Understood. This is the **right question**, and it explains why parts of the tutorial felt "mechanical" or confusing earlier.

What you are asking for is **Layer −1** (sometimes called *pre-framework mental prerequisites*).
This is **before Moqui, before XML, before screens, before entities**.

I will be strict, slow, and foundational.
No Moqui jargon until the ground is built.

# LAYER −1 — FOUNDATIONAL MENTAL MODELS (BEFORE ANY MOQUI LAYER)

If someone skips this layer, they **memorize Moqui**.
If someone understands this layer, they **understand Moqui**.

## 1. You must separate **DATA**, **BEHAVIOR**, and **PRESENTATION**

Most confusion comes from mixing these three.

### Data

- What exists
- What is stored
- What is persistent
- Independent of UI or logic

Example (conceptual):

"A Tutorial exists with an id and description."

Not:

- how it is created
- who can see it
- how it is displayed

### Behavior

- What *happens*
- Rules
- Operations
- Side effects
- Transitions

Example:

"Create a Tutorial"
"Validate a Tutorial"

“Delete a Tutorial”

Behavior **acts on data**, but is not data.

### Presentation

- How humans interact
- Screens
- Forms
- Buttons
- Layout

Example:

“Show a list of Tutorials”
“Provide a form to create a Tutorial”

Presentation **invokes behavior**, but should not contain logic.

### Why this matters

Moqui enforces this separation **aggressively**.

If you try to:

- put logic in screens → confusion
- put UI assumptions in services → rigidity
- put behavior in entities → chaos

Moqui will feel “weird” unless this separation is internalized.

## 2. You must understand **indirection** (this is the core concept)

Moqui is built almost entirely on **indirection**.

### What indirection means (plain language)

Instead of connecting A directly to B, we connect A → X → B.

Why?

- flexibility
- override capability
- late binding
- configuration instead of code

### Example you already saw (without realizing)

Instead of:

Screen → Permission

Moqui does:

Screen → ArtifactGroup → ArtifactAuthz → UserGroup

That is **intentional indirection**.

If you dislike indirection, Moqui will feel “overengineered”.
If you understand indirection, Moqui feels *powerful*.

## 3. You must abandon the idea that **files define behavior**

This is subtle and critical.

In Moqui:

- files do **not** define execution
- files define **metadata**
- execution is driven by **facades**

Example:

- Screen XML does not “run”
- Service XML does not “execute”
- Entity XML does not “create tables directly”

They are **inputs to engines**.

### Engines you must mentally accept:

- Screen Facade
- Service Facade
- Entity Facade
- Security Facade

Each engine:

- reads definitions
- validates
- resolves conflicts
- executes

If you think “this file runs”, you will be confused.

## 4. You must understand **late binding**

Late binding means:

The final decision is made as late as possible, not at compile time.

Moqui resolves:

- which service implementation to use
- which entity definition wins
- which screen is active
- which authorization applies

**at runtime**, not when files are written.

This is why:

- directory structure matters
- component load order matters
- overrides work without changing callers

## 5. You must stop thinking in “classes” and start thinking in **registries**

Java-trained developers struggle here.

Moqui does not think in:

- classes
- constructors
- object graphs

Moqui thinks in:

- registries
- definitions
- lookups
- resolution rules

Example:

- Services are registered by **name**
- Entities are registered by **entity-name**
- Screens are registered by **location**

Everything is:

“Ask the facade to resolve X”

## 6. You must understand **“describe, don’t control”**

Moqui is not imperative.

You do not say:

- open transaction
- begin query
- commit

You describe:

- what data you want
- what operation you want
- what screen structure you want

Moqui controls:

- transaction boundaries
- error handling
- retries
- security checks

This is why words like *declarative* matter.

## 7. You must understand **framework responsibility vs application responsibility**

This is a huge divider.

### Application responsibility

- Domain concepts
- Business rules
- Screens
- Services
- Custom logic

### Framework responsibility

- Transactions
- Security enforcement
- Data integrity
- Caching
- Execution order

Moqui is opinionated:
**you do not get to control framework responsibilities**.

Trying to do so leads to frustration.

## 8. You must understand **configuration > code**

Moqui prefers:

- XML over Java
- data over hardcoding
- configuration over annotations

Why?

- runtime changeability
- customer-specific overrides
- upgrade safety

This is why:

- security is data-driven
- services can be overridden
- entities can be extended

## 9. You must understand **override, not modify**

This is non-negotiable in Moqui.

Correct mindset:

“I add another definition that wins.”

Wrong mindset:

“I change the original file.”

This applies to:

- services
- entities
- screens
- data
- security

Everything is designed to be **layered**, not edited.

## 10. You must accept that **directory structure is semantic**

This was a major source of your errors — and a good learning moment.

In Moqui:

- directories are not cosmetic
- they define namespaces
- they define resolution order

- they define override boundaries

This is why:

- service location matters
- screen hierarchy matters
- component name matters

If you treat directories casually, Moqui will punish you.

## 11. You must understand **“tools exist because complexity exists”**

The Tools app is not optional fluff.

It exists because:

- runtime state is complex
- multiple definitions coexist
- resolution is dynamic

So:

- Tools → Services tells you what is *actually registered*
- Tools → Entities tells you the *final resolved model*
- Tools → Security tells you what *actually applies*

If you don’t use Tools, you are blind.

## 12. Final grounding sentence (this anchors everything)

Before reading **any Moqui layer**, you should internalize this:

“Moqui is a declarative, metadata-driven framework where behavior is resolved at runtime by facades using layered definitions and indirection.”

If that sentence feels *obvious* instead of *abstract*, you are ready.

## What happens next (important)

Now that **Layer –1 is complete**, everything you learned before will suddenly feel clearer.

If you want, next we can:

1. Re-walk **Screen Layer** in 30% less time (because foundations are set), or
2. Produce a **final consolidated mental map** that connects all layers together, or
3. Convert everything into **interview-ready explanations** with failure cases.

