mvc

Here is the **complete interview question bank for MVC (Model–View–Controller)** — relevant for **.NET + Spring Boot + Angular architecture understanding**.

🔥 MVC — Full Interview Question List

📌 1. MVC Basics

1) What is MVC architecture?

■ Explanation (simple, conceptual)

MVC (Model–View–Controller) is a design pattern used to separate an application into **three layers** — Model, View, and Controller.

It divides UI, business logic, and input handling to ensure clean and structured development.

■ When/Why is it used

- To separate UI code from processing logic
- To improve maintainability and scalability
- To allow developers to work on UI and backend independently

■ Example / Diagram

```
User → Controller → Model → Database
          ↓
        View (UI)
```

■ Short summary

> MVC is a design pattern that splits UI, logic, and data handling into three separate components.

---

2) Full form of MVC & responsibility of each layer

■ Explanation

M → Model

Handles application data, business objects, and DB communication.

V → View

UI layer that presents data to the user (HTML, Razor, Angular template).

C → Controller

Handles incoming requests, processes data via model, and returns View/Response.

■ When/Why used

- To implement separation of concerns
- To keep UI, logic, and data independent
- To enable parallel development between teams

■ Example

```
Controller receives request
↓ calls Model for data
Model returns data
↓ Controller passes data to View
View displays output to user
```

■ Summary

> Model = Data, View = UI, Controller = Request handler connecting UI and data.

---

3) Why MVC is used in web applications?

■ Explanation (concept)

MVC is used because it reduces complexity by clearly separating presentation, logic, and data. This makes the application more maintainable, scalable, and testable.

■ When/Why used

- Clean separation of UI and business logic

- Easier debugging and enhancement
- View can change without affecting backend
- Supports test-driven development (TDD)

■ **Example / Use case**

```
UI redesign? → Only update Views
DB rule changes? → Only update Model/Service
```

■ **Summary**

> MVC is used in web apps to achieve separation of concerns, flexibility, and easier maintenance.

---

4) How MVC separates concerns?

■ **Explanation**

MVC separates concerns by assigning **different responsibilities** to its components:

- Controller handles **requests and routing**
- Model handles **business logic + data management**
- View handles **UI rendering only**

■ **When/Why used**

- Allows independent modification of UI and backend logic
- Reduces code duplication
- Makes testing easier

■ **Example**

Without separation ❌

Business logic + HTML mixed together

With MVC ✓

```
| Controller → Input Handling
| Model → Business/Data Logic
```

```
| View → Only UI Display
```

■ **Summary**

> MVC separates responsibilities so UI, logic, and data can evolve independently.

---

5) What happens when a request comes in MVC?

■ **Explanation**

When a request arrives, it's routed to a controller action, which then interacts with the model and finally returns a View or JSON as response.

■ **Request Flow (Step-by-step)**

```
1. User enters URL or clicks button
2. Routing sends request to a Controller & Action Method
3. Controller interacts with Model (business/data)
4. Model fetches data from DB and returns result
5. Controller passes result to View
6. View renders UI and sends final HTML/JSON to user
```

■ **Example**

```
// URL → /Expense/Add
Routing → ExpenseController.Add()
Controller → calls Model to save
Model → saves DB entry
View → returns response "Expense Added Successfully"
```

■ **Summary**

> Request flows from Router → Controller → Model → View → Browser as HTML/JSON.

📌 2. Components

1) What is Model? What does it contain?

■ **Explanation (simple, conceptual)**

Model represents the **data and business entities** of the application. It interacts with the database and holds validation + domain rules. It may be an ORM entity, DTO, or business model depending on architecture.

■ **When/Why used**

- To encapsulate data structure and business rules
- To map database tables into objects
- To separate data/logic from UI and Controller

■ **Example**

```
public class Expense
{
    public int Id { get; set; }
    public string Category { get; set; }
    public decimal Amount { get; set; }
    public DateTime Date { get; set; }
}
```

■ **Summary line**

> Model contains application data, validation rules, and DB mappings — forming the foundation for business logic.

---

2) What is Controller? Key responsibilities.

■ **Explanation (simple, conceptual)**

Controller is the **decision-maker** — it handles incoming requests, interacts with Model/Services, and passes response to the View.

■ **Key responsibilities**

✔️ Receive HTTP requests

✔️ Call business logic (Service/Model)

✔️ Return View or JSON response

✔️ Validate input (light validation)

■ **Example**

```
public IActionResult GetExpenses()
{
    var expenses = _service.GetAll();
    return View(expenses);
}
```

■ **Summary line**

> Controller handles HTTP requests, calls business layer, and returns response to View.

---

3) What is View? What should not be written inside View?

■ **Explanation (simple, conceptual)**

View is the **presentation/UI layer**, responsible for rendering data as HTML/Angular UI for the user.

