BECOMING A SOFTWARE ARCHITECT

Code It Up Workshop Vol. 4

FOR QUESTIONS DURING THE LIVE EVENT

https://sli.do **#softarch**

The YouTube Live Chat Is Also Monitored

FOR QUESTIONS IF YOU WATCH THE RECORDING

Just Send Me A Message

wewritesoftware@gmail.com or Messenger

https://www.facebook.com/groups/codeitup/

FOR LENGTHY DISCUSSIONS

Use The Facebook Group

https://www.facebook.com/groups/codeitup/

RESOURCES

http://bit.ly/software-architectures

LIVE STREAM TROUBLESHOOTING

- Sometimes issues happen during a live stream…
  - And sometimes disasters happen, this is how memes are born…
- If my Internet goes down and the stream stops:
  - Wait for 5 minutes, I have another one on a different network
- If YouTube is showing "stream ended":
  - I will post a new link in the comments section below this video
- If something else happens unexpectedly:
  - Well, I will add a solution to this slide during my next event… ☹
- If a major showstopper is happening – no electricity, for example
  - I will write in the Code It Up group and we will schedule a new date…

THE PRESENTER

- **Ivaylo Kenov – Quality Code Advocate**
  - Various job titles at the same time:
    - Organizer & Speaker @ Code It Up
    - CTO @ SoftUni
    - Full Stack Technical Trainer @ Everywhere
    - Code General @ https://MyTestedASP.NET
    - Meme Copy Machine @ Daily Programming Fun
    - {*Insert Job Title Here*}
  - Contacts
    - https://github.com/ivaylokenov
    - https://facebook.com/ivaylo.kenov
    - https://linkedin.com/in/kenov
    - https://www.instagram.com/ivaylokenov/
  - YouTube
    - https://youtube.com/MyTestedASPNETTV

EXCLUSIVE WORKSHOP PARTNER – LAB08

- Lab08 offers product development services
- The company works with clients from the Scandinavian market, developing web platforms in various domains
  - Video-based user testing
  - Issuance and verification of digital credentials
  - Marketing through gamification and influencer-based social media marketing
- When a Lab08 employee seeks development - the person can move on to a new product or domain within the company, being able to explore the product area in depth
- Careers: https://lab08.com/careers/

CODE IT UP MENTORSHIP PROGRAM ON PATREON

- Target – junior to regular programmers with 0 to 2 years practical work
- Help with:
  - Becoming better software developers
  - Code improvement in terms of quality and logic
  - Career choices and advancement
  - Interview preparation
  - And more
- Approach – private groups, exclusive lessons, live classes, one-on-one meetings
- From 2021 – project mentorship – DDD, Microservices, CI/CD, Kubernetes, and more
- More information on Patreon: **https://www.patreon.com/ivaylokenov**

WHAT ARE WE GOING TO COVER

- About Code It Up
- About This Workshop
- Why Software Architecture
- The Architect And The Team
- What Makes A Great Architect
- What Is Software Architecture?
- Unified Modeling Language
- Designing Solution Architectures
- Common Technology Stacks
- Architecture Design Patterns
- Choosing The Right Patterns
- Architecture Quality Attributes
- System-Wide Considerations
- Deployment Considerations
- The Architecture Document
- Designing A Real-Life Solution

ABOUT CODE IT UP

CODE IT UP

- The Code It Up initiative:
  - Aims to provide detailed knowledge on advanced software development topics
  - Aimed at people with at least 1 year of C# experience
- Code It Up Online
  - Free live-streamed online events
  - Led by me – mainly .NET, architecture, and infrastructure
- Code It Up Guests
  - Free live-streamed online events
  - Guest speakers from the industry
  - Variety of technologies
- Code It Up Workshop
  - Paid events containing theory & practical exercises for the attendees

UPCOMING CODE IT UP EVENTS

- Free upcoming events:
  - January 12 - Let's Get Functional With C#
  - February 13 – Enter the Swarm: Native Docker Clustering
- Events with no specific order or dates:
  - The Limits of the C# Runtime
  - API Scenarios - REST, GraphQL & gRPC
  - Serverless Real-Time C# Applications? Yes, It's Possible & Easy!
  - C# On Rocket Fuel – Writing High Performance .NET
  - Software Engineering – Patterns & Anti-Patterns
  - Let's Learn the Roslyn Compiler!
  - Security Considerations We Should All Know!
  - Plus many more!

THANKFUL IF YOU SHARE A STORY

- I am super excited about this lecture!
- You can be extremely helpful to Code It Up
- Just share a story on Facebook or Instagram during the lecture
- Make sure you tag me so that I can reshare your post - **@ivaylokenov**
- Bonus – add the **#codeitup** hashtag
- Thank you! You rock!

ABOUT THIS WORKSHOP

ABOUT THIS WORKSHOP

- A theoretical lecture on software architectures
  - Widely-used design patterns in production
  - Depending on your level, you may be familiar with some of the topics
  - A real-life project example
- A practical guidebook for architecting various solutions
  - 5 real-life projects on more than 50 pages
  - Legacy systems, vast data load, lots of concurrent users, working with critical data, and more
  - Please report in the group, if you find any “bugs” in the book!
- And I have a lot more to add in the future!
  - You receive free updates of all resources!
- MOST IMPORTANTLY – HUGE THANK YOU! <3

ABOUT THIS WORKSHOP

- HUGE DISCLAIMER!
- **There are a lot of things to know about software architectures**
  - **You may recognize some of the patterns**
  - **And we most probably will not mention all of them**
- The technology world is moving very fast
  - Some of the example shown here may be considered anti-patterns in the future
  - But the overall concept and process stays the same
- As all my workshops – this one is super intense too!
  - So, give yourself time and most importantly – finish the book!
  - And don't worry! The knowledge provided here will save you weeks of reading!

WHY SOFTWARE ARCHITECTURE

WHY THIS TOPIC

- A little back-story, first…
- 4-5 years ago, I was on interview for a tech lead or a software architect
  - Depending on my skills and knowledge
- And after the interview, they told me "you are great, but you need a little bit more to become an architect"
  - I was both disappointed and motivated
- So, I started researching a lot about software architectures and patterns
  - I purchased lots of courses, read lots of books
  - Started analyzing how the big guys are doing it – StackOverflow or Netflix, for example
- This workshop is the culmination of the story above! Enjoy!