Tell me **which one**.

Good. We will proceed layered, deep, and sequential, and we will finish one layer completely before moving to the next. No skipping, no abstraction jumps.

Even though you listed
Screen → Entity → Service → Data → Security,
those layers sit on top of something more fundamental.

So the true foundation layer is:

Component & Artifact Architecture (how Moqui even knows what exists)

If this layer is not solid, everything else feels magical.
So we start here.

LAYER 0 — FOUNDATION

Component & Artifact Architecture (Moqui's spine)

This layer answers:

- *What exists?*
- *Where does Moqui look?*
- *How does it decide what wins when there are conflicts?*

0.1 What a Component is (definition)

In Moqui, a component is:

A self-contained unit of artifacts (screens, entities, services, data, scripts) that can be loaded, overridden, or replaced as a whole.

Important:
A component is not just a folder.
It is a namespacing + override boundary.

0.2 Why components exist (what breaks without them)

Without components:

- all screens would live in one global namespace
- services would collide
- upgrades would overwrite custom code
- extensions would require modifying core framework files

Components solve:

- isolation
- override
- upgrade safety
- modularity

This is why HotWax / Mantle / POPC exist as components, not projects.

0.3 Component directory = component identity

Your component:

runtime/component/tutorial/

This immediately defines:

- component name = tutorial
- artifact namespace root = tutorial

No XML needed for this.
Directory name is identity.

If you rename the folder, you rename the component.

0.4 Standard component structure (why these folders exist)

tutorial/

├── MoquiConf.xml

├── data/

├── entity/

├── screen/

├── service/

├── script/

Each directory maps to an artifact type.

This is not optional.

Moqui scans these directories using convention, not configuration.

0.5 Artifacts (very important term)

An artifact is:

Any framework-recognized unit that Moqui can load, reference, secure, and execute.

Artifacts include:

- Screens
- Entities
- Services
- Entity data
- Scripts

Artifacts:

- have identity
- can be authorized
- can be overridden
- can be referenced by other artifacts

This is why security talks about ArtifactAuthz, not “screen auth”.

0.6 How Moqui discovers artifacts (startup flow)

At startup, Moqui roughly does this:

1. Scan runtime/component/*
2. For each component:
   - read component.xml (if present)
   - read MoquiConf.xml fragments
3. Build:
   - component dependency graph
   - artifact load order
4. Index artifacts by:
   - type
   - name
   - component
5. Resolve conflicts based on load order

Key point:

Artifacts are indexed before they are executed.

That’s why your service existed in Tools → Services
even when your screen couldn’t call it.

0.7 Role of MoquiConf.xml (critical)

MoquiConf.xml is not optional decoration.

It is used to:

- extend global configuration
- mount screens

- define screen roots
- influence load order indirectly

Example you used:

```
<screen-facade>

  <screen location="component://webroot/screen/webroot/apps.xml">

    <subscreens-item name="tutorial"

                location="component://tutorial/screen/tutorial.xml"/>

  </screen>

</screen-facade>
```

What this actually does:

- says: “this screen exists”
- says: “this screen belongs in the screen tree”
- assigns URL path
- assigns menu entry

Without this:

- screen file exists
- but is unreachable

This is why screens ≠ services in resolution.

0.8 Why services don’t need MoquiConf.xml

You observed this correctly.

Services:

- are not navigational
- are invoked, not visited
- are resolved at call-time

So Moqui:

- scans service/
- indexes services automatically
- exposes them to Service Facade

Screens:

- must be mounted
- must be placed in hierarchy

- must be reachable via URL

Different lifecycle → different rules.

0.9 Artifact identity vs file location (your key confusion earlier)

Screens

Identity comes from:

- screen tree position
- file name
- subscreen hierarchy

Services

Identity comes from:

- directory path under service/
- XML file name
- <service verb="" noun="">

Example:

service/tutorial/TutorialService.xml

Service name prefix becomes:

tutorial.TutorialService

Then combined with:

create#Tutorial

Final service name:

tutorial.TutorialService.create#Tutorial

When you called:

TutorialService.create#Tutorial

It worked because:

- you were already inside the same component
- Moqui resolves locally first

This is intentional scoping, not magic.

0.10 Override and precedence (why components matter)

If two components define:

tutorial.entity.Tutorial

Then:

- Moqui merges definitions
- conflicts resolved by component load order
- later component overrides earlier one

This is how:

- custom components override base ones
- HotWax extends Mantle
- Mantle extends Moqui

This cannot work without components.

0.11 Why entity collisions created one table (your experiment)

You observed:

two entities with same name but different packages → one table

Reason:

- entity-name defines table identity
- package is logical namespace, not physical
- Moqui assumes extension, not duplication

This is deliberate to support:

- incremental extension
- field additions
- behavior overrides

Base entity = first loaded definition
Extensions = later-loaded components

0.12 Minimal mental model (lock this in)

Components define who owns artifacts
Artifacts define what exists
MoquiConf defines how screens are reached
Services and entities are discovered automatically

END OF LAYER 0

We are done with the foundation.

Nothing else should feel mysterious now:

- not service resolution
- not entity merging
- not screen mounting

Next step (required to proceed)

Confirm explicitly:

“Proceed to Layer 1 — Screen Layer”

Once you confirm, we will go extremely deep into:

- <screen>
- <actions>
- <widgets>
- <form-list>
- search behavior confusion you had
- why Find Options felt like “nothing happens”

Do not skip this confirmation.

# Screen

Good. We proceed.

From here onward, **everything you saw in the UI will stop feeling magical**.
We will finish the **Screen Layer completely** before touching Entity or Service again.

I will strictly follow this flow for the Screen Layer:

**definition → concrete mental model → tags → attributes → execution flow → why your confusion happened**

No frontend assumptions.

# LAYER 1 — SCREEN LAYER (XML Screen)

## 1.1 What a “Screen” actually is (definition)

In Moqui, a **screen** is:

A declarative description of **what data is prepared**, **what UI widgets are rendered**, and **what transitions are allowed**, assembled into a navigable node in the screen tree.