■ **View must NOT contain**

❌ Business logic

❌ Complex calculations

❌ DB calls or Repository logic

✔ It should only format and display data.

■ **Example**

```
@model List<ExpenseDto>
<table>
 @foreach(var item in Model)
 {
   <tr><td>@item.Category</td><td>@item.Amount</td></tr>
 }
</table>
```

■ **Summary line**

> View is UI only — it must display data, not compute business logic or call DB.

---

4) Why business logic should not be in Controller?

■ Explanation (simple, conceptual)

Controller should only coordinate; if business logic is done here, code becomes heavy, repetitive, untestable, and violates separation of concerns.

■ When/Why used

- Logic belongs to Service Layer
- Makes controller lighter and reusable
- Service logic can be reused across multiple APIs/UI screens

■ Example

❌ Bad:

```
if(amount > 10000) discount = 15;
```

✔ Good:

```
discount = _expenseService.CalculateDiscount(amount);
```

■ Summary line

> Business logic stays in service layer so controllers remain clean, testable, and maintainable.

---

5) What is ViewModel?

■ Explanation (simple, conceptual)

ViewModel is a **UI-focused model** that contains exactly what the view needs — sometimes combining multiple Models/DTOs into one.

■ When/Why used

- To prepare UI data in one object
- To avoid sending raw database Entities
- Used when View requires combined/mapped values

■ Example

```
public class DashboardViewModel
{
```

```
    public decimal MonthlyTotal { get; set; }
    public List<ExpenseDto> RecentExpenses { get; set; }
    public List<string> Categories { get; set; }
}
```

■ **Summary line**

> ViewModel is a UI-specific data model combining only necessary fields required for rendering a page.

📌 3. Request Life Cycle

1) Explain the complete flow from request → controller → view

■ **Explanation (simple, conceptual)**

When a user sends a request (URL hit), MVC routes it to the correct controller and action method. The controller calls model/service to process data, then passes data to the view, which finally renders UI to the browser.

■ **When/Why used**

- To cleanly divide request handling, business logic, and UI
- Makes debugging and development structured
- Each layer performs only its own responsibility

■ **Example / Flow Diagram**

```
User Request (URL)
      ↓
Routing → Chooses Controller + Action
      ↓
Controller → Calls Service/Model
      ↓
Model → Fetches/Processes Data
      ↓
Controller → Sends Model Data to View
      ↓
View → Generates HTML/JSON
```

```
    ↓
Response Sent to Browser
```

■ **Short summary line**

> Request passes through Routing → Controller → Model → back to View, which renders final UI to browser.

---

2) How routing works in MVC?

■ **Explanation**

Routing maps URL patterns to corresponding **Controllers & Action Methods**.

MVC reads routing rules and decides which controller/action should process the request.

■ **When/Why used**

- To control URL structure
- To route requests to correct actions
- To create meaningful and SEO-friendly URLs

■ **Example**

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

URL → `/Expense/List` maps to:

```
Controller = ExpenseController
Action     = List
```

■ **Short summary line**

> Routing is responsible for mapping URL to correct controller and action.

---

3) Attribute Routing vs Conventional Routing

Feature	Conventional Routing	Attribute Routing
Defined Where?	In startup or RouteConfig	On controller/actions
URL controlled by?	Central rule	Individual attributes
Flexibility	Less flexible	Very flexible
Example Use	Simple apps	APIs or RESTful services

■ Example Code

Conventional Routing

```
app.MapControllerRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

Attribute Routing

```
[Route("api/expenses")]
public class ExpenseController : Controller
{
    [HttpGet("all")]
    public IActionResult GetAll() { ... }
}
```

■ Short summary line

> Conventional routing is centralized, while Attribute routing is flexible and defined directly on controller/action methods.

---

4) What is Action Method?

■ Explanation

Action Method is a function inside a Controller that handles an incoming HTTP request and returns a response (View/JSON/Redirect).

■ When/Why used

- Triggered each time URL hits controller
- Processes input, performs logic via services
- Returns view or API result

■ Example

```
public IActionResult Dashboard()
{
    var result = _service.GetMonthlySummary();
    return View(result);
}
```

■ **Short summary line**

> Action Method is a controller function that processes a request and returns output to the user.

---

5) ActionResult vs ViewResult

■ **Explanation (simple, conceptual)**

ActionResult is a **base return type**, capable of returning different outputs (View, JSON, Redirect, File, StatusCode etc.).

ViewResult returns only a **View (.cshtml UI page)**.

■ **When/Why used**

Type	When to use
ActionResult	When action could return multiple response types
ViewResult	When you always return a View (UI HTML only)

■ **Example**