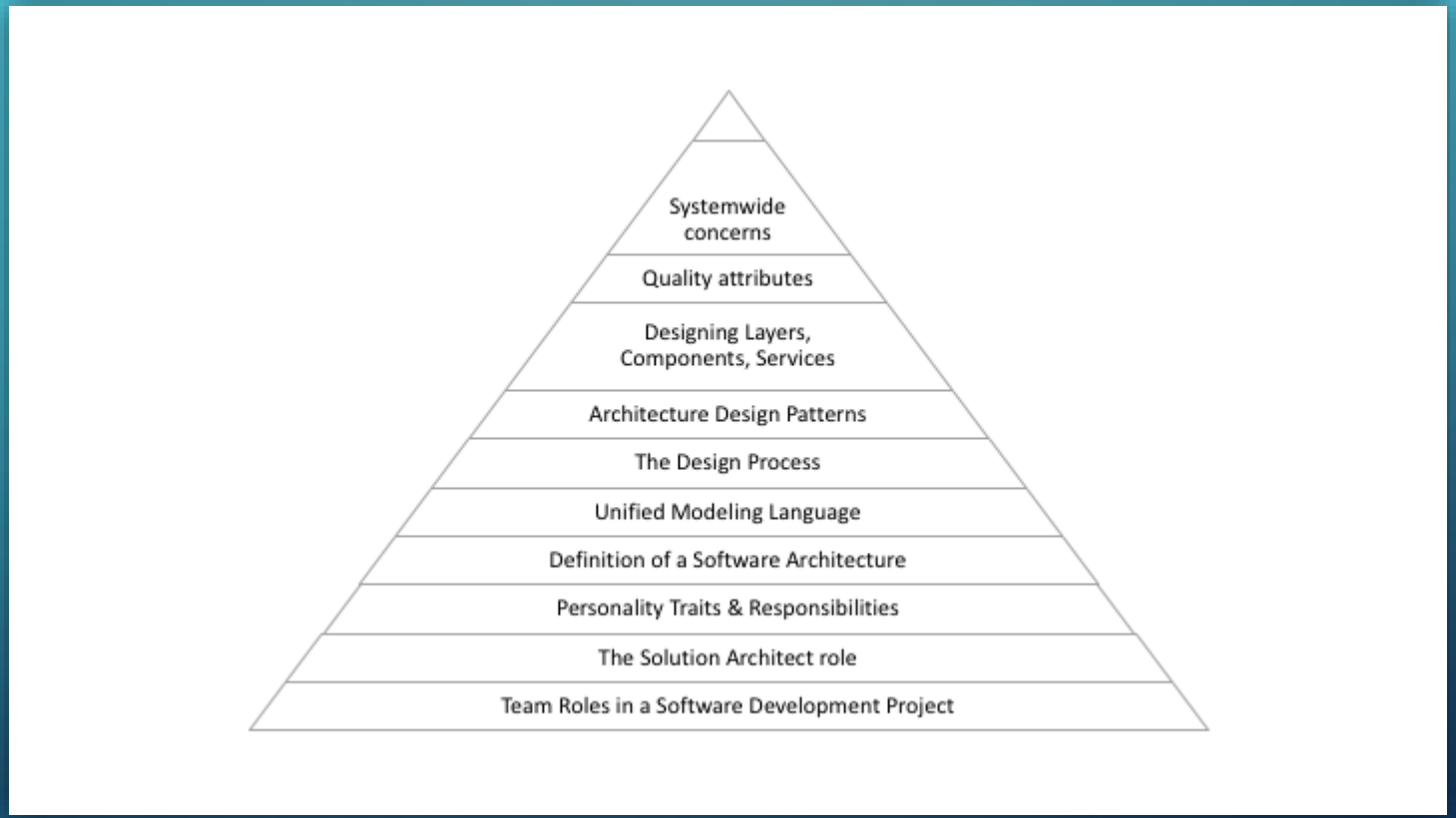
WHY SOLUTION ARCHITECTS

- Why do we need solution architects?
  - If we have a perfect tech lead?
  - And very skilled developers?
- Consider constructing a building…
  - Imagine if there is no design and architecting plans
  - Just the workers and their team leader coming every day to work
- It is the same with software development!
  - We need an architect to design and lead our high-level solution!

REASONS TO BECOME ARCHITECTS

- It is interesting!
  - You will have way more variety of work to do!
  - You will solve different challenges!
- Career path and visibility!
  - You will meet powerful people – CEOs and CTOs!
  - You learn a lot from such individuals!
- Money!
  - Software architects may earn double the salary of senior developers!

THE KNOWLEDGE PATH

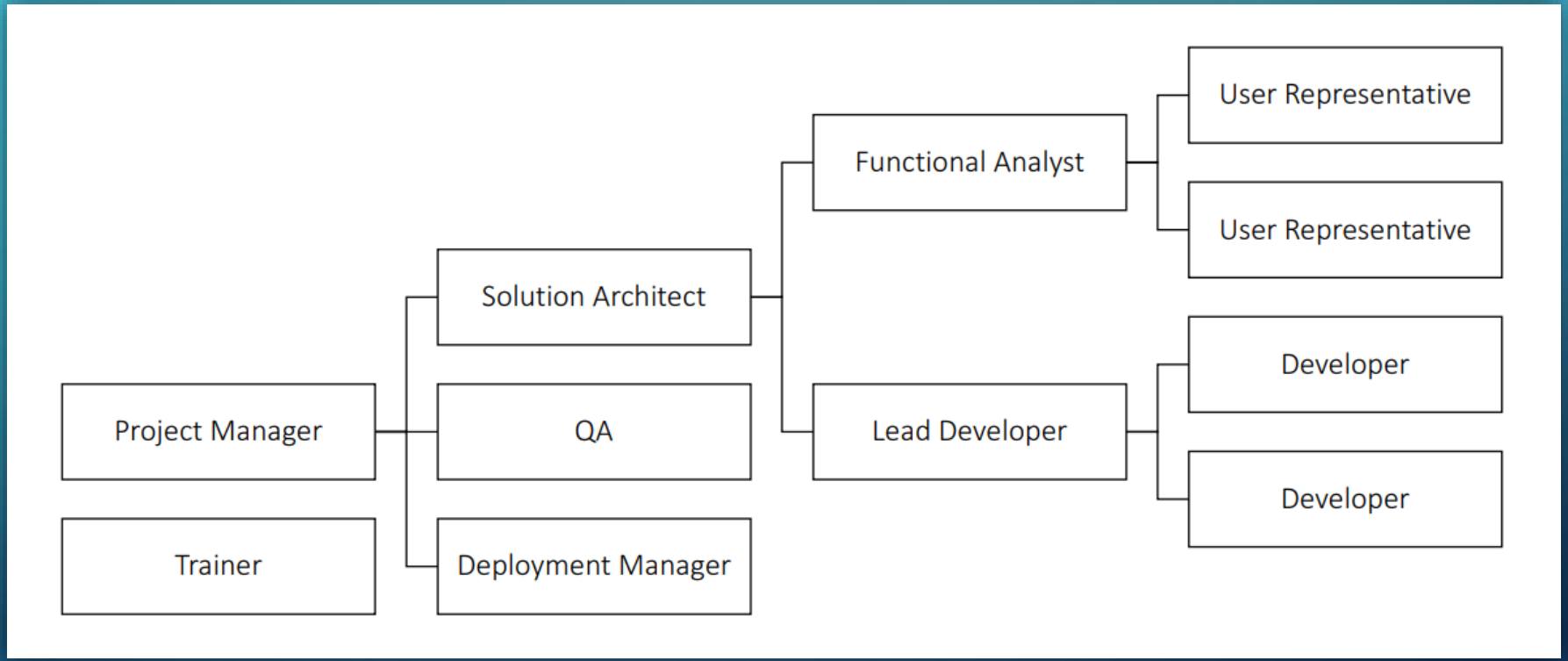


THE ARCHITECT AND THE TEAM

PRINCIPLES OF SOFTWARE DEVELOPMENT

- The development process is different in each company
- But there is always a need to:
  - Understand business problem
  - Document non-technical business solution
  - Convert solution to technical architecture
  - Convert architecture to code
  - Manage developers
  - Test code
  - Deploy code
- Of course, in some companies these responsibilities are mixed