A screen is **not**:

- a page template
- a controller
- a view only

It is a **composed artifact**.

## 1.2 Mental model (this is critical)

Think of a screen as having **three phases**, executed **in order**:

1. **Actions phase**
   → Prepare data (queries, calculations, context variables)
2. **Widgets phase**
   → Render UI using the prepared data
3. **Transitions phase**
   → Define what can happen *after* user interaction

Nothing else happens.

If something is not in one of these phases, **it does not execute**.

## 1.3 The <screen> tag (root)

<screen>

This defines:

- one screen node
- one execution boundary
- one context scope

Every variable created inside this screen:

- lives for this request
- is passed to widgets
- is passed to transitions (unless overridden)

## 1.4 Screen execution order (important)

Even though XML order may differ, **logical order is fixed**:

1. <transition> definitions (registered)
2. <actions> executed
3. <widgets> rendered
4. <subscreens-active> rendered (if present)

This explains many “why didn’t it run?” moments.

# 1.5 <actions> — data preparation layer

## Definition

<actions> is where you **prepare data**, not display it.

Example you used:

```
<actions>
  <entity-find entity-name="tutorial.entity.Tutorial" list="tutorialList">
    <search-form-inputs/>
  </entity-find>
</actions>
```

## What this really does

- Executes **before UI is rendered**
- Runs an entity query
- Stores result in **context variable** tutorialList

No UI. No buttons. No interaction.

# 1.6 <entity-find> (screen-level query)

## Definition

<entity-find> is a declarative query instruction that executes via the Entity Facade and places results into the screen context.

Key attributes:

- entity-name → which entity
- list → where results are stored

This is equivalent to:

```
ec.entity.find("tutorial.entity.Tutorial").list()
```

...but declarative and inspectable.

# 1.7 <search-form-inputs> (your biggest confusion)

## What it is NOT

It is **not**:

- a submit button
- a search trigger
- a frontend handler

### What it actually is

It maps incoming **request parameters** to query conditions automatically.

Meaning:

- if request has description=abc
- it becomes WHERE description LIKE '%abc%'

It assumes:

**The page reload itself is the submission**

There is no explicit “Apply” button unless you add one.

## 1.8 Why your Find dialog felt “dead”

Now the key insight.

You used:

<form-list header-dialog="true" skip-form="true">

This caused **three things simultaneously**:

1. header-dialog="true"
   - find fields moved into a popup
2. skip-form="true"
   - **no HTML <form> tag is generated**
3. No submit button
   - no request reload trigger

So:

- values change in UI
- **no request is sent**
- actions never rerun
- results never change

Nothing was broken.
You told Moqui **not to generate a form**.

## 1.9 <widgets> — rendering phase

Widgets **never fetch data**.

They only:

- read context
- render UI
- bind fields

Your example:

<widgets>

<form-list name="ListTutorials" list="tutorialList">

<auto-fields-entity .../>

</form-list>

</widgets>

This means:

- read tutorialList
- render rows
- render header fields (if defined)

## 1.10 <form-list> (list renderer, not form logic)

### Definition

<form-list> renders a tabular view over a list of records.

Despite the name, it:

- may or may not generate an HTML <form>
- is primarily a **list renderer**

Key attributes you used:

- list → which context variable
- header-dialog="true" → move find fields into popup
- skip-form="true" → do not generate <form>

## 1.11 <auto-fields-entity> (schema → UI mapping)

### Definition

Automatically generates UI fields based on entity definition.

Your usage:

<auto-fields-entity

```
    entity-name="tutorial.entity.Tutorial"

    field-type="find-display"/>
```

This generates:

- find inputs (header)
- display fields (rows)

No logic. No persistence. Only mapping.

## 1.12 Field override (explicit <field>)

You later added:

```
<field name="description">

    <header-field>

        <text-find hide-options="true"/>

    </header-field>

    <default-field>

        <display/>

    </default-field>

</field>
```

What this does:

- overrides the auto-generated field
- keeps same data binding
- customizes UI behavior only

This does **not** affect queries directly.

## 1.13 <transition> — navigation + behavior

### Definition

A transition defines **what happens when a user submits or triggers an action**.

Example:

```
<transition name="createTutorial">

    <service-call name="TutorialService.create#Tutorial"/>
```

<default-response url="."/>

</transition>

This defines:

- a callable endpoint
- optional service execution
- navigation response

Transitions are **not executed automatically**.
They execute **only when invoked**.

## 1.14 <form-single> → transition binding

<form-single transition="createTutorial">

This is the moment:

- UI → behavior is connected
- submit → service-call happens

Without:

- no transition
- no service
- no state change

## 1.15 Why screen felt “static” before

Because:

- actions run only on request
- no submit
- no reload
- no transition

So:

UI updated locally, server never re-executed actions.

Correct behavior, confusing at first.

## 1.16 One-sentence truth (lock this in)

**Screens prepare data once per request; widgets only render; transitions are the only way to change state or trigger services.**

## END OF SCREEN LAYER

You now understand:

- why Find felt broken
- why no Apply button existed
- why skip-form matters
- why search-form-inputs is passive
- how screen execution really works

### Next required confirmation

Say explicitly:

“Proceed to **Layer 2 — Entity Layer**”

We will then dissect:

- entity definition
- primary keys
- sequencing
- entity merging
- your duplicate-entity experiment
- why tables merged

Do not skip.

Good. We proceed to the **Entity Layer**.

This is the layer where **most people lose correctness**, because entities *look* simple but carry the **strongest guarantees in Moqui**. We will go slowly and finish it fully.

I will follow this exact structure:

1. What an Entity **is** (definition, not analogy)
2. Why entities exist (what breaks without them)
3. Entity identity (the source of most of your confusion)
4. Entity definition anatomy (every tag and attribute you used)
5. Primary keys, sequencing, and lifecycle
6. Entity extension & merge (your duplicate-entity experiment)
7. Entity Facade and why everything goes through it

No services, no screens, no data loading yet. Only entities.

