

0. First: What You Are Actually Building (Clarity)

Before tech, be clear on:

What problem this system solves

Who are the users

What is a ticket in your system

What makes your system realistic, not toy-like

If this is unclear, architecture will be weak.

1. System Architecture (Big Picture)

You should understand:

Why frontend and backend are separate

How frontend talks to backend (HTTP APIs)

Why backend talks to database directly

Why database is NOT exposed

Key decision:

Modular monolith (not microservices)

Reason:

Easier to build

Easier to explain

Matches real-world evolution

2. Backend Architecture (ASP.NET Core Web API)

Things you must research & understand:

a) Layered Structure (WHY it exists)

Controller layer → HTTP responsibility

Service layer → business rules

Data layer → database access

You should know:

Why controllers should be thin

Why business logic must NOT be in controllers

Why repositories exist

b) Dependency Injection (VERY IMPORTANT)

Why services are injected

Lifetime types (Scoped, Transient, Singleton)

Why DB context is scoped

If you can't explain DI, interviews will expose it.

c) DTOs vs Entities

Why API should not expose DB entities

Why mapping is required

What problems DTOs solve

d) Error Handling Strategy

Why try-catch everywhere is bad

Why global exception handling exists

How APIs return consistent error responses

 3. Authentication & Authorization (CRITICAL)

You must deeply understand:

a) Authentication

What JWT is

Why JWT is stateless

How token is created

What goes inside token (claims)

b) Authorization

Role-based access

Why authorization is NOT same as authentication

How backend enforces access rules

Why frontend checks are not enough

Key concept:

Backend is the final authority.

4. User Roles & Access Control

You must define:

Who can create tickets

Who can see which tickets

Who can update status

Who can assign tickets

This affects:

API design

Database design

Security

This is core business logic, not UI.

5. Ticket Lifecycle & State Management

Before coding, clearly understand:

Possible ticket states

Valid transitions

Invalid transitions

Example:

Open → In Progress → Resolved → Closed

Closed → Reopen (maybe)

Open → Closed (maybe not)

Why this matters:

Prevents bad data

Shows domain thinking

6. Database Choice & Data Modeling

You already chose PostgreSQL — good.

Now you must understand:

Why relational DB fits this problem

What relationships exist

Why normalization matters

What should be indexed

Why transactions matter

Also:

Why DB schema design affects performance

Why changing schema later is painful

7. API Design Principles

You should research:

REST principles

Proper HTTP methods

Proper status codes

Naming conventions

Pagination patterns

Understand:

Why APIs should be predictable

Why versioning exists (even if you don't implement it)

8. Performance Considerations (Early Awareness)

Not optimization — awareness.

Know:

Why pagination is mandatory

Why filtering should be server-side

Why N+1 query problem exists

When caching makes sense (and when it doesn't)

Important:

Performance is a design concern, not a last-minute fix.

9. Validation & Data Integrity

You should think about:

Input validation (backend > frontend)

Required vs optional fields

Length limits

Status consistency

Why:

Frontend can be bypassed

APIs must protect the system

10. Frontend Architecture (React – At Right Depth)

You don't need to be UI expert, but must know:

Why frontend is stateless

How auth flow works in SPA

How protected routes work

Why frontend stores token safely

How frontend handles API errors

Frontend responsibility:

UX

State

API consumption NOT business rules.

11. Security Awareness (Must Know)

Before coding, know:

Why secrets never go in code

Why env variables exist

Why CORS is required

Why SQL injection is a thing (even with ORM)

Why authorization checks must be server-side

12. Deployment & Environment Thinking

Even before coding, know:

Dev vs Prod environment difference

Why connection strings differ

Why logging matters in production

Why “works on my machine” is useless

13. Documentation Mindset

You should think:

Can someone else understand my project?

Can interviewer read my README and get clarity in 2 minutes?

Can I explain architecture verbally?

Good projects are explainable, not just working.

14. Interview Mapping (VERY IMPORTANT)

For every design decision, ask:

Why did I choose this?

What alternative existed?

Why did I reject it?

Interviewers love:

“I chose X because Y, and avoided Z due to trade-offs.”



15. What NOT to Overthink (Yet)

Don't deep dive now into:

Microservices

Kubernetes

Event-driven systems

Heavy DevOps

Extreme optimization

Those come after fundamentals.



Final Mental Model

Before coding, you should be able to answer:

What problem am I solving?

Who are the users?

How does data flow?

Where is business logic enforced?

How is security guaranteed?

How would I explain this to another developer?

If you can answer these, you're ready to start coding.

Next (ONLY when you're ready)

We can:

Review your mental design

Validate your assumptions

Stress-test your architecture decisions

No code. No rush. Just say when 👍

all points are great and I would love to research all before starting the project. can you revised again and see if you left any points others than these 15-16 points.

Love this approach. You're thinking like someone who wants to build once, explain confidently, and switch jobs — not like someone rushing to code.