```
public ActionResult GetExpense()       // Can return many formats
{
    if (id == 0) return NotFound();    // JSON
    return View(expense);              // ViewResult output
}

public ViewResult Show()               // Always returns View
{
    return View();
}
```

■ **Short summary line**

> ViewResult returns only a View UI, while ActionResult can return View, JSON, Redirect or HTTP status codes.

📌 4. Model Binding & Validation

1) What is Model Binding?

■ Explanation (simple, conceptual)

Model Binding automatically maps incoming request data (form fields, query params, JSON body) into C# objects.

It removes the need to manually extract values from Request objects.

■ When/Why used

- To directly receive form/API data inside action method
- Reduces boilerplate code
- Converts HTTP input → Strongly typed object automatically

■ Example

```
public IActionResult CreateExpense(ExpenseDto model)  // Model binding
{
   // model.Amount , model.Category populated automatically
}
```

■ Summary line

> Model Binding converts incoming request data into C# objects automatically.

---

2) How Model Validation works?

■ Explanation

Model Validation checks values of model properties against validation attributes (like Required, Range) before controller executes business logic.

■ When/Why used

- To prevent invalid data from reaching DB/service
- To enforce input correctness centrally

- Used for both MVC and Web API requests

■ **Example**

```
public IActionResult Create([FromForm] ExpenseDto dto)
{
   if(!ModelState.IsValid)   // Validation check
      return BadRequest(ModelState);
}
```

■ **Summary line**

> Model Validation checks incoming model values against validation rules before processing request.

---

3) Data Annotations — Required, MaxLength, Email

■ **Explanation (simple)**

Data Annotations are attributes used to validate model properties.

■ **Examples with code**

```
public class UserDto
{
   [Required(ErrorMessage="Name is mandatory")]
   public string Name { get; set; }

   [MaxLength(50)]
   public string Address { get; set; }

   [EmailAddress]
   public string Email { get; set; }
}
```

■ **When/Why used**

- Validate inputs at model-level
- Standard validation with minimal code
- Works for both Forms and APIs

■ Summary line

> Required prevents nulls, MaxLength restricts length, Email ensures proper email format.

---

4) Difference between TempData, ViewBag, ViewData

Feature	ViewData	ViewBag	TempData
Type	Dictionary	Dynamic wrapper over ViewData	Dictionary (persists longer)
Lifetime	Only current request	Only current request	Persists until next request
Usage	Pass value to View	Shortcut for ViewData	Redirect scenarios
Syntax	`ViewData["msg"]`	`ViewBag.msg`	`TempData["msg"]`

■ Example

```
ViewData["Title"] = "Dashboard";
ViewBag.Name = "Mohini";
TempData["Success"] = "Expense Added Successfully"; // survives redirec
t
```

■ Summary line

> ViewData & ViewBag work for same request; TempData persists across redirects — used for messages/alerts.

---

5) What is strongly-typed view?

■ Explanation

A strongly-typed view is bound with a specific Model type using `@model`, enabling IntelliSense and type checking directly inside the view.

■ When/Why used

- For model-driven forms/data rendering
- Ensures compile-time error checking
- Helps with model iteration & binding cleanly

■ **Example (View)**

```
@model List<ExpenseDto>

@foreach(var item in Model)
{
   <p>@item.Category - @item.Amount</p>
}
```

■ **Summary line**

> Strongly-typed views bind a specific model type to the view using @model, enabling typed access with IntelliSense.

📌 5. Views + UI

1) What is Razor View Engine?

■ **Explanation (simple, conceptual)**

Razor View Engine is the templating engine used in ASP.NET MVC to generate dynamic HTML using C# code mixed with markup inside `.cshtml` files.

It allows switching between C# and HTML using `@` syntax.

■ **When/Why used**

- To generate UI dynamically using backend values
- Cleaner syntax than ASPX WebForms
- Lightweight, fast, server-side rendering

■ **Example**

```
<h2>Welcome @Model.UserName</h2>
```

■ **Summary line**

> Razor is a lightweight view engine that generates dynamic HTML using @ syntax inside Views.

---

2) @model vs @using vs @inject in Views

Keyword	Meaning	Used For
@model	Defines model type for view	Strongly-typed view
@using	Import namespace	Access classes without full path
@inject	Inject service into view	DI inside Razor view

■ **Example**

```
@model ExpenseDto          // strongly typed model
@using MyApp.ViewModels    // namespace import
@inject IExpenseService service // injecting a service
```

■ **Summary line**

> @model binds model, @using imports namespace, @inject injects DI services inside Razor views.

---

3) Partial Views — what are they used for?

■ **Explanation**

Partial Views are reusable view components used to avoid duplicating UI code.

Common UI pieces like headers, forms, table rows can be written once and reused.

■ **When/Why used**

- Shared UI elements (Navbar, Footer, Cards)
- Reusing forms across multiple pages
- Better modularity + maintainability

■ **Example**

```
@Html.Partial("_ExpenseFormPartial", Model)
```

■ **Summary line**

> Partial Views are reusable Razor UI fragments to avoid code duplication across pages.

---

4) Layout vs View vs Partial View

Component	Purpose
Layout	Master template (header/sidebar/footer)
View	Full page for UI screen
Partial View	Small reusable UI block

■ Example Structure