# LAYER 2 — ENTITY LAYER

## 2.1 What an Entity is (precise definition)

In Moqui, an **entity** is:

A declarative definition of a **persistent data model**, including its fields, keys, constraints, and lifecycle behavior, managed exclusively through the Entity Facade.

Important consequences:

- an entity is **not a table**
- an entity is **not a class**
- an entity is **not ORM boilerplate**

It is a **contract**.

## 2.2 Why entities exist (what breaks without them)

Without entities:

- every service would write SQL
- every screen would know database structure
- schema changes would break everything
- security, caching, validation would fragment

Entities centralize:

- schema
- constraints
- identity
- lifecycle hooks

This is why Moqui is **entity-centric**, not service-centric.

## 2.3 Entity identity (this is critical)

An entity is identified by **entity-name**, not file name, not package, not folder.

Example:

<entity entity-name="Tutorial">

This defines the **entity identity**.

Now the namespace:

tutorial.entity.Tutorial

Breakdown:

- tutorial → component
- entity → artifact type
- Tutorial → entity-name

**Only Tutorial is the actual entity identity.**

Everything else is **scoping**.

This is why your experiment resulted in **one table**.

## 2.4 Why two entities with same name merged

You tested:

- tutorial.entity.Tutorial
- tutorial.entities.Tutorial

Same entity-name → same entity.

Moqui interprets this as:

“This entity is being extended, not duplicated.”

That is **by design**.

## 2.5 Base entity vs extension (how Moqui decides)

The **base entity** is:

- the first definition loaded (by component order)

Extensions:

- later definitions
- add fields
- override attributes
- refine constraints

This is controlled by:

- component load order
- depends-on in component.xml

There is **no separate syntax** for “base entity”.

Order defines authority.

## 2.6 Entity definition file anatomy

A typical entity file you used:

<entities>

  <entity entity-name="Tutorial" package-name="tutorial.entity">

```
        <field name="tutorialId" type="id"/>

        <field name="description" type="text-short"/>

        <prim-key field="tutorialId"/>

    </entity>

</entities>
```

Now we dissect every part.

## 2.7 <entities> (container only)

- groups entity definitions
- no behavior
- no execution

Purely structural.

## 2.8 <entity> tag

Attributes:

- entity-name → identity (mandatory)
- package-name → logical namespace

Important:

package-name does **not** affect table identity.

It exists for:

- clarity
- conflict avoidance
- code readability

## 2.9 <field> tag

Defines a **column contract**, not just a column.

Example:

<field name="tutorialId" type="id"/>

What this implies:

- persistent
- participates in PK
- indexed by default

- eligible for sequencing

Types you used:

- id
- text-short

Types are **semantic**, not SQL types.

Moqui maps them per database.

## 2.10 <prim-key> (identity guarantee)

<prim-key field="tutorialId"/>

This defines:

- record identity
- update/delete resolution
- service auto behavior
- cache key

Without a primary key:

- entity cannot be updated
- entity-auto services cannot work
- cache integrity breaks

This is why Moqui is strict here.

## 2.11 Sequencing (how IDs appear magically)

When you did:

if (!tutorial.tutorialId)

  tutorial.setSequencedIdPrimary()

or used entity-auto services

Moqui:

- checks PK
- if missing
- uses sequence keyed by entity-name

Sequence key = Tutorial

This is why **entity-name stability matters**.

## 2.12 Entity lifecycle (important)

Every entity operation goes through:

- validation
- EECA rules
- transaction management
- caching
- auditing

This is enforced by the **Entity Facade**.

There is no supported bypass.

## 2.13 Entity Facade (why everything goes through it)

Definition:

The Entity Facade is the runtime system that manages entity definitions, persistence, caching, validation, and lifecycle events.

What it does:

- loads entity definitions
- merges extensions
- generates schema
- executes queries
- fires EECA rules

This is why:

- screens use <entity-find>
- services use entity-auto
- scripts use ec.entity

All roads lead here.

## 2.14 Why entity-auto services work at all

Because:

- entity definition exists
- entity facade understands schema
- lifecycle rules are known

Service runner just delegates.

No entity → no auto service.

## 2.15 Your duplicate-entity experiment (final verdict)

What you observed was **correct behavior**.

Moqui assumed:

- same entity
- extension intent
- merged schema

What would break it:

- conflicting PK definitions
- incompatible field types
- incompatible constraints

In those cases:

- startup error
- schema validation failure

## 2.16 One-sentence truth (lock this in)

**Entities define identity and lifecycle; tables are just a physical side-effect.**

# END OF ENTITY LAYER

You now fully understand:

- why tables merged
- what base entity means
- how identity is decided
- why entity-name is sacred
- why everything routes through Entity Facade

### Required confirmation to proceed

Say explicitly:

“Proceed to **Layer 3 — Service Layer**”

Next layer will fully resolve:

- implicit vs explicit services
- entity-auto runner
- in-map confusion
- override and versioning
- why your service call failed and then worked

No shortcuts.

Good. We proceed to **Layer 3 — Service Layer**.

This layer is where **behavior lives**. If the Entity layer defines *what exists*, the Service layer defines *what happens*. Most of your earlier confusion (implicit vs explicit, entity-auto, directory structure, service resolution, in-map, override) lives entirely here.

We will complete this layer fully before touching Data or Security.

I will follow this exact flow:

1. What a service **is** (precise definition)
2. Why services exist (what breaks without them)
3. Service identity and naming (your biggest pain point)
4. The three types of services (implicit, explicit entity-auto, explicit custom)
5. EntityAutoServiceRunner (how entity-auto really works)
6. Service call mechanics (service-call, in-map)
7. Override, precedence, and versioning (what "overridable" actually means)
8. How to *know* which service is being called

# LAYER 3 — SERVICE LAYER

## 3.1 What a Service is (definition)

In Moqui, a **service** is:

A named, callable unit of business behavior that executes within a transaction and operates on entities through the framework, not directly on the database.