MAIN ROLES IN SOFTWARE DEVELOPMENT



REQUIRED SKILLS BY EVERYBODY IN THE TEAM

- Understanding the Business
  - The more business understanding you have, the more useful you will be
- Cross-Domain Understanding
  - You know bits of the other roles in the team as well
- Multiple Perspectives
  - You need to see the problem from everyone's point of view
- People Skills
  - Just to make sure communication goes smoothly
- Lifelong Learning
  - IT is moving super fast; you need to stay relevant

RESPONSIBILITIES OF THE FUNCTIONAL ANALYST

- A product/project owner
- Responsible for functionality
- Captures, consolidates, and communicates information
- Constantly asks questions:
  - What do you mean?
  - How does this fit in with…?
  - Etc.
- Identifies and resolves conflicts
- Produces requirements specification

SKILLS OF THE FUNCTIONAL ANALYST

- Precise communicators
- Great attention to detail
- Adept at dealing with differing opinions and conflicts
- Know when detail is necessary and when not
- Great relationship skills
- Very good listener
- Can create clear and precise documents
- Skilled in using Office tools

IF YOU WANT TO BE FUNCTIONAL ANALYST

- Pros:
  - Key role
  - Lots of interactions
- Cons:
  - Must work with bad user representatives
  - Can expect conflict
  - Will receive blame if functionality is missing

RESPONSIBILITIES OF THE LEAD DEVELOPER

- Leads and mentors developers
- Assigns tasks to developers
  - Based on their skill level
- Details and partitions work
- Ensures that all developers are successful
- It is not always an official role
  - Usually, it is the person who helps everyone else

SKILLS OF THE LEAD DEVELOPER

- Awesome programming skills
- Willing to mentor and be value-driven
- Grows out of the Developer role
- Requires great relationship with Architect
- Wide knowledge of libraries/tools/techniques
- Adept at creating technical specifications
- Adept at build & configuration management
- Adept at debugging, post-mortem log inspection, etc.
- Can create own tools if needed

IF YOU WANT TO BE LEAD DEVELOPER

- Pros:
  - Lead-in to a Solution Architect
  - Involves coding (optional, not necessary, but suggested)
  - Can pick & choose cool tasks
- Cons:
  - Can get squeezed between the Solution Architect and the Developers
  - Lifelong learning required
  - Turns into a Developer if the Project Management is weak
    - Loses motivation
  - Team might be too small
  - Vulnerable to offshoring

RESPONSIBILITIES OF THE SOLUTION ARCHITECT

- Massive responsibility
  - Functional understanding, technical knowledge and leadership skills
- Responsible for the technology stack
- Converts functional requirements to a technical architecture
- Carefully balances patterns/requirements/elegance/concepts
- Researches key technologies
- Has deep understanding of design and architectural patterns
- Motivates and guides development team
- Ensures that the Lead Developer is successful

SKILLS OF THE SOLUTION ARCHITECT

- Grows out of Lead Developer role
- Requires great relationship with Lead Developer
- Always maintains helicopter view
  - It is not bad to help with code, but there is another role for that
- Deep understanding of design patterns
- Fluent in UML or other design tools
- Experience with tools & code generators

IF YOU WANT TO BE SOLUTION ARCHITECT

- Pros:
  - High value position
  - Great salary
  - Visible role
  - Lots of interactions
  - Safe from outsourcing
- Cons:
  - Difficult to stay up to date
  - Difficult to get right
  - Can receive bad requirements
  - First in line to receive blame

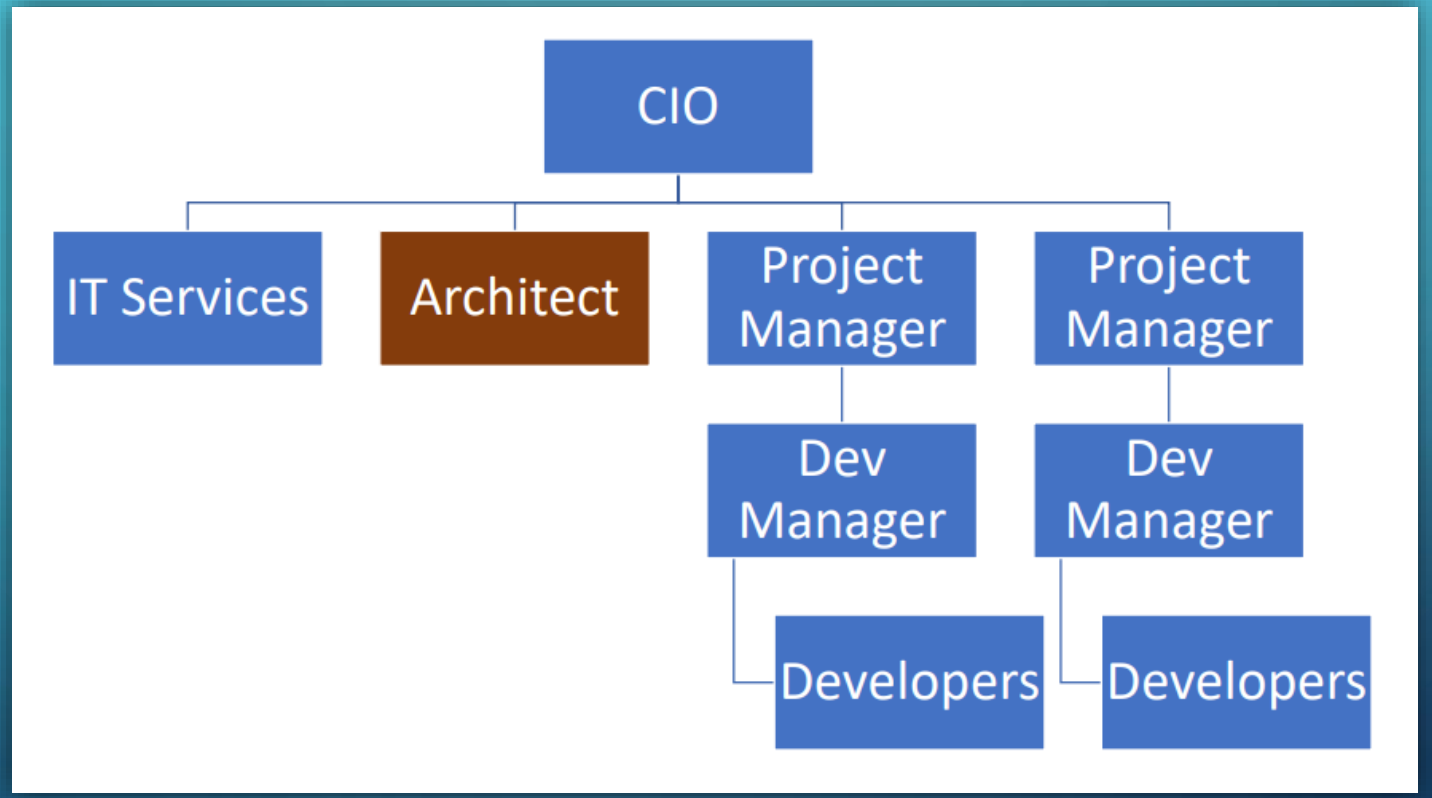
TYPES OF ARCHITECTS IN THE IT WORLD

- Infrastructure Architects
  - Design the infrastructure
  - Servers, VMs, network, storage, etc.
  - Familiar with requirements
  - Promoted from Infrastructure Expert
- Solution/Software/System Architect
  - Responsible for the architecture of the software
- Enterprise Architect
  - Works with top level management - CEO, CIO
  - Streamlines the IT to support the business
  - No development-oriented tasks
  - Promoted from Senior Solution Architect / Project Manager