```
_Layout.cshtml    → outer shell (applies to all pages)
Index.cshtml      → full page (view)
_ExpenseForm.cshtml → partial view reused in multiple pages
```

■ Summary line

> Layout = Master page, View = Full UI page, Partial View = Reusable UI block inside view.

---

5) How to pass data from Controller → View?

■ Explanation

Data is passed using **Model**, **ViewBag**, **ViewData**, or **TempData**.

Strongly-typed Model is the most recommended.

■ Example

```
public IActionResult Dashboard()
{
   var data = _service.GetSummary();
   return View(data); // Passing Model
}
```

In View:

```
@model DashboardViewModel
<p>Total: @Model.MonthlyTotal</p>
```

■ Summary line

> Data is sent from Controller to View using Model (best), ViewBag, ViewData or TempData.

6) How to pass data View → Controller (form)?

■ Explanation

Data is submitted from View using **form POST**, where Model Binding maps form fields to model parameters automatically.

■ When/Why used

- To save user inputs (login, expense entry, form submission)

■ Example

View (.cshtml)

```
@model ExpenseDto
<form asp-action="Create" method="post">
    <input asp-for="Category" />
    <input asp-for="Amount" />
    <button type="submit">Save</button>
</form>
```

Controller

```
[HttpPost]
public IActionResult Create(ExpenseDto model)
{
    // model.Category and model.Amount auto-filled
}
```

■ Summary line

> View sends data to Controller using form POST, and Model Binding maps inputs to action parameters.

📌 6. Routing

1) What is RouteConfig?

■ Explanation (simple, conceptual)

`RouteConfig` is the configuration file (in older ASP.NET MVC) used to define routing rules for mapping URLs to controller/action methods.

In .NET Core the equivalent is `app.MapControllerRoute`.

■ When/Why used

- To define URL structure of the application
- To choose which controller/action handles request
- To create SEO-friendly & readable URLs

■ Example

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParam
eter.Optional }
        );
    }
}
```

■ One-line summary

> RouteConfig defines URL routing rules to map requests to the correct controller and action.

---

2) Route Parameters vs Query Parameters

Type	Format in URL	Purpose
Route Parameter	`/expense/details/5`	Required part of URL, used for entity identification
Query Parameter	`/expense/list?month=Jan&page=2`	Optional filters/search values

■ Example

```
// Route parameter
/Expense/Details/10

// Query parameter
/Expense/List?category=Food&page=2
```

■ **One-line summary**

> Route params identify a resource, Query params are used for filtering and search.

---

3) Example of URL Routing with parameters

■ **Explanation**

Routing lets you bind URL segments to action parameters.

■ **Example**

Conventional Routing

```
app.MapControllerRoute(
    name: "ExpenseRoute",
    pattern: "expense/details/{id}",
    defaults: new { controller = "Expense", action = "Details" }
);
```

URL:

```
/expense/details/5
```

Goes to → `ExpenseController.Details(int id)`

Attribute Routing

```
[Route("expense/details/{id}")]
public IActionResult Details(int id)
{
    var exp = _service.Get(id);
```

```
    return View(exp);
}
```

■ **One-line summary**

> /expense/details/{id} is a routed URL mapping to Details(id) action using conventional or attribute routing.

---

4) What is default controller/action?

■ **Explanation**

Default controller & action are used when the URL does not explicitly specify a controller or action.

It improves navigation and allows root URL `/` to open a default page.

■ **When/Why used**

- To make Home page accessible without typing `/Home/Index`
- To avoid 404 errors on base URL
- Standard entry point of application

■ **Example (Conventional Routing)**

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

If user enters only:

```
http://localhost:5000/
```

It automatically routes to:

```
Controller → HomeController
Action     → Index()
```

■ **One-line summary**

> Default controller/action handles base URL routing when no controller/action is specified.

📌 7. Filters (Important for real applications)

1) What are Filters?

■ Explanation (simple, conceptual)

Filters are components in MVC that allow you to run code **before or after controller execution**, enabling reusable logic like logging, authorization, caching, and exception handling across multiple actions without rewriting code.

■ When/Why is it used

- To apply cross-cutting concerns globally
- To reduce duplicate code inside controllers
- To inject pre-processing or post-processing logic

■ Example

```
[Authorize]  // filter
public IActionResult Dashboard() { ... }
```

■ One-line summary

> Filters run code before/after controller actions and are used for cross-cutting concerns like auth, logging, and exception control.

---

2) Types: Authorization, Action, Resource, Exception

■ Explanation

MVC provides multiple filter types, each executed at different pipeline stages.