Key implications:

- services are **behavior-focused**
- services are **transactional**
- services are **framework-controlled**
- services are **addressable by name**

A service is closer to a **use-case function**, not a helper method.

## 3.2 Why services exist (what breaks without them)

Without services:

- screens would contain business logic
- scripts would perform persistence directly
- transactions would be inconsistent
- security could not be enforced at behavior level

- reuse would collapse

Services give Moqui:

- a stable behavior boundary
- a place to attach security
- a transaction scope
- override points

This is why **screens never create entities directly**; they call services.

## 3.3 Service identity (this is critical)

A service is identified by:

[optional prefix].[service-file].[verb]#[noun]

Example you used:

TutorialService.create#Tutorial

Breakdown:

- TutorialService → service definition file (without .xml)
- create → verb
- Tutorial → noun

Important correction (based on your resolved bug):

**The component name is NOT part of the service name unless you explicitly include it.**

So this failed:

tutorial.TutorialService.create#Tutorial

And this worked:

TutorialService.create#Tutorial

Because:

- service resolution is **component-local first**
- Moqui searches within the current component
- only then searches dependencies

This is intentional scoping, not a mistake.

## 3.4 Why directory structure matters for services

Your structure:

service/tutorial/TutorialService.xml

This does **not** create tutorial.TutorialService.

It creates:

TutorialService

The subfolder (tutorial/) is organizational, not namespacing.

This differs from entities and screens deliberately.

Why?

Because services are:

- behavior units
- often overridden
- meant to be swapped transparently

So Moqui avoids deep namespacing by default.

## 3.5 The three types of services (very important)

### Implicit entity-auto service

You never define it.

Example call:

create#tutorial.entity.Tutorial

What happens:

- Service Facade sees create
- Sees entity name
- Delegates to EntityAutoServiceRunner
- Behavior derived entirely from entity definition

No XML. No file. No implementation.

### Explicit entity-auto service

You define it, but **do not implement logic**.

Your example:

<service verb="create" noun="Tutorial" type="entity-auto">

  <in-parameters>

```
    <auto-parameters include="all"/>
  </in-parameters>
  <out-parameters>
    <auto-parameters include="pk" required="true"/>
  </out-parameters>
</service>
```

What this does:

- declares the service explicitly
- exposes it in tools
- allows security, override, documentation
- still delegates execution to EntityAutoServiceRunner

Behavior is **derived**, not written.

### Explicit custom service (inline / script / Java)

Example (inline):

```
<service verb="create" noun="Tutorial" type="inline">
  <actions>
    <entity-make-value .../>
    <entity-set .../>
    <entity-create .../>
  </actions>
</service>
```

Here:

- you fully control behavior
- you still must use Entity Facade
- you still get transactions, EECA, validation

## 3.6 EntityAutoServiceRunner (the “engine”)

You asked: *“How does entity-auto know what to do?”*

Answer:

EntityAutoServiceRunner derives behavior by inspecting the entity definition and the service verb.

For create:

- reads entity PK
- sequences ID if missing
- validates fields
- executes create
- returns PK

For delete:

- expects PK
- enforces FK rules
- executes delete

No magic. Just rules.

This is why:

- entity definition must be correct
- PK must exist
- types must align

## 3.7 auto-parameters (derived contract)

When you write:

<auto-parameters include="all"/>

You are saying:

“Derive this service’s parameters from the entity definition.”

This gives you:

- declarative contract
- schema-driven behavior
- automatic updates when entity changes

This is what “derived” actually means.

## 3.8 service-call (invocation)

Example:

<service-call name="TutorialService.create#Tutorial"/>

What happens at runtime:

1. Screen transition triggers
2. Service Facade resolves service name
3. Service definition is located
4. Input parameters are mapped
5. Transaction starts
6. Service runner executes
7. Output is returned
8. Response is applied

## 3.9 in-map="context" (parameter mapping)

This tells Moqui:

“Use the screen context as the input map for this service.”

Meaning:

- form fields
- request parameters
- variables from actions

All flow into the service.

Without in-map:

- Moqui uses default mappings
- often sufficient for entity-auto services
- explicit mapping needed for complex cases

## 3.10 Override and “overridable” (correcting your hypothesis)

You were **directionally correct**, but not precisely.

Services are overridable because:

- service identity is stable
- later-loaded components can redefine the same service
- Service Facade uses **last definition wins**

This is **not polymorphism**, but **replacement**.

No inheritance. No super call (unless you do it manually).

Versioning is typically handled by:

- separate service names
- or separate components

## 3.11 How to know which service is being called (practical rule)

To know which service executes:

1. Look at the **service name** in service-call
2. Resolve it **locally** (current component)
3. Check service tools UI
4. Check component load order
5. Last loaded definition wins

If it appears in Tools → Services and works there, it exists.

If screen fails to call it:

- name mismatch
- scoping mismatch
- transition not invoked

Exactly what you debugged.

## 3.12 One-sentence truth (lock this in)

**Services define behavior boundaries; entity-auto services derive behavior from entity definitions; override is replacement, not inheritance.**

# END OF SERVICE LAYER

You now fully understand:

- why your service call failed
- why directory structure mattered
- how entity-auto works
- what “derived” actually means
- what overridable really implies

### Required confirmation to proceed

Say explicitly:

“Proceed to **Layer 4 — Data Layer (entity-facade-xml)**”

This next layer will fully close:

- declarative vs imperative
- environment-specific loading
- validation & EECA participation
- why seed-data is discouraged
- how your security data was loaded

No skipping.

Good. We proceed to **Layer 4 — Data Layer (entity-facade-xml)**.

This layer answers the questions you explicitly struggled with:

- *What does "declarative" actually mean here?*
- *What is environment-specific loading?*
- *How does data "participate in validation and EECA"?*
- *Why entity-facade-xml exists when seed-data already exists?*

We will finish this layer completely and cleanly.