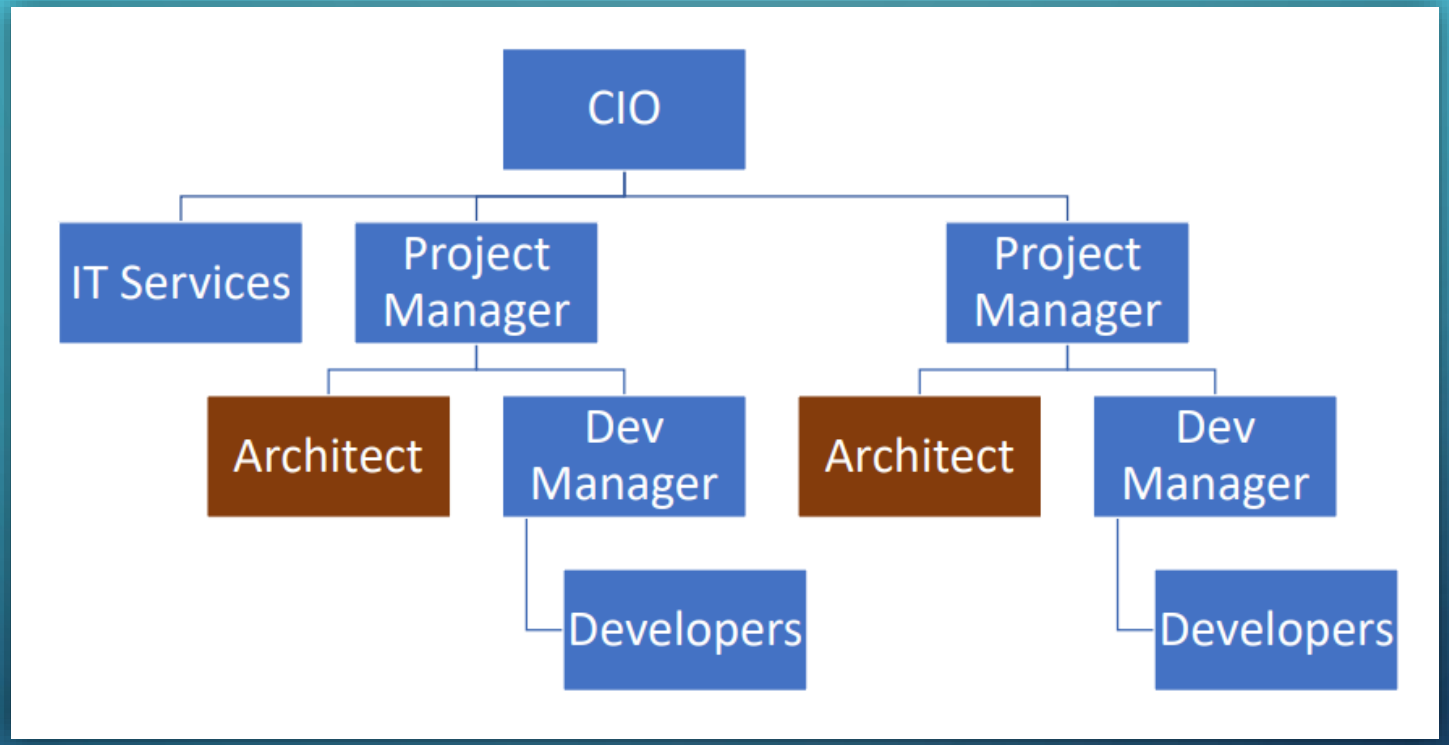
IDEAL WORLD VS REAL-LIFE

- We've looked at software development team roles in an ideal world
- Real-life is not exactly like that
  - Missing roles
  - Individuals with conflicting roles
  - Disempowered roles
  - Other inconsistencies
- The organization itself might lack the understanding of how developer teams are supposed to function

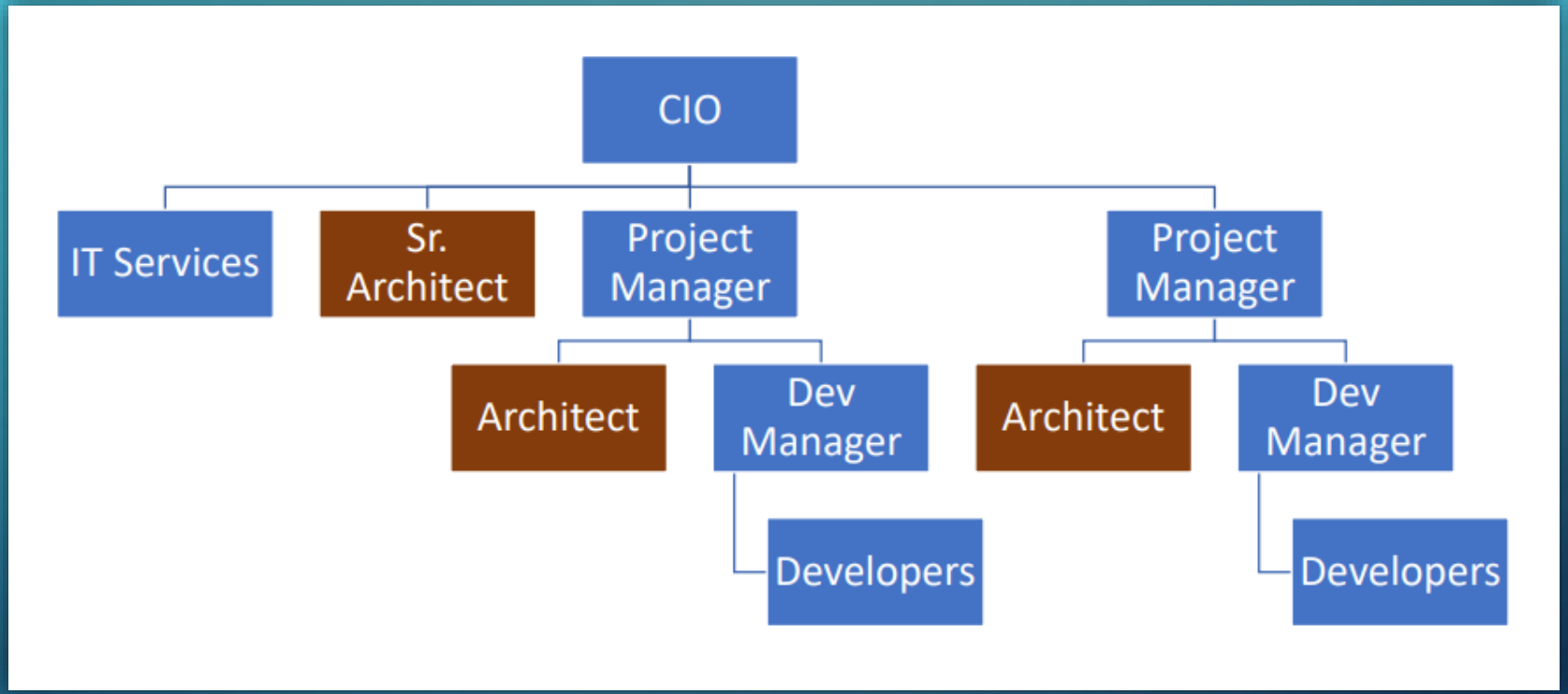
ORGANIZATIONAL CHART EXAMPLE 1



ORGANIZATIONAL CHART EXAMPLE 2



ORGANIZATIONAL CHART EXAMPLE 3



THE TYPICAL WAYS TO BECOME ARCHITECT

- From Senior Developer
  - Usually in small companies
- From Team Leader / Lead Developer
  - Very common path
  - Usually, the choice is between an Architect and a Project Manager
- From Development Manager after being Team Leader
  - Usually in bigger companies with bigger structures

IF YOU GET THE ROLE

- If you get a more advanced role and the organization is dysfunctional
- Do not go into firefighting mode!
  - Resist the temptation to fix what is not working optimally
  - You will be swamped with work and you will never have time to fix the process
- Allow the old structure to burn into the ground!
- And create something better!

