1: Cache schedule definitions

text Copy code

```
Key: schedule:{scheduleId}
Value: cron + metadata
TTL: optional
```

Flow:

1. User updates schedule

2. Write to SQL (Quartz)
3. Update Redis
4. API reads Redis for UI / validation

Quartz **still reads SQL**

🔹 Pattern 2: Market holiday cache

Instead of querying DB every time:

```
text
Key: market:holidays:2025
Value: [dates...]
```

Copy code

On startup:

- Load holidays from Redis
- Update Quartz calendars

🔹 Pattern 3: Idempotency guard (VERY IMPORTANT)

Even with Quartz, always protect Akka execution:

```
text
Key: job-executed:{scheduleId}:{fireTime}
SETNX with TTL
```

Copy code

This gives:

- Extra safety
- Protection from edge cases
- Operational confidence

6️⃣ Why SQL JobStore + Redis cache is the best combo

Aspect	SQL	Redis
Source of truth	✅	❌
Trigger locking	✅	❌
Performance	❌	✅
Scalability	⚠️	✅
Reliability	✅	⚠️

👉 **Use each tool for what it's best at**

7️⃣ When would Redis-only scheduling ever make sense?

Only if:

- You build a **custom scheduler**
- You accept **at-least-once**
- You handle duplicates
- You don't need cron / holidays
- You don't need long-term reliability

⚠️ This is **NOT your case**

8️⃣ Final, clear recommendation

> **Do NOT use Redis as Quartz JobStore.**
> **Use SQL for Quartz clustering.**
> **Use Redis for caching, fast reads, and coordination.**

This gives you:

- Exactly-once triggers
- Horizontal scalability
- No duplicate firing
- Fast API responses
- Clean architecture

9️⃣ One-line takeaway (remember this)

> **Quartz needs a database; Akka needs messages; Redis needs to stay a cache.**

If you want next, I can:

- Show **exact Redis + Quartz code**
- Design **idempotency keys**
- Explain **misfire strategies**
- Compare **SQL Server vs Postgres for Quartz**

Just tell me 👍

So, give me clear cut, so I understood it's perfect, it's perfect, as per your suggestion also, it's perfect, go ahead with PODS.NET as triggering mechanism and ACCA.NET which takes up the job action. Okay, so now, my question is, like as I said, the scheduling is dynamic, right? So, now I want, I have a job schedule table where I can define the jobs. Okay, so like for example, I will define a job 1, like it should happen every Monday at 9 a.m. Okay, that is a job. Then, this job I want to attach to one of the, this job schedule I want to attach to one of the background process, which now in the system I have introduced as a new functionality for that particular long-running task. Okay, so now, as part of that, that particular long-running task, like for example, it is kind of a portfolio import. Okay, so this import process should happen every day. So, now this import process is for one particular adapter, and now I want to call that portfolio import actor. Okay, so this, this message or like I will, I can keep these details in the job definition table, right? And so, is there any kind of, I should be, I should be doing some kind of registration, right? Inside that, I job in implementation. So, the how that will be, consider there is an actor, and how that will be, how that piece of code will be.

Perfect 👍 you're aligned now.
I'll give you a **clear-cut, no-ambiguity design** and then **concrete code** so you can literally implement this.

I'll structure this into **5 parts**:

1. **Final architecture (authoritative)**
2. **DB model: JobSchedule vs JobDefinition**
3. **How "attaching a schedule to a background process" actually works**
4. **Registration pattern (THIS is the key you're asking about)**
5. **End-to-end code flow (Quartz → Akka Actor)**

No fluff, no theory loops.

1️⃣ Final architecture (lock this in)

> **Quartz.NET = Trigger engine**
> **Akka.NET = Job execution engine**

Quartz:

- Knows **WHEN**
- Knows **calendar / cron**
- Knows **cluster coordination**
- Does **ZERO business logic**

Akka:

- Knows **WHAT to run**
- Knows **HOW to run**
- Knows **WHERE to run (cluster routing)**

2️⃣ Database model (very important separation)

You need **two tables**, not one.

🔹 JobDefinition (WHAT)

```sql
JobDefinition
-------------
JobKey          (PK)    -- "PortfolioImport"
ActorType       -- "PortfolioImportActor"
MessageType     -- "RunPortfolioImport"
AdapterCode     -- "ZERODHA"
IsActive
MetadataJson
```

Copy code

This table defines **what job exists in the system**.

🔹 JobSchedule (WHEN)

```sql
JobSchedule
-----------
ScheduleId      (PK)
JobKey          (FK -> JobDefinition.JobKey)
CronExpression
CalendarName
TimeZone
IsActive
```

Copy code

This defines **when it runs**.