I will follow the same disciplined flow:
definition → concrete example → execution flow → why it exists → what breaks without it.

# LAYER 4 — DATA LAYER (ENTITY-FACADE-XML)

## 4.1 What the Data Layer is (definition)

In Moqui, the **Data Layer** refers to:

Declarative loading of entity records through the Entity Facade, using XML files that describe *what data should exist*, not *how to insert it*.

The primary mechanism for this is:

<entity-facade-xml>

This is **not schema** and **not behavior**.
It is **state definition**.

## 4.2 What "declarative" means (no buzzwords)

You asked specifically about this word, so here is the exact meaning.

Declarative means:

You describe the **desired end state**, and the framework decides the steps to reach it.

Imperative (what you already know from Java / SQL):

INSERT INTO table (...) VALUES (...);

Declarative (entity-facade-xml):

<moqui.security.ArtifactGroup artifactGroupId="TUT_APP"/>

You are not saying:

- how to open a transaction
- how to check if it exists
- whether to insert or update
- how to handle constraints

You are saying:

“This record should exist.”

Moqui figures out the rest.

That is **declarative**.

## 4.3 What <entity-facade-xml> actually is

Precise definition you can say:

entity-facade-xml is a data definition format that instructs the Entity Facade to create, update, or manage entity records using normal entity rules.

Important:

- it is processed by **EntityFacade**
- not raw SQL
- not bypass logic
- not a special loader

This is why it integrates cleanly with the rest of the system.

## 4.4 Why the Entity Facade is mandatory here

Every record loaded via entity-facade-xml goes through:

- primary key checks
- foreign key validation
- data type validation
- sequencing rules
- EECA (Entity Event Condition Actions)
- transactions
- cache updates

This answers your question:

### “What does ‘participates in validation and EECA’ mean?”

It means:

Data loaded this way behaves exactly like data created by a service at runtime.

If there is an EECA rule:

- on create
- on update
- on delete

It **will fire**.

Seed-data does **not** give this guarantee.

## 4.5 <entity-facade-xml> structure

A typical file you used:

<entity-facade-xml type="seed-initial">

<moqui.security.ArtifactGroup artifactGroupId="TUT_APP"/>

<moqui.security.ArtifactAuthz .../>

</entity-facade-xml>

Let’s break this down.

## 4.6 The type attribute (environment-specific loading)

This is where “environment-specific” becomes concrete.

Examples:

- seed-initial
- seed-demo
- test
- bootstrap
- production

What this means:

Moqui decides **when** to load this data based on runtime configuration and database state.

For example:

- seed-initial → loaded only when DB is empty
- seed-demo → loaded for demo/dev environments
- test → loaded only during test runs

This prevents:

- demo data in production
- reloading config on every startup
- accidental overwrites

This is **environment-specific loading**.

Not magic. Controlled.

## 4.7 Why this cannot be done with seed-data

Now the direct comparison.

### seed-data (inside entity file)

- always tied to schema
- always loaded with entity
- no environment awareness
- no lifecycle control
- not scalable

### entity-facade-xml

- separated from schema
- loadable by type
- environment-aware
- reloadable
- overridable
- transactional

This is why Moqui discourages seed-data except for **very small defaults**.

## 4.8 Why security data must use entity-facade-xml

You loaded:

- ArtifactGroup
- ArtifactGroupMember
- ArtifactAuthz

These are **system configuration**, not domain data.

They must:

- load before users interact
- be overrideable
- survive upgrades
- be environment-aware

Embedding this inside entity files would be wrong.

Hence entity-facade-xml.

## 4.9 Execution flow (what actually happens at runtime)

When you run:

gradlew load

Moqui does this:

1. Scan all components
2. Discover entity-facade-xml files
3. Group them by type
4. Decide which types to load (based on config)
5. For each record:
    - resolve entity definition
    - validate fields
    - apply sequencing
    - execute create/update
    - fire EECA
    - commit transaction

Nothing is bypassed.

## 4.10 Diff-friendly and inspectable (your last missing terms)

You also asked about these terms earlier.

### Diff-friendly

Because:

- XML records are explicit
- no generated SQL
- no timestamps unless defined
- stable ordering

You can:

- review changes in Git
- see exactly what data changed

### Inspectable

Because:

- Tools → Entity Data
- Tools → Entity Definitions
- Tools → Logs

You can see:

- what was loaded
- when
- how
- by which file

This is impossible with raw SQL scripts.

## 4.11 One-sentence truth (lock this in)

**entity-facade-xml defines system and demo data declaratively so it loads safely, predictably, and consistently across environments using normal entity rules.**

# END OF DATA LAYER

You now fully understand:

- what declarative means (for real)
- what environment-specific loading actually is
- why validation and EECA matter
- why seed-data is limited
- why entity-facade-xml is preferred

### Required confirmation to proceed

Say explicitly:

“Proceed to **Layer 5 — Security Layer**”

This will complete the entire quick-tutorial mental model and close the loop on:

- ArtifactGroup
- ArtifactAuthz
- why screens were blocked
- how authorization actually works

Once that is done, we will produce your **final consolidated notes**.

Good. We proceed to **Layer 5 — Security Layer**.

This layer is where most Moqui beginners get confused because **security is not attached to screens or services directly**. It is attached to **artifacts via indirection**. We will remove all ambiguity.

I will be strict and foundational.

# LAYER 5 — SECURITY LAYER (Artifact Authorization)

## 5.1 What the Security Layer is (definition)

In Moqui, the security layer answers exactly one question:

**Is the current user allowed to perform a specific action on a specific artifact?**

Important:

- Not “can the user see this screen?”
- Not “is the user logged in?”
- Not “does the screen say require-authentication?”

Security is evaluated **at artifact execution time**.

## 5.2 What an “artifact” is (you must be precise)

An **artifact** is any executable or addressable thing in Moqui, such as:

- a screen
- a service
- an entity
- a REST endpoint

Artifacts are **identified by location and type**, not by Java class or XML file alone.