ASSESS YOU TEAM

- Think about your current team!
- Are there individuals embodying more than one role at the same time?
- Are these roles compatible, or is there a conflict of interest?
- Is the organization knowledgeable about each role?
- Is everybody aware of the importance of QA, and the need for frequent user testing?
- There is nothing to be worried about though!
  - My team is not perfect too!
- Just try to always improve the process!

MY TEAM ASSESSMENT

- Throughout the years I was in various situations
  - Without any management (be my own "everything")
  - In a team without technical leadership
  - In a team with bad technical leadership
  - In a team with mixed roles
  - In a team with being every single role
  - In my current team, we are almost perfect
    - Started from having no development management
    - We build it to the perfect diagram without a functional analyst
    - I helped a lot of people grow and convinced the CEO we need to hire leaders
    - And I am quite happy with our accomplishment ☺

WHAT MAKES A GREAT ARCHITECT?

EXAMPLE JOB DESCRIPTIONS

- "Make intuitive high-level decisions."
- "You will see the big picture and create architectural approaches for software design and implementation to guide the development team."
- "Strong technical background and excellent IT skills."
- "Experienced in designing…"
- "Unified vision for software characteristics and functions."
- "Provide a framework for the development of a software or system that will result in high quality IT solutions."

EXAMPLE REAL-WORLD RESPONSIBILITIES

- Collaborate with team to draft functional and non-functional requirements (mixed)
- Use tools and methodologies to create representations for functions and UIs (architect)
- Develop high-level product specifications (architect)
- Define all aspects of development (lead)
- Communicate all concepts and guidelines to development team (lead)
- Oversee progress of development team (mixed)
- Provide technical guidance and coaching to developers and engineers (lead)
- Ensure software meets all requirements (mixed)
- Approve final product before launch (mixed)

EXAMPLE REQUIREMENTS

- Proven experience as software architect
- Experience in software development and coding
- Excellent knowledge of software and application design and architecture
- Excellent knowledge of UML and other modeling methods
- Familiarity with UI/UX design
- Understanding of software quality assurance principles
- A technical mindset with great attention to detail
- High quality organizational and leadership skills
- Outstanding communication and presentation abilities
- MSc/MA in computer science, engineering or relevant field

RESPONSIBILITIES OF A GREAT ARCHITECT

- There are four main groups of responsibilities
  - Insight
    - You need to understand the software development and the business domain
  - Leadership
    - You need to be able to mentor others and advance their skills
  - Vision
    - You need to understand the business and its direction
  - Communication
    - You need to be able to express yourself and defend your decisions

INSIGHT

- Abstract Complexity
  - A great architect abstracts the complexity of a system into a manageable model.
  - The model describes the essence of a system by exposing important details and significant constraints.
  - Divide a system into subsystems.
- Understand Trade-offs
  - People want cheap, fast, quality products. You need to compromise efferently.
  - A great architect makes critical decisions in terms of implementation, operations, and maintenance. These decisions must be backed up by an understanding and evaluation of alternative options.
  - These decisions result in tradeoffs that must be well documented and understood by others.

LEADERSHIP

- Maintain Control
  - Often the system starts to deviate from the initial design.
  - A great architect maintains control over the architecture lifecycle by continuously monitoring that the implementation adheres to the chosen architecture.
  - Communication with the lead developer is required.
- Stay On Course
  - A great architect needs to be flexible and modify the architecture if challenges occur.
  - Not always the initial design with be technically possible.
  - A great architect stays on course in line with the long-term vision. When confronted with scope creep, the architect must know when to say no to some requests in order to say yes to others.

COMMUNICATION

- Explain The Benefits
  - A great architect works closely with executives to explain the benefits and justify the investment in the chosen software architecture.
  - He or she must deliver results that have an impact on the business development.
  - Scale better, attract more customers, sell more products, etc.
- Inspire Stakeholders
  - A great architect inspires, mentors, and educates the team about the solution architecture.
  - Everyone should think that the project is awesome and cool.
  - All stakeholders must be able to understand, evaluate, and reason about the software architecture.

VISION

- Focus On The Big Picture
  - A great architect has a holistic view and always sees the big picture to understand how the software system works.
  - Top level view for at least 6 months or a year ahead is required.
- Act As Change Agent
  - Because of mixed responsibilities. Educate the company. Talk with people.
  - A great architect acts as an agent of change in organizations were process maturity is not sufficient for creating and maintaining the architecture.

PERSONALITY TRAITS TO CULTIVATE

- Steadfast
  - The only thing constant in Software Development is change itself. A great architect must be patient and resilient to adapt to the way stakeholders operate.
  - Budgets get cut. Requirements change. Technologies are deprecated. Systems go offline.
- Trustworthy
  - A great architect conveys a sense of credibility and trust and must be perceived as successful.
  - An architect can attain such status by prior successful experience, formal training in the field, and by his or her ability to deliver successful and relevant results.
  - To grow into a trustworthy position, make sure your previous work is absolutely brilliant.
  - If this is your first architect job – just deliver your first versions as perfect as possible.

PERSONALITY TRAITS TO CULTIVATE

- Persuasive
  - A great architect is a skilled and diplomatic negotiator. Principled negotiation is the tactic of choice for an architect to seek cooperation with the project stakeholders.
  - An architect will be expected to deliver better, faster, and cheaper, but must negotiate to decide which two out of three aspects will be considered first and under what conditions.
- Confident
  - You own this technology. If you choose a solution, be proud of it, and defend it.
  - You can't show your inability to deliver, even if you are insecure about it.
  - In a leadership position, attitude is everything. A great architect believes in his or her ability to perform.
  - An architect must have a passion for success.