👉 One JobDefinition can have **many schedules**
👉 Schedules are **purely timing**

3️⃣ How “attaching a schedule to a background process” works

This is the **core mental model**:

> A schedule NEVER points directly to an actor
> A schedule points to a **JobKey**
> The JobKey is resolved at runtime to an actor + message

Flow at runtime

```pgsql
```

```
Quartz fires ScheduleId
        ↓
Load JobSchedule
        ↓
Resolve JobDefinition (JobKey)
        ↓
Create Akka message
        ↓
Send to actor
```

Copy code

This gives you:

- Dynamic binding
- No hard-coding
- Runtime extensibility

4️⃣ Registration pattern (THIS answers your main question)

Yes — **you need registration**, but **NOT in Quartz**.

You register **job handlers inside Akka**, not Quartz.

🔑 The correct pattern: Job Registry

🔹 Job registry interface

```csharp
public interface IJobHandler
{
    string JobKey { get; }
    void Handle(JobExecutionContext context);
}
```

Copy code

🔹 Concrete job handler (Portfolio Import)

```csharp
public class PortfolioImportJob : IJobHandler
{
    public string JobKey => "PortfolioImport";

    public void Handle(JobExecutionContext context)
    {
        context.ActorSystem
            .ActorSelection("/user/portfolio-import")
            .Tell(new RunPortfolioImport(
                context.AdapterCode,
                context.ScheduleId
            ));
    }
}
```

Copy code

🔹 JobExecutionContext

```csharp
public class JobExecutionContext
{
    public ActorSystem ActorSystem { get; init; }
```

```
    public string ScheduleId { get; init; }
    public string AdapterCode { get; init; }
    public DateTime FireTimeUtc { get; init; }
}
```

Copy code

🔹 Job registry (central piece)

csharp

```
public class JobHandlerRegistry
{
    private readonly Dictionary<string, IJobHandler> _handlers;

    public JobHandlerRegistry(IEnumerable<IJobHandler> handlers)
    {
        _handlers = handlers.ToDictionary(h => h.JobKey);
    }

    public IJobHandler Resolve(string jobKey)
    {
        if (!_handlers.TryGetValue(jobKey, out var handler))
            throw new InvalidOperationException($"No handler for job {jobKey}");

        return handler;
    }
}
```

Copy code

👉 This is **compile-time safety + runtime flexibility**

5️⃣ Quartz job (trigger-only, generic)

Quartz job does **not know anything about portfolio import**.

csharp

```
public class TriggerJob : IJob
{
    private readonly JobHandlerRegistry _registry;
    private readonly ActorSystem _actorSystem;
    private readonly IJobRepository _repo;

    public TriggerJob(
        JobHandlerRegistry registry,
        ActorSystem actorSystem,
        IJobRepository repo)
    {
        _registry = registry;
        _actorSystem = actorSystem;
        _repo = repo;
    }

    public async Task Execute(IJobExecutionContext context)
    {
        var scheduleId = context.JobDetail.JobDataMap.GetString("ScheduleId");

        var schedule = await _repo.GetSchedule(scheduleId);
        var definition = await _repo.GetJobDefinition(schedule.JobKey);

        var jobContext = new JobExecutionContext
        {
            ActorSystem = _actorSystem,
            ScheduleId = scheduleId,
            AdapterCode = definition.AdapterCode,
            FireTimeUtc = context.FireTimeUtc.UtcDateTime
```

```
        };

        var handler = _registry.Resolve(definition.JobKey);
        handler.Handle(jobContext);
    }
}
```

Copy code

---

6 Akka Actor (long-running work)

csharp

```
public class PortfolioImportActor : ReceiveActor
{
    public PortfolioImportActor()
    {
        Receive<RunPortfolioImport>(msg =>
        {
            // long-running import
            // idempotent by ScheduleId + FireTime
        });
    }
}
```

Copy code

---

7 How adding a NEW background job works (zero Quartz changes)

Example: "TradeReconciliation"

1. Add new actor
2. Add new `IJobHandler`
3. Insert JobDefinition row
4. Add JobSchedule row

❌ No Quartz job changes
❌ No scheduler restart
❌ No API redeploy required (if handler already deployed)

---

8 Why this design is *correct*

Concern	Solved
Dynamic schedules	✅
Dynamic job binding	✅
Cluster safe	✅
Extensible	✅
No hardcoding	✅
Clean responsibility	✅

---

9 Final one-paragraph summary (save this)

> *Quartz.NET acts as a clustered, persistent trigger engine that fires schedules stored in the database. Each trigger resolves a JobDefinition that maps to a registered Akka job handler. The handler constructs and sends a message to the appropriate Akka actor, which performs the long-running business operation. This design cleanly separates scheduling from execution, supports dynamic schedules, and scales safely across clustered API and Akka nodes.*

🔟 If you want next

I can:

- Add **misfire strategies**
- Show **idempotency design**
- Add **job versioning**
- Add **pause/resume per adapter**
- Add **multi-tenant schedules**

Just tell me which one 👍