I'll re-audit your checklist and add only the points that were missing or under-represented, so you end up with a complete, no-blind-spots mental model.

Below is the REVISED + EXPANDED “before coding” checklist. I'll mark NEW / OFTEN MISSED items clearly.

✅ COMPLETE PRE-CODING AWARENESS CHECKLIST

(Everything you should know before writing code)

You already covered ~80%. This fills the remaining 20%.

① Problem & Domain Understanding

(You had this – keeping for completeness)

What exact problem is solved?

Who are the users?

What is in scope vs out of scope?

What assumptions are you making?

NEW nuance:

What real-world constraints exist? (time, approvals, ownership)

② System Boundaries (NEW – Very Important)

You must define:

What your system does

What it explicitly does NOT do

Example:

✗ Email sending (mocked)

✗ External integrations

✗ Mobile app

This prevents scope creep and interview traps.

③ Architecture Style (You had this)

Why modular monolith

Why not microservices now

How system could evolve later

You should be able to say:

“This architecture supports future extraction of services.”

④ Backend Layering & Responsibility Split

(You had this, but one thing is often missed)

NEW sub-point: Boundary enforcement

Controllers must not contain logic

Services must not know HTTP

Repositories must not know business rules

Interviewers test this indirectly.

⑤ Domain Modeling (NEW – Critical)

Before DB or API, you must understand:

What is an entity

What is a value object

What is behavior vs data

Example:

Ticket status is behavior, not just a column.

This is where many projects feel “CRUD-ish”.

⑥ State Transitions & Business Rules

(You had lifecycle, but this is deeper)

You must define:

Allowed transitions

Who can trigger them

Under what conditions

Example:

Only Admin can close escalated tickets

Closed tickets cannot be edited

This is real backend thinking.

⑦ Data Ownership & Multi-Tenancy Thinking (NEW)

Even if single-tenant, think about:

Who owns a ticket?

Can two users ever see same data?

How do you enforce this in queries?

This affects:

Security

Query design

Authorization logic

⑧ Database Modeling & Integrity

(You had DB choice; this is conceptual depth)

You should know:

Why foreign keys matter

Why cascading deletes can be dangerous

Why audit fields exist

Why soft delete vs hard delete

Interviewers love asking:

“What happens if a user is deleted?”

9 Transactions & Consistency (NEW)

Understand:

When multiple operations must succeed together

What happens on partial failure

Why DB transactions exist

Even if you don't implement many, you must know when they're needed.

10 API Contract Design

(You had REST, but missing one thing)

NEW sub-points

Request/response consistency

Error response shape

Validation error strategy

APIs are contracts, not just endpoints.

11 Authentication vs Authorization

(You had this, but add this nuance)

You must clearly understand:

What frontend checks are allowed to do

What ONLY backend can enforce

Why token ≠ permission

This is a common interview separator.

12 Security Threat Awareness (Expanded)

You covered basics. Add awareness of:

Broken object level authorization (BOLA)

Over-posting attacks

ID guessing attacks

You don't need to fix all — but you must recognize them.

13 Validation Strategy (Expanded)

Know:

Where validation belongs

What should be validated at API boundary

What should be validated at domain level

Why:

Validation is part of business correctness, not UI.

14 Performance Awareness (Expanded)

You covered pagination/caching.

Add:

Query efficiency awareness

Indexing basics

Avoiding over-fetching

Not optimization — awareness.

15 Logging & Observability (NEW – Often Missed)

Before coding, know:

Why logs exist

Difference between logs vs errors

What should be logged

What should NOT be logged (PII)

Even basic logging shows maturity.

16 Environment & Configuration Management

(You had deployment; add this nuance)

Understand:

Why dev/test/prod configs differ

Why secrets are injected

Why config \neq code

This is very important for hosted apps.

17 Failure Scenarios (NEW – Senior Thinking)

Ask yourself:

What if DB is down?

What if token is expired?

What if request is malformed?

What if user retries?

You don't need complex handling — just awareness.

18 Testing Strategy (Conceptual Only)

Not how to write tests — just:

What should be unit-tested

What should be integration-tested

What is okay to skip

This shows engineering judgment.

19 Documentation & Explainability

(You had this, reinforcing)

Ask:

Can I explain this in 5 minutes?

Can I explain trade-offs?

Can I defend my decisions?

If not, redesign mentally.

Resume & Interview Mapping (Expanded)

Before coding, decide:

What this project proves about me

Which skills it highlights

Which questions it prepares me for

This ensures resume alignment.

 Explicitly Out of Scope (NEW but Important)

You should consciously exclude:

Premature microservices

Over-engineered patterns

Fancy UI work

Overuse of buzzwords

Knowing what not to do is senior behavior.

 Final Meta-Check (Golden Rule)

Before coding, you should be able to answer:

“If an interviewer asks why for any decision, do I have a clear reason?”

If yes → you're ready.