COMMON PITFALL FOR NEW ARCHITECTS

- Most of the time, problems occur of not using "power" correctly
- There are three types of power:
  - Implicit Power
    - The power which goes with your job.
    - You are the CEO – if you say something, people will do it, because you are in charge.
  - Granted Power
    - Everybody approaches you as the leader even though you are not formally one.
    - You grew as a lead naturally. The team granted you the power to be their superior.
  - Personal Power
    - The ability to absorb "punches" and recover.
    - Stand back to your feet after a disaster.
- Every problem creates a power imbalance

COMMON PITFALL FOR NEW ARCHITECTS

- Every problem creates a power imbalance
- Here are the most common issues:
  - The Lead Developer loses trust
  - Functional requirements are invalid
  - The planning is too optimistic
  - The architect role holds no power

THE LEAD DEVELOPER LOSES TRUST

- The Solution Architect is being robbed of his implicit power
  - He or she wants his/her design
- The Lead Developer is being robbed of his granted power
  - He or she wants to change the design but does not have the power
- Developers are also affected by this problem
- Endless discussions and implementation starts to deviate from architecture
- The Solution Architect's job is to make the Lead Developer succeed!
  - Empower your Lead Developer
  - Always present a united front
  - Be open to feedback
  - Change architecture if needed

FUNCTIONAL REQUIREMENTS ARE INVALID

- It is very difficult to realize this problem
  - A symptom is usually an unmotivated functional analyst
- But the Solution Architect will be the one who is responsible
  - Executives do not understand the difference between functionality and technology
- The analyst is being robbed from his personal power
  - He is doing a bad job, realizes it, but passes it to the Solution Architect anyway
  - There might be a conflict between the User Representatives and the Functional Analyst
- Solution:
  - Find the reason – why the Functional Analyst is not doing his job correctly?
  - Try to restore the personal power of the Functional Analyst
  - Escalate up if needed – talk to the CTO or CEO

THE PLANNING IS TOO OPTIMISTIC

- The Project Manager is responsible for this problem
  - And is usually unaware of it
- The QA is being robbed of his personal power
  - "We are in a hurry, no time for testing!"
- The Lead Developer is being robbed of his personal power
  - "There is no time for mentoring, training and reviews, just keep pushing code!"
- Solution:
  - Bring the Project Manager in on the dev team
  - Find the reason! Optimistic PM? Naïve Lead Developer? Slow Developers?
  - Coach Lead Developer, Restructure Developers, Readjust schedule, Alter planning

THE ARCHITECT ROLE HOLDS NO POWER

- The Project Manager and Functional Analyst see the Software Architect as a producer of documentation
- The Software Architect is being robbed of implicit power
  - No communication and no leadership
  - The organization does not understand what an architect should do
- The project will most probably fail
  - "We are in a hurry, no time for testing!"
- Solution:
  - Educate the C-section, be a change-maker
  - Team up with Project Management, QA, Functional Analyst, and Lead Developer
  - Become the owner of the entire process
  - Leave if the organization is unwilling to change

ARCHITECTS SHOULD UNDERSTAND THE BUSINESS

- Every architect should understand the business very well
  - Weaknesses
  - Strengths
  - Competition
  - Growth strategy
- Every architect should understand the system goals
  - Goals are not requirements
    - They are not "what the system should do"
  - Goals describe the effect on the organization
  - They are usually described by a client
  - Think about the big picture
  - Work for your client's clients

SYSTEM GOALS EXAMPLES

- HR system
  - Organization – product-oriented company
  - Goal – streamline the recruitment process
- Reporting & mapping criminal incidents system
  - Organization – large city
  - Goals
    - Improve police response time
    - Attract new residents
- Mobile flash sales system
  - Organization – small startup
  - Goals
    - Generate quick revenue stream
    - Attract investors

ARCHITECTS SHOULD WATCH THEIR LANGUAGE

- With Project Managers – they care about project success and budgets:
- Avoid:

> *"This is the latest and greatest pattern, and we'll be the first to test it out! We can write a blog post about it!"*

- Use:

> *"This new technology can help us write the code twice as fast, so we can cut our schedule and budget accordingly!"*

ARCHITECTS SHOULD WATCH THEIR LANGUAGE

- With Lead Developer – they care about code and technology:
- Use:

```
"Have you heard about the latest React version?
We're going to use it!"
```

ARCHITECTS SHOULD WATCH THEIR LANGUAGE

- With CEO/CTO – they care about financial bottom line:
- Avoid:

```
"Any technical buzzwords/mumbo-jumbo…"
```

- Use:

```
"The architecture I've designed will ensure the continuity of
the business, and will be able to cope with the high loads expected
during Black Friday sales!"
```

SOFT SKILL REQUIRED

- Listening
  - Sounds easy, but it is not. Consider the person you are talking to!
  - Never think you know all about technology! I learnt new tricks from this workshop too!
  - Assume you are not the smartest person in the room!
  - Collective wisdom is better than individual one!
- Dealing with criticism
  - Your work is going to be criticized! You will be asked to explain the logic behind your decisions!
  - Don't attack back! It will put you in a bad position!
  - Consider the reasons:
    - Genuine questions – provide facts and logic. It is OK to say "Good question, let me think about it!"
    - Mocking you – think about the "why" here, but never get offended. It is not you, it's probably them!

SOFT SKILL REQUIRED

- Be smart, not right!
  - For example – you are presenting your architecture in front of the CTO and senior experts
  - And somebody questions your design in a calm and professional manner:
    - "I believe the stateless pattern is useless here. It will hinder the performance!"
  - You know you are right, but do not attack back
  - Your goal is to get an approval of the architecture
  - So, don't do the thing to make you feel right. Don’t say:
    - "You are wrong. The system is under heavy load. The stateless pattern will help us with that!"
  - Instead, answer smartly to get everyone's approval:
    - "That's a great point. We actually had a lot of thinking about this specific issue, and we believe this is the better option. Let's have a further meeting to discuss it in more detail?"
  - This way you got more respect and avoided a long and boring technical argument (and most probably the meeting afterwards)

SOFT SKILL REQUIRED

- Deal with organizational politics
  - We all want patterns and code, but you must cope with the politics
  - For example, the CTO of the project doesn't want to do microservices:
    - "It's new, untested, and immature pattern. It is too risky!"
  - No matter the case studies we provide him, he refuses to accept microservices
  - So instead of searching for technical solutions, try to understand the situation
  - It turns out the CTO is going to be replaced soon and he is just rejecting everything
  - So, a better approach is to prepare the future CTO and discuss the patterns with him
  - Being aware of organizational politics, but never be part of them!
    - It may give you short-term leverage, but in the long-term, it will hurt you badly!

SOFT SKILL REQUIRED

- Public speaking
  - Architects have no formal authority
  - They do not take the final decision
  - Therefore, you can influence using speaking skills
    - You will be part of many meetings
  - You must be able to stand in front of an audience and present your solution
  - Here are the main concepts:
    - Define a clear goal
    - Know your audience
    - Be confident or pretend it
    - Don't read the content from slides
    - Maintain eye contact
  - Take a course on public speaking. It is very important skill!

SOFT SKILL REQUIRED

- Constant learning
  - The development world changes in a blink of an eye
  - Always keep learning
  - A lot of frameworks get completely forgotten in months
  - Consider jQuery or Grunt
    - Some of you may have never used it!
  - Angular is also dying right now
    - There are better alternative – React, Vue, the new kid Svelte
  - You may not like that, but keep adapting
    - You do not have to know the tools in detail
    - That's the developer's job
    - You should know when and why to use a tool
  - Main source – blogs, books and webinars

ASSESS YOURSELF

- Find your personality traits
- There is a good test for that
  - 16 Personalities - https://www.16personalities.com/
- Great architect is that it requires somewhat incompatible personality traits
- You need very strong intuition, insight, and vision
  - Introvert-Analytical personalities are quite strong in that area
- But you're also going to need strong communication and leadership skills
  - This requires an Extrovert-Feeling personality
- Find your weaknesses in the 4 architect skills
  - Insight, Leadership, Vision, and Communication

MY ASSESSMENT

- My test resulted in Assertive Commander
  - Like Steve Jobs, Gordon Ramsay, Jim Carrey, Dr. Strange, and others
- I love good challenges and achievements
- I have ruthless level of rationality
  - But I overwhelm the more sensitive people
- Strength – efficient, energetic, self-confident, strategic, inspiring
- Weaknesses – stubborn, impatient, intolerant, arrogant, emotionless
- Perfect career path
  - Executive positions or entrepreneurship
  - I crave responsibility, growth, and opportunity
  - Remember the back story?