Example:

component://tutorial/screen/tutorial.xml

That string is the **artifact name**.

## 5.3 Why Moqui does NOT secure screens directly

Most frameworks do this:

@Secured("ROLE_ADMIN")

Moqui explicitly avoids that.

Why?

Because:

- screens change

- services are reused
- entities are accessed indirectly
- authorization must be configurable, not hard-coded

So Moqui introduces an **indirection layer**.

## 5.4 The three security building blocks (memorize the roles, not the XML)

There are exactly **three core concepts**:

1. **ArtifactGroup**
2. **ArtifactGroupMember**
3. **ArtifactAuthz**

They form a chain.

## 5.5 ArtifactGroup (definition)

An ArtifactGroup is a **logical grouping of artifacts** that should share the same authorization rules.

It does **nothing by itself**.

It is just a container.

Example intent:

“Everything under my Tutorial app”

## 5.6 ArtifactGroupMember (what it actually does)

This is where artifacts are **attached to the group**.

Key fields:

- artifactGroupId
- artifactName
- artifactTypeEnumId
- inheritAuthz

Example (yours, conceptually):

You added:

- tutorial root screen
- with inheritAuthz="Y"

Meaning:

All subscreens under this screen **automatically belong** to this group.

This answers an earlier confusion you had:

“How does FindTutorial get authorized without being listed?”

Because authorization **inherits down the screen tree**.

## 5.7 ArtifactAuthz (where permission actually happens)

This is the **only place** where permission is granted.

It connects:

- a **user group**
- to an **artifact group**
- with **actions allowed**

Important fields:

- userGroupId
- artifactGroupId
- authzTypeEnumId
- authzActionEnumId

This is the rule that says:

“Members of group X can do action Y on artifacts in group Z.”

Nothing else grants access.

## 5.8 UserGroup and why ALL_USERS worked

You used:

userGroupId="ALL_USERS"

This is a **special system group**.

Facts:

- Every user belongs to it
- Including anonymous (if enabled)
- Including logged-in users

This is why your app became accessible.

If you had used:

- ADMIN

- EMPLOYEE
- CUSTOM_GROUP

Only those users would pass authorization.

## 5.9 Authentication vs Authorization (your earlier confusion)

These are **separate checks**.

### Authentication

- “Who are you?”
- Handled by login, sessions, tokens

### Authorization

- “Are you allowed?”
- Handled by ArtifactAuthz

This explains why:

- removing require-authentication didn’t magically make things work
- adding ArtifactAuthz suddenly fixed access

They are independent layers.

## 5.10 What require-authentication actually does (and does NOT do)

This attribute:

require-authentication="anonymous-all"

Controls:

- whether login is required
- whether an anonymous user is injected

It does **NOT**:

- grant permissions
- override ArtifactAuthz
- bypass security rules

It only affects **authentication**, not **authorization**.

## 5.11 Execution flow (this is critical)

When a user hits:

/apps/tutorial/FindTutorial

Moqui does this:

1. Resolve screen path → artifact name
2. Check authentication requirement
3. Determine current user groups
4. Find ArtifactGroupMembers matching artifact
5. Resolve inherited groups
6. Check ArtifactAuthz rules
7. Allow or deny execution

This happens **before widgets render** and **before actions execute**.

## 5.12 What breaks without this layer (minimal failure case)

If you remove ArtifactAuthz:

- screen loads → **403 / access denied**
- even if file exists
- even if URL is correct
- even if user is logged in

If you remove ArtifactGroupMember:

- group exists
- authz exists
- but nothing is linked
- result: denied

If you remove ArtifactGroup:

- nothing to attach to
- no authorization possible

All three are mandatory.

## 5.13 Why this design exists (architectural reason)

Moqui is designed for:

- large ERP systems
- configurable permissions
- customer-specific rules
- runtime overrides

Hardcoding security would break:

- extensibility

- upgrades
- multi-tenant setups

This indirection is intentional and necessary.

## 5.14 One-sentence interview-ready definition

You can safely say:

“Moqui secures applications by authorizing user groups against artifact groups, not individual screens or services, enabling inheritance, configurability, and runtime control.”

That sentence shows understanding, not memorization.

# END OF LAYER 5 — SECURITY

You have now completed:

- Screen layer
- Entity layer
- Service layer
- Data layer
- Security layer

That is the **entire Moqui Quick Tutorial architecture**.

### Final step (mandatory)

Confirm:

“Proceed to **final consolidation and notes**”

Next, I will:

- consolidate everything into **clean, layered notes**
- remove tutorial noise
- produce **interview-grade mental models**
- give you a reusable reference you can trust

Understood. I will now do **exactly what was promised**, with no gaps, no tutorial noise, and no hand-waving.

What follows is a **final consolidation** of the Moqui Quick Tutorial into a **clean, layered, interview-grade mental model**, starting from foundations and moving upward.
This is written so that **each layer stands on its own**, and you can explain *why it exists*, *what breaks without it*, and *how Moqui uses it*.

No memorization. Only understanding.

# FINAL CONSOLIDATED NOTES — MOQUI QUICK TUTORIAL

(FOUNDATION → SCREENS → ENTITIES → SERVICES → DATA → SECURITY)

## LAYER −1: FOUNDATIONAL MENTAL MODELS (BEFORE MOQUI)

You must understand these **before** touching Moqui.

### 1. Separation of Concerns (non-negotiable)

Everything in Moqui strictly belongs to **one** of these:

- **Data** – what exists (entities, records)
- **Behavior** – what happens (services, actions)
- **Presentation** – how humans interact (screens, forms)

Moqui enforces this separation at the framework level.
Violating it causes confusion, not flexibility.

### 2. Declarative vs Imperative

Declarative means:

You describe the desired state or behavior, not the steps to achieve it.