Filter Type	Purpose	Executes When
Authorization Filter	Authentication + role access	Before controller/action
Action Filter	Pre & Post logic around actions	Before/After action method
Resource Filter	Request short-circuiting & caching	Before rest of pipeline
Exception Filter	Handle unhandled errors centrally	When exception occurs

■ **Example**

```
[Authorize]          // Auth Filter
[ServiceFilter(typeof(LogFilter))]   // Action Filter
```

■ **One-line summary**

> Authorization secures access, Action wraps action execution, Resource handles request caching, ExceptionFilter manages errors centrally.

---

3) Where do we use ExceptionFilter?

■ **Explanation (simple, conceptual)**

ExceptionFilter is used to catch and handle unhandled exceptions raised inside controllers or action methods. Instead of try-catch everywhere, we apply a global ExceptionFilter to return a clean, uniform response.

■ **When/Why used**

- To centralize exception handling
- To return structured error responses
- To log errors in one place
- To avoid repeated try/catch blocks inside controllers

■ **Example**

```
public class GlobalExceptionFilter : IExceptionFilter
{
    public void OnException(ExceptionContext context)
    {
        context.Result = new JsonResult("Error occurred");
        context.HttpContext.Response.StatusCode = 500;
    }
}
```

Register globally:

```
services.AddControllersWithViews(options =>
{
```

```
    options.Filters.Add<GlobalExceptionFilter>();
});
```

■ **One-line summary**

> ExceptionFilter handles errors globally so you don’t write try-catch in every controller.

---

4) Custom filter creation

■ **Explanation**

We create custom filters to add reusable logic like logging, performance checking, header validation, auditing etc.

■ **When/Why used**

- When built-in filters (Authorize/Exception) aren’t enough
- To reuse logic across multiple controllers/actions
- For cross-cutting tasks like tracking execution time, request logging

■ **Example**

```
public class LogActionFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        Console.WriteLine("Action started at " + DateTime.Now);
    }

    public override void OnActionExecuted(ActionExecutedContext context)
    {
        Console.WriteLine("Action finished at " + DateTime.Now);
    }
}
```

Use:

```
[LogActionFilter]
public IActionResult GetExpenses() => View();
```

■ One-line summary

> Custom filters allow reusable cross-cutting logic like logging/performance tracking without repeating code in controllers.

📌 8. Security

1) Form Authentication vs Token Authentication

■ Explanation (simple, conceptual)

Both are authentication mechanisms, but differ in how login state is maintained:

Form Authentication	Token Authentication (JWT, Bearer Token)
Stores user login using **Cookies**	Stores user identity using **Token in header**
Session lives at server	Stateless — no server memory required
Suitable for MVC websites	Best for APIs, mobile, SPA (Angular)
Cookie sent with each request	Token sent via `Authorization: Bearer <token>`

■ When/Why used

- **Form Auth** → Traditional server-rendered MVC apps
- **Token Auth** → REST APIs, mobile apps, Angular/React frontends, microservices

■ Example

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

■ One-line summary

> Form Authentication uses cookies + server sessions, Token Authentication is stateless using JWT — ideal for APIs & Angular apps.

---

2) AntiForgeryToken — why used?

■ Explanation (simple, conceptual)

`@AntiForgeryToken` protects forms from **CSRF attacks** where a malicious website tricks a logged-in user to unknowingly submit requests.

It generates a unique hidden token + cookie pair, and MVC validates both. If mismatch → request blocked.

■ When/Why used

- To ensure form submissions come from legit UI only
- Prevents fake/malicious form POST execution
- Mandatory for sensitive operations like payments/edit/delete

■ Example

View

```
<form asp-action="TransferAmount" method="post">
    @Html.AntiForgeryToken()
    <input type="text" name="amount"/>
    <button type="submit">Send</button>
</form>
```

Controller

```
[ValidateAntiForgeryToken]
public IActionResult TransferAmount(decimal amount) { ... }
```

■ One-line summary

> AntiForgeryToken protects against CSRF attacks by ensuring form submissions originate only from the valid application.

---

3) How role-based access works in MVC?

■ Explanation

Role-based access ensures specific pages or actions are only accessed by users with specific roles like `Admin`, `User`, `Manager`.

MVC enforces role restrictions using the `[Authorize(Roles="...")]` attribute.

■ When/Why used

- To secure sensitive pages (Admin Panel, Delete operations)
- To allow different permissions for different user roles

- To protect business-critical APIs

■ Example

```
[Authorize(Roles = "Admin")]
public IActionResult DeleteUser(int id)
{
    // Only Admin can access
}
```

Multiple roles:

```
[Authorize(Roles = "Admin,Manager")]
public IActionResult ApproveTransaction() { ... }
```

■ One-line summary

> Role-based access in MVC is enforced using [Authorize(Roles="Admin")] to restrict actions based on user roles.

📌 9. Performance and Optimization

1) View Caching vs Output Caching

■ Explanation (simple, conceptual)

Caching reduces reprocessing time by storing rendered results.

