Architecture Handbook

Contents

[Object Oriented Design Principles: 2](#_Toc152508759)

[Interviewer’s Question – how to deal with situation 2](#_Toc152508760)

[Software Development Approaches 2](#_Toc152508761)

[Monolithic architecture vs. Microservices vs. Miniservices 3](#_Toc152508762)

[Access control 3](#_Toc152508763)

[Design Approaches 4](#_Toc152508764)

# Object Oriented Design Principles:

|  |  |
| --- | --- |
| Object Oriented Principles | Detail |
| SOLID | **S** - Single-responsiblity Principle   * A class should have one and only one reason to change, meaning that a class should have only one job.   **O** - Open-closed Principle   * Objects or entities should be open for extension but closed for modification.   **L** - Liskov Substitution Principle   * The Liskov Substitution Principle states that subclasses should be substitutable for their base classes. * Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.   **I** - Interface Segregation Principle   * A client should never be forced to implement an interface that it doesn’t use, or clients shouldn’t be forced to depend on methods they do not use.   **D** - Dependency Inversion Principle   * Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions. |
| KISS | **K**eep **I**t **S**imple, **S**tupid!   * Most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided. |
| DRY | **D**on't **R**epeat **Y**ourself   * Reduce Repetition by abstraction, data normalization. Extract common functionality. |

# Interviewer’s Question – how to deal with situation

|  |  |
| --- | --- |
| Reaction approach | Detail |
| STAR | **S**ituation, **T**ask, **A**ction, **R**esult |
| SOARA | **S**ituation, **O**bjective, **A**ction, **R**esult, **A**ftermath |

# Software Development Approaches

|  |  |
| --- | --- |
| Software Development approaches | Detail |
| TDD | **T**est **D**riven **D**evelopment   * Write a failing test * Write the code to make the test pass * Refactor the code to improve its quality |
| BDD | **B**ehavior **D**riven **D**evelopment  Extended from TDD, a test specification is a behavior   * Define the behavior of the software * Write tests that describe the behavior * Write code to make the tests pass * Refactor the code to improve its quality |
| DDD | **D**omain **D**riven **D**esign  Has broader scope that TDD and BDD   * Understand the domain of the software using **ubiquitous** language (or shared language, no concrete example for it, probably a dictionary or glossary, a readme file etc. where all the User Case, Class and Activity diagram coming from it.) * Define the domain model * Implement the domain model in code * Continuously refine the domain model |

# Monolithic architecture vs. Microservices vs. Miniservices

|  |  |
| --- | --- |
| Architecture Patterns | Detail |
| Monolithic architecture | A traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications.:   * Monolithic applications have all business services and functions, including their supporting databases, deployed as a single platform, software development and deployment are relatively faster and easier. * Debugging is also more straightforward because you can open up the entire project within a single IDE instance * But complex to maintain, refactor and reuse |
| Microservices | A collection of autonomous microservices designed around specific business capabilities , granularity is smaller:   * Each service has only one single responsibility * Each service has its own database * All services are developed, deployed, maintained, and run independently of other microservices * Microservices are loosely coupled * All microservices communicate with each other via an event-driven communication |
| Miniservices | As monoliths are challenging to scale because of size, and microservices are a lot more complex to orchestrate and maintain, miniservices architecture fits the middle ground between monolith and microservices architectures, a design that assumes a more realistic approach to implementing the microservices concept:   * Related services can share the same database * Related services can share codebase and infrastructure * Due to its derived design, miniservices inherit all benefits of a microservice architecture including scalability, fault tolerance, and robustness. |

# Access control

|  |  |
| --- | --- |
| Types | Detail |
| Role based | Roles can be thought of as Job Titles. Like "Sales Manager", "Marketing Manager", "Admin" etc.  Role-based Access Control:  [Authorize(Roles="Sales Manager")]  public ActionResult CreateCustomer()  {  return View();  }  [Authorize(Roles="Marketing Manager")]  public ActionResult EditLandingPage()  {  return View();  } |
| Claim based | Claims can be broader than a Role. You can think about Claim as a TAG. For example, you can tag a person as "Friendly", "Talkative", "European", "Photographer", "Adult-who-is-18-years-old" etc. Technically, a role can be thought of as a claim too, but claims cannot be thought of as roles.  Claim-based Access Control, Claims="Adult-over-18years") is a good example:  [ClaimsAuthorize(Claims="Senior-Employee, Award-Winner-Employee, Experienced-On-Sales")]  public ActionResult CreateCustomer()  {  return View();  }  [ClaimsAuthorize(Claims="Trust-worthy-Employee, President")]  public ActionResult DeleteCustomer()  {  return View();  }  [ClaimsAuthorize(Claims="Adult-over-18years")]  public ActionResult ViewImagesOfViolence()  {  return View();  } |
| Permission based | Define permission and create an administrative table to link it to who (Role/Claim) can get access to what (Role):  [Authorize(Permission="CanCreateCustomer")]  public ActionResult CreateCustomer()  {  return View();  } |

# Design Approaches

|  |  |
| --- | --- |
| Name | Detail |
| API First | The API-first approach prioritizes APIs at the beginning of the software development process, positioning APIs as the building blocks of software. API-first organizations develop APIs before writing other code, instead of treating them as afterthoughts. This lets teams construct applications with internal and external services that are delivered through APIs. |
| Service | * Functionality * Independence * Reusability * Scalability * Security * Testability * Maintainability |

# Architecture Process

* Understand the System’s Requirements (Functional Requirements)
* Understand the Non-Functional Requirements
* Map the Components
* Select the Technology Stack
* Design the Architecture
* Write Architecture Document
* Support the Team