Moqui:

- you **describe** entities
- you **describe** services
- you **describe** screens
- the framework executes

If you think in “steps” instead of “definitions”, Moqui will feel hostile.

### 3. Indirection is intentional

Moqui rarely connects A → B directly.

Instead:

- Screen → ArtifactGroup → Authorization
- Transition → Service Name → Service Resolution
- Entity Name → Definition → Final Model

This enables:

- overrides
- extensions
- upgrades
- customer-specific behavior

Indirection is not overengineering here; it is the design.

### 4. Files do not execute — facades do

XML files are **metadata**.

Execution is done by:

- Screen Facade
- Entity Facade
- Service Facade
- Security Facade

Files **register definitions**.
Facades **resolve and execute**.

## LAYER 1: SCREEN LAYER (PRESENTATION)

### What a Screen is

A **screen** is:

A declarative description of user interaction flow and UI composition.

A screen:

- does not contain business logic
- does not access the database directly
- does not decide authorization

It:

- declares transitions
- declares actions to run
- declares widgets to render

### Screen Structure (mental model)

<screen>

├── <transition>  (navigation + service calls)

├── <actions>  (data preparation)

└── <widgets>  (UI rendering)

Each part has a single responsibility.

## actions

Purpose:

Prepare data **before rendering**

Example:

```
<entity-find entity-name="tutorial.Tutorial" list="tutorialList">
    <search-form-inputs/>
</entity-find>
```

This:

- reads request parameters
- performs a query
- stores results in context

It does **not** render anything.

## widgets

Purpose:

Render UI using prepared data

Example:

```
<form-list list="tutorialList">
    <auto-fields-entity .../>
</form-list>
```

Widgets:

- never fetch data
- never modify state
- never enforce security

They **consume context only**.

## subscreens

Purpose:

Structural composition and URL mapping

Subscreens define:

- URL paths
- menu hierarchy
- inheritance of layout and authorization

### Key UI realization (your earlier confusion)

- Find dialogs do **not** submit forms
- They change query parameters
- The screen reloads automatically
- No “Apply” button is needed

This is **query-driven UI**, not form submission UI.

## LAYER 2: ENTITY LAYER (DATA MODEL)

### What an Entity is

An entity is:

A canonical definition of a persistent data structure.

It is **not**:

- a Java class
- an ORM object
- a DAO

It is a **schema contract**.

### Entity definition responsibilities

An entity defines:

- fields
- types
- primary keys
- relationships
- constraints

It does **not** define:

- behavior
- validation logic
- access rules

### Important rule (you discovered this)

If two entities share the **same entity-name**, Moqui merges them.

This means:

- fields are combined
- conflicts must be compatible
- one becomes an extension, not a replacement

The **base entity** is resolved by:

- component load order
- dependency order
- first definition wins, others extend

### What breaks without this layer

Without entities:

- services cannot operate
- screens cannot query
- data has no contract

Everything collapses.

## LAYER 3: SERVICE LAYER (BEHAVIOR)

### What a Service is

A service is:

A named unit of business behavior.

Think:

- function, not class
- behavior, not data
- callable, not visual

### Three service types (clear separation)

1. Implicit Entity-Auto

create#tutorial.Tutorial

- no file
- derived entirely from entity
- fast
- generic

Used when:

- behavior is pure CRUD
- no custom logic needed

2. Explicit Entity-Auto

<service verb="create" noun="Tutorial" type="entity-auto"/>

- file exists
- behavior still derived from entity
- allows parameter control
- overridable

Called as:

TutorialService.create#Tutorial

3. Explicit Custom (inline / script)

- behavior fully defined
- can add rules, conditions, orchestration
- still uses Entity Facade

This is where domain logic lives.

### How service resolution actually works (your big confusion)

Service name format:

[package].[ServiceFile].[verb]#[noun]

Package = directory under service/
ServiceFile = XML filename (no extension)

You fixed your bug when you realized:

- directory structure **defines the namespace**
- not the component name

This is critical knowledge.

## LAYER 4: DATA LOADING (ENTITY-FACADE-XML)

### What entity-facade-xml is

It is:

Declarative data loading through the Entity Facade.

Meaning:

- data goes through validation
- EECA rules fire
- transactions are respected

### Why not seed-data

Seed-data:

- static
- schema-bound
- always loads

Entity-facade-xml:

- environment-aware
- reloadable
- override-friendly
- participates in system behavior

This is why security data **must** use entity-facade-xml.

### Environment-specific loading

The type attribute controls **when** data loads.

Example:

- seed-initial → first startup only
- demo → dev/demo
- test → test environments

This prevents data pollution.

## LAYER 5: SECURITY LAYER (AUTHORIZATION)

### Core realization

Moqui does **not** secure screens or services directly.

It secures **artifacts via indirection**.

### The three required pieces

1. **ArtifactGroup**
   Logical grouping
2. **ArtifactGroupMember**
   Which artifacts belong (with inheritance)
3. **ArtifactAuthz**
   Who can do what

All three are mandatory.

### Why this design exists

Because:

- screens change
- services are reused
- permissions vary by customer
- upgrades must be safe

Hardcoded security would break all of this.

### Authentication vs Authorization

- Authentication: who are you
- Authorization: are you allowed

require-authentication affects only authentication.
Authorization is always checked separately.

## FINAL ONE-PARAGRAPH SUMMARY (INTERVIEW READY)

Moqui is a declarative, metadata-driven framework that separates presentation, behavior, and data into screens, services, and entities, all resolved at runtime by facades. Screens declare interaction flow, services encapsulate business behavior, entities define data contracts, entity-facade-xml manages environment-aware data loading, and security is enforced through artifact-based authorization using indirection for flexibility and override safety.

If you can explain **that paragraph**, you understand Moqui.

## What you now have

- A correct mental model
- No memorized explanations
- Clear reasoning paths
- Failure cases understood
- Debugging ability
- Interview confidence

If you want next:

- a **1-page cheat sheet**
- **mock interview questions**
- or **re-walking one layer with production examples**

Say which one.