Two types most used in MVC:

Type	What is cached?	When used?
View Caching	Only View/UI output (HTML markup)	When data is static but page layout costly to render
Output Caching	Final response of controller action (View + Model data)	When full response can be reused without recalculation

■ When/Why used

- To reduce server computation & API calls
- Faster page loading with fewer DB hits

■ Example (Output Cache)

```
[OutputCache(Duration = 60)] // caches for 60 sec
public ActionResult Dashboard() { ... }
```

■ **Summary line**

> View caching caches UI rendering, Output caching stores full response — reducing repeat execution time.

---

2) How to reduce View rendering time?

■ **Explanation**

Rendering time reduces by minimizing work performed inside the view.

Views should be lightweight, using preprocessed ViewModels and avoiding heavy loops/logic.

■ **Effective Ways**

✔️ Use ViewModel → avoid extra formatting calculations in View

✔️ Use caching for repeated UI blocks

✔️ Use Partial Views only when necessary

✔️ Reduce server-side loops and transform data in Controller/Service

✔️ Enable bundling & minification for scripts/CSS

✔️ Preload essential data → lazy load heavy sections

■ **Example (convert heavy logic to controller)**

❌ Bad inside View:

```
@foreach(var e in Model.Where(x⇒x.Amount>50000))
```

✔️ Good inside Controller:

```
var filtered = model.Where(x⇒x.Amount>50000).ToList();
return View(filtered);
```

■ **Summary line**

> Prepare data in Controller/Service and keep view light, using caching and bundling for fast rendering.

3) How bundling/minification work in MVC?

■ **Explanation**

Bundling combines multiple CSS/JS files into one file.

Minification removes spaces/comments from code → reducing file size for faster load.

■ **When/Why used**

- Decreases number of HTTP requests
- Reduces download size → improves UI load time
- Great for large MVC views with many scripts/styles

■ **Example**

```
bundles.Add(new ScriptBundle("~/bundle/js")
    .Include("~/Scripts/jquery.js", "~/Scripts/bootstrap.js"));

bundles.Add(new StyleBundle("~/bundle/css")
    .Include("~/Content/site.css", "~/Content/bootstrap.css"));
```

In view:

```
@Scripts.Render("~/bundle/js")
@Styles.Render("~/bundle/css")
```

■ **Summary line**

> Bundling merges multiple JS/CSS files, Minification compresses them — reducing load time and network calls.

4) How to handle large UI pages efficiently?

■ **Explanation**

Large pages must be optimized so UI loads fast without blocking user activity.

Techniques like pagination, lazy loading, AJAX calls, and partial rendering improve scalability.

■ **When/Why used**

- When UI displays big tables, dashboard widgets, reports
- To avoid loading entire UI content at once
- To improve user experience and reduce server load

■ **Real Techniques**

✔ **Pagination + Server-side filtering** for large tables

✔ **Lazy loading / infinite scrolling**

✔ **Load heavy components via AJAX instead of initial load**

✔ **Use Partial Views** to load UI blocks separately

✔ Avoid returning 1000+ records → paginate instead

✔ Cache UI-friendly static content

■ **Example**

```
public ActionResult List(int page=1)
{
  var data = _repo.GetPaged(page, pageSize:50); // server-side pagination
  return View(data);
}
```

■ **Summary line**

> Use pagination, lazy loading, partial rendering & AJAX calls to handle heavy UI screens efficiently.

📌 10. MVC vs Others

1) MVC vs MVVM vs MVP

■ **Explanation (simple, conceptual)**

All three are UI architectural patterns but differ in how UI & logic communicate.

Pattern	Components	How they communicate	Best used for
MVC	Model–View–Controller	View ↔ Controller ↔ Model	Web apps (ASP.NET MVC)
MVVM	Model–View–ViewModel	View binds directly to ViewModel	Angular, WPF, React apps

Pattern	Components	How they communicate	Best used for
MVP	Model–View–Presenter	Presenter updates View	Windows forms, legacy UI

■ Example (simple)

MVC → Controller updates View

MVVM → View auto updates via bindings

MVP → Presenter controls View like mediator

■ One-line summary

> MVC uses Controller, MVVM uses ViewModel with bindings, MVP uses Presenter to UI output.

---

2) MVC vs Web API

Point	MVC	Web API
Output Type	Views (HTML UI)	JSON/XML Response
Used For	Web page rendering	REST API for client apps
Communication	Browser requests	Mobile/Angular/React clients
View Engine	Razor	No View Engine

■ When used in real-world

- MVC → when server generates pages (`.cshtml`)
- Web API → when Angular/React needs JSON data

■ Example

```
// MVC
return View(model);

// Web API
return Ok(data);
```

■ One-line summary

> MVC returns Views, Web API returns JSON — ideal for SPAs and microservices.

---

3) Why Angular/React used with MVC backend?