ASSESS YOURSELF - COMMUNICATION

- Bad communication habits:
  - Criticism – do not criticize without valid arguments and background knowledge
  - Contempt – do not underestimate people and their abilities
  - Defensiveness – do not attach yourself to your own solutions and designs
  - Stonewalling – being stubborn about everything you do not believe in
- Remember that you want to inspire and motivate the people around you
  - You can only accomplish this with healthy, powerful, and uplifting communication
- You can try the questions from the provided document
- Think how you can improve yourself!

ASSESS YOURSELF - LEADERSHIP

- Leadership is tricky business. You need:
  - Socio-Political Power – how you present yourself unconsciously
  - Positional Power – what is current position in a group or organization (paid or unpaid)
  - Informal Power – what power the other people give you "by default"
  - Historical Power – growing up, childhood and teen years
  - Personal Power – make friends, cope with challenges, bounce back from setbacks
- You can try the questions from the provided document
- Think how you can improve yourself!

MY ASSESSMENT

- Throughout the years, I've dealt with all of it
  - Criticism, contempt, defensiveness, and stonewalling
- I learnt with time how to solve these personal issues
  - Mainly by thinking about them during various situations
- I also worked hard for my power
  - Socio-Political Power – I was never the "cool" guy, so I started doing "cooler" things that I like
  - Positional Power – I always accepted challenges and learnt on the fly (remember the back story?)
  - Informal Power – I always try to help and mentor others or at least point them in a direction
  - Historical Power – anxiety, shame, getting bullied, hard to recover from these
  - Personal Power – had a great depression 3 years ago after a huge failure

WHAT IS SOFTWARE ARCHITECTURE?

SOFTWARE ARCHITECTURE

- Structured solution to address all common software attributes:
  - Meets all the requirements
  - Performance optimized
  - Security in check
  - Manageability for developers
  - Scalability for business growth
  - Accessibility for easier user experience
- Major requirements:
  - User requirements – the way end-users interact with the system
  - Business requirements – cheaper, faster, better than competitors
  - IT system requirements – infrastructure requirements

WHAT COMPOSES AN ARCHITECTURE?

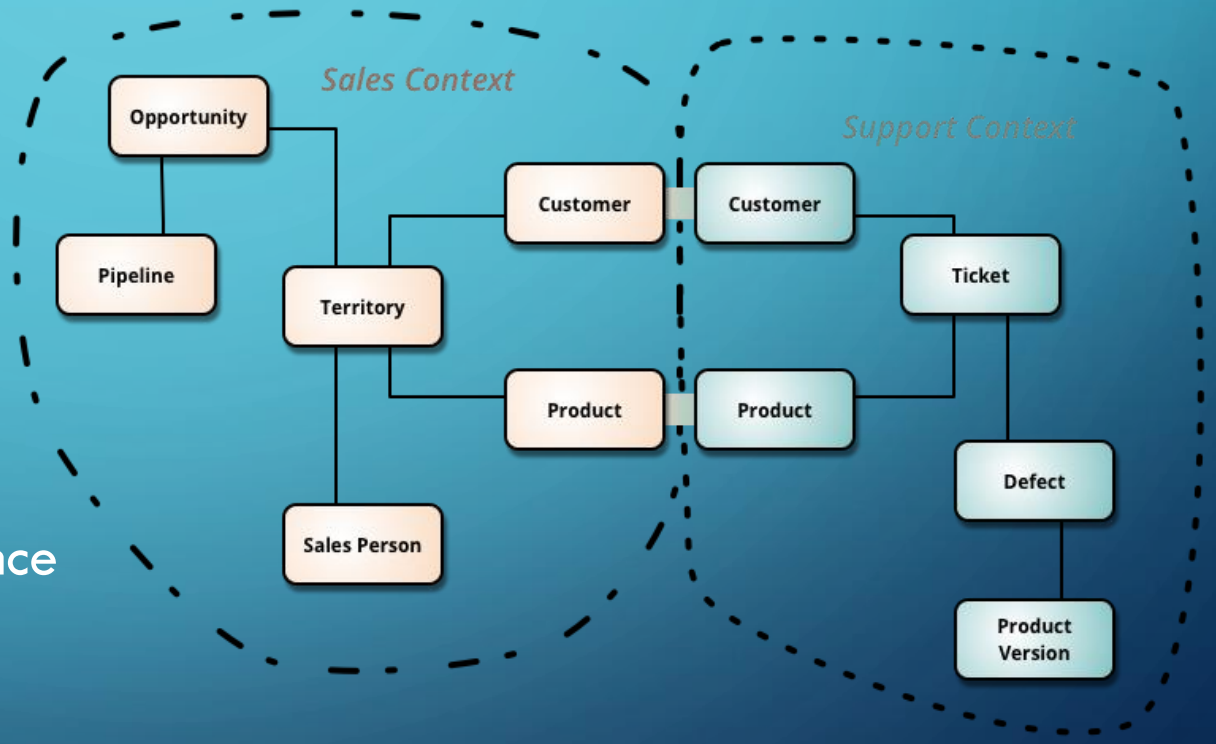
- The structural elements and interfaces composing the system
  - Objects – the low-level building blocks of the system
- How these elements behave in collaboration
  - How communication is done in the architecture
- The composition of elements into larger subsystems
  - How the elements are mapped to trees or graphs – the high-level data structure
- The architectural style that guides this composition

WHAT COMPOSES AN ARCHITECTURE?

- Additionally, our design should cover:
  - Functionality
  - Usability
  - Resilience
  - Performance
  - Economic and technology constraints
  - Trade-offs and aesthetic concerns
- The goal is to:
  - Document high-level structure
  - Do not go into implementation details
  - Minimize complexity
  - Address all requirements
  - Be compatible with all use cases and scenarios

OUR ARCHITECTURE NEEDS

- Separation of Concerns
- Encapsulation
- Dependency Inversion
- Explicit Components
- Single Responsibility
- Don't Repeat Yourself
- Persistence & Infrastructure Ignorance
- Presentation Ignorance
- Bounded Contexts
- Testability