■ Explanation

Angular/React provide a dynamic, fast client-side UI, while MVC/Web API backend gives secure data, business logic, and DB operations.

■ When/Why used

✔ Better UI experience (SPA)

✔ No page reload — smoother interactions

✔ Backend and frontend can scale independently

✔ Reusable APIs for mobile + web apps

■ Architecture flow

```
Angular/React UI → calls → MVC Web API → Service → Repository → DB
```

■ One-line summary

> Angular/React builds rich UI at client side while MVC/Web API backend processes data & business logic.

---

4) Difference between MVC .NET Framework vs MVC in .NET Core

Feature	.NET Framework MVC	.NET Core MVC
Platform	Windows only	Cross-platform (Win/Linux/Mac) ✔
Performance	Slower	Faster, lightweight runtime
Hosting	IIS only	IIS + Kestrel + Docker/K8s
Configuration	web.config	appsettings.json
DI Support	Limited/Manual	Built-in DI support ✔
Modern API	Older tech	Razor Pages, Minimal APIs

■ When to choose which

- .NET Framework → only for legacy enterprise Windows apps
- .NET Core → modern scalable APIs & cross-platform deployments

■ One-liner summary

> .NET Core MVC is faster, cross-platform, cloud-ready with built-in DI — better for modern web apps.

📌 11. Real Time Scenario Questions

1) Explain MVC architecture of your project.

■ Explanation (simple, conceptual)

In my Expense Tracker project, I followed a **clean MVC architecture**:

- **Model** → expense entity, DTOs, and ViewModels
- **View** → Razor views showing expense list, forms, dashboard
- **Controller** → handled requests, called service layer, returned View or JSON

I also had an internal **service + repository** structure inside the MVC project to keep business logic and DB access separate.

■ When/Why is it used

I used MVC to:

- Separate **UI (Razor)** from **business logic (Service)** and **data access (Repository/EF)**
- Make it easier to maintain features like filters, validation, and reporting
- Allow the same backend logic to be reused later via API if needed

■ Example / Flow in my project

For "Add Expense" scenario:

1. **User** fills the Add Expense form in View (`Create.cshtml`).
2. Form posts to `ExpenseController.Create(ExpenseViewModel model)`.
3. Controller checks `ModelState.IsValid` and calls `_expenseService.AddExpense(model)`.
4. **Service Layer** applies rules (amount > 0, date not in future), maps ViewModel → Entity, calls Repository.
5. **Repository (DAL)** uses EF/SQL to insert into `Expenses` table in SQL Server.
6. On success, controller redirects back to **Index View** with success TempData.

```
public class ExpenseController : Controller
{
    private readonly IExpenseService _expenseService;

    public ExpenseController(IExpenseService expenseService)
    {
        _expenseService = expenseService;
    }

    [HttpPost]
    public IActionResult Create(ExpenseViewModel model)
    {
        if (!ModelState.IsValid)
            return View(model);   // return same view with validation errors

        _expenseService.AddExpense(model);
        TempData["Success"] = "Expense added successfully";
        return RedirectToAction("Index");
    }
}
```

■ Short summary line

> In my project, MVC architecture was: View → Controller → Service → Repository → DB, with each layer having clear responsibilities for UI, logic, and data.

---

2) Where did you use Partial Views?

■ Explanation (simple, conceptual)

I used Partial Views for **reusable UI sections** that appeared on multiple pages, so I didn't repeat HTML and logic everywhere. Typical examples: **expense form**, **summary cards**, and common **layout blocks**.

■ When/Why is it used

- To **reuse the same form** for Create and Edit expense
- To keep views clean and avoid copy-paste

- To update a UI block in one place and reflect changes everywhere

■ Example / Usage in my project

1. **Expense Form Partial View**
    - File: `_ExpenseForm.cshtml`
    - Used in both `Create.cshtml` and `Edit.cshtml`

```
@model ExpenseViewModel
<form asp-action="@ViewData["Action"]" method="post">
    @Html.AntiForgeryToken()
    <div>
        @Html.LabelFor(m ⇒ m.Category)
        @Html.TextBoxFor(m ⇒ m.Category)
        @Html.ValidationMessageFor(m ⇒ m.Category)
    </div>
    <div>
        @Html.LabelFor(m ⇒ m.Amount)
        @Html.TextBoxFor(m ⇒ m.Amount)
        @Html.ValidationMessageFor(m ⇒ m.Amount)
    </div>
    <button type="submit">Save</button>
</form>
```

Create View:

```
@model ExpenseViewModel
@{
    ViewData["Action"] = "Create";
}
@Html.Partial("_ExpenseForm", Model)
```

Edit View:

```
@model ExpenseViewModel
@{
    ViewData["Action"] = "Edit";
```

```
}
@Html.Partial("_ExpenseForm", Model)
```

■ **Short summary line**

> I used Partial Views for reusable pieces like the Add/Edit Expense form so the same UI and validation are shared across multiple views.

---

3) How do you structure folder layout in MVC?

■ **Explanation (simple, conceptual)**

I followed the standard MVC folder structure to keep things organized and easily maintainable.

■ **When/Why is it used**

- To quickly locate controllers, views, and models
- To keep feature-specific views inside corresponding controller folder
- To make onboarding and collaboration easier for team members

■ **Example / Folder structure from my project**

```
/Models
   Expense.cs
   Category.cs
   ViewModels/
      ExpenseViewModel.cs
      DashboardViewModel.cs

/Controllers
   HomeController.cs
   ExpenseController.cs
   CategoryController.cs

/Views
   /Shared
      _Layout.cshtml
      _SummaryCard.cshtml
      _ExpenseForm.cshtml
```

```
    /Home
       Index.cshtml
    /Expense
       Index.cshtml   (list)
       Create.cshtml
       Edit.cshtml
       Details.cshtml

 /Services
    IExpenseService.cs
    ExpenseService.cs

 /Repositories
    IExpenseRepository.cs
    ExpenseRepository.cs
```

I also maintain:

- `wwwroot/` for CSS, JS, images
- `appsettings.json` or `web.config` for configuration

■ Short summary line

> I followed a clean MVC structure: Controllers, Models, Views (with feature folders), plus separate Services and Repositories for business and DB logic.

---

4) How do you handle form validation in MVC project?

■ Explanation (simple, conceptual)

I handle validation on **two levels**:

1. **Client-side** using Razor helpers + unobtrusive validation (jQuery) for immediate feedback.
2. **Server-side** using **Data Annotations** on ViewModel and checking `ModelState.IsValid` in the controller.

■ When/Why is it used

- Client-side → better user experience
- Server-side → security & correctness (never trust only frontend)

- Ensures that invalid data does not reach service/DB layer

■ Example / Code

ViewModel with Data Annotations:

```
public class ExpenseViewModel
{
    [Required]
    public string Category { get; set; }

    [Required]
    [Range(1, double.MaxValue, ErrorMessage = "Amount must be greater th
an 0")]
    public decimal Amount { get; set; }

    [Required]
    public DateTime Date { get; set; }
}
```

Controller:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create(ExpenseViewModel model)
{
    if (!ModelState.IsValid)
    {
        // Return same view with validation messages
        return View(model);
    }

    _expenseService.AddExpense(model);
    TempData["Success"] = "Expense saved successfully";
    return RedirectToAction("Index");
}
```

View (.cshtml):

```
@model ExpenseViewModel
<form asp-action="Create" method="post">
    @Html.AntiForgeryToken()
    <div>
        @Html.LabelFor(m ⇒ m.Category)
        @Html.TextBoxFor(m ⇒ m.Category)
        @Html.ValidationMessageFor(m ⇒ m.Category)
    </div>
    <div>
        @Html.LabelFor(m ⇒ m.Amount)
        @Html.TextBoxFor(m ⇒ m.Amount)
        @Html.ValidationMessageFor(m ⇒ m.Amount)
    </div>
    <button type="submit">Save</button>
</form>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

■ **Short summary line**

> I use Data Annotations + ModelState on server and unobtrusive validation on client to ensure robust and user-friendly form validation.

---

5) MVC best practices you follow?

■ **Explanation (simple, conceptual)**

I follow best practices focusing on **separation of concerns, reusability, and security** to keep the MVC application maintainable and scalable.

■ **When/Why is it used**

- To avoid “God controllers” and messy views
- To make future changes safe and predictable
- To keep code clean for team collaboration and code reviews

■ **Some key best practices I follow**

1. **Thin Controllers, Fat Services**
   - Business logic in Services, not in controllers

```
// Controller: just coordinates
public IActionResult Create(ExpenseViewModel model)
{
    if (!ModelState.IsValid) return View(model);
    _expenseService.AddExpense(model);
    return RedirectToAction("Index");
}
```

2. **Use ViewModels for Views**
   - Never expose Entity directly to Views; use ViewModels / DTOs.
3. **Use Partial Views & Layouts**
   - Reuse common UI parts (header, footer, forms, cards).
4. **Validation at Model level + Server-side checks**
   - Always verify with Data Annotations and ModelState.
5. **Security Best Practices**
   - `[ValidateAntiForgeryToken]` on POST actions
   - `[Authorize]` and role-based access for restricted screens
6. **Use Dependency Injection**
   - Controllers depend on interfaces (`IExpenseService`), not concrete classes.
7. **Logging and Error Handling**
   - Use centralized exception handling/middleware/filters rather than per-action try/catch.

■ Short summary line

> I keep controllers thin, use services & repositories, apply ViewModels, proper validation, DI, security attributes, and partial views to maintain a clean and scalable MVC application.

This covers everything **MVC-based questions for full-stack roles**.