ARCHITECTURE ABSTRACTION

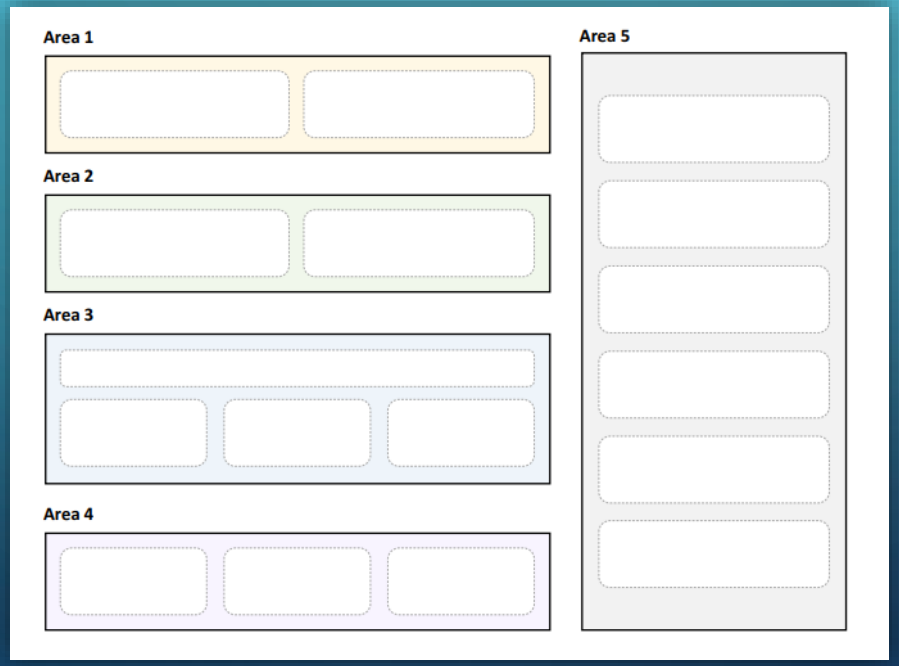
- Layers:
  - System, Sub-systems, Layers, Components, Classes, Data and Methods
- Bad Architecture:
  - Complex, Incoherent, Brittle, Untestable, Unmaintainable
- Good Architecture:
  - Simple, Understandable, Flexible, Testable, Maintainable

SOFTWARE ARCHITECTURE DESIGN TIPS

- Build to change instead of building to last
  - You are going to change your solution multiple times
  - Create around modularity and flexibility
- Use models, but only to analyze and reduce risk
  - The models should be more abstract, otherwise you are taking the role of a developer
- Use visualizations to communicate and collaborate
  - Your architecture is going to be a large diagram
- Identify and research critical points of failure
  - Put in work in research in everything advanced
  - Security, scalability, resiliency

AREAS OF SOFTWARE ARCHITECTURES

- The goal of a software architect is to minimize complexity
- This can be accomplished by separating the design into different areas of concern
- Also known as modules (or components)



KEY PRINCIPLES OF SOFTWARE ARCHITECTURES

- Separation of concerns
  - Each object and module should be in its own concern and context
- Single responsibility principle
  - Elements in our design must have a single purpose
- Principle of least knowledge
  - Components do not know about the internals of other components
  - Done through interfaces
- Don’t repeat yourself
  - Don't have multiple components with the same purpose
- Minimize upfront design
  - Try to design the minimum architecture so that developers can work
  - You will polish the next sections later

GENERAL GUIDELINES

- Use consistent patterns in each layer
  - If you decide to use MVC pattern in the presentation layer – stick to it
  - If you use CQRS in the application layer – stick to it
- Do not duplicate functionality
  - Do not have a caching component in all layers
- Prefer composition over inheritance
  - It is complicated to create huge hierarchies
  - Work with the Lead Developer here
- Establish a code convention
  - Define the code style and preferably automate it
  - Work with the Lead Developer here

LAYER GUIDELINES

- Separate areas of concern
  - Each layer should have one role only
- Define communication between layers
  - How does the presentation layer communicate with the business layer?
- Use abstraction to loosely couple layers
  - Use interfaces for every communication
  - Objects should not be tight to a particular platform or technology
- Don’t mix different types of components in a layer
- Use a consistent data format within a layer

COMPONENT GUIDELINES

- No component should rely on the internals of another
  - Components should be black boxes
- Do not mix roles in a single components
  - Web controllers should not have business logic
  - Business layer should not have HTTP concerns
- Define clear contracts for components
  - All public methods and properties should be scoped in advance
- Abstract system wide components away from other layers
  - As soon you have a component which is accessible in multiple layers
  - Put it in a system-wide area of concern

UNIFIED MODELLING LANGUAGE

WHAT IS UML?

- From Wikipedia:
  - "A general-purpose, developmental, modeling language that is intended to provide a standard way to visualize the design of a system"
- Main attributes:
  - Visual – it is easy to see the representation of the architecture
  - Abstract – it stays away from implementation details
  - Descriptive – shows the complete representation
  - Standard – UML is the world standard
  - Supports Code Generation – specific sections can be converted to code
  - Supports Reverse Engineering – create UML from code
- You should know it in case the client wants it
  - But can easily avoid it

UML DESIGN ELEMENTS

- Models
  - Container for the design
  - Are you looking at the entire system? A subsystem? A feature?
- Views
  - A way to look at the system
  - Is it an external view? Or the internal structure?
- Diagrams
  - Specific drawings that illustrate the architecture

UML MODEL TYPES

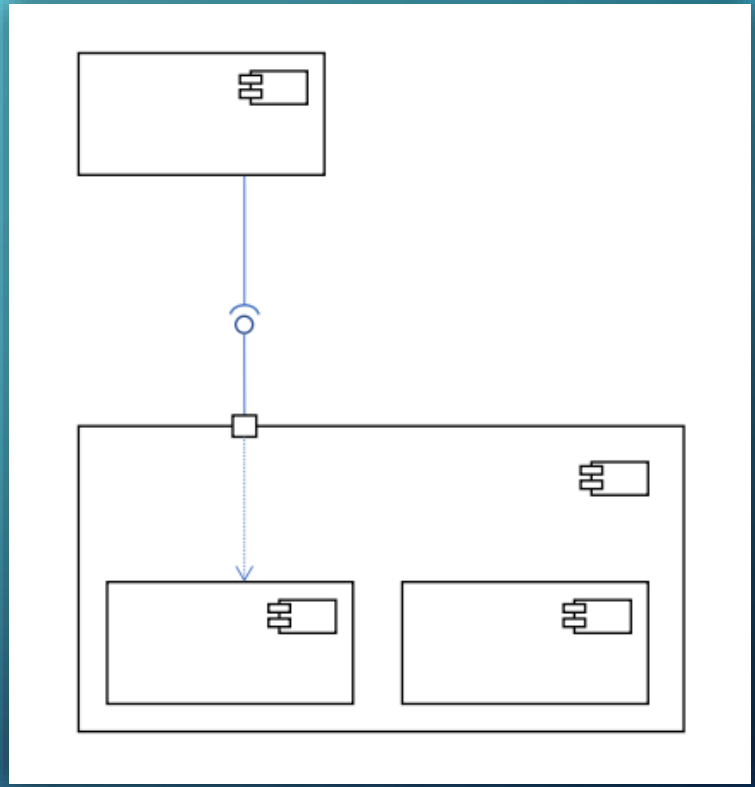
- Business System Model
  - Non-technical model
  - External View
    - Looking at the system as a black box
  - Internal View
    - A lot more details
- IT System Model
  - Specifically looks at technology
  - Static View (Structure View)
    - How the different elements fit together
  - Dynamic View (Behavioral View)
    - How these elements call each other

UML DIAGRAM TYPES

- Component Diagram
- Class Diagram
- Sequence Diagram
- State Diagram
- Activity Diagram
- Layer Diagram
- Use Case Diagram
- There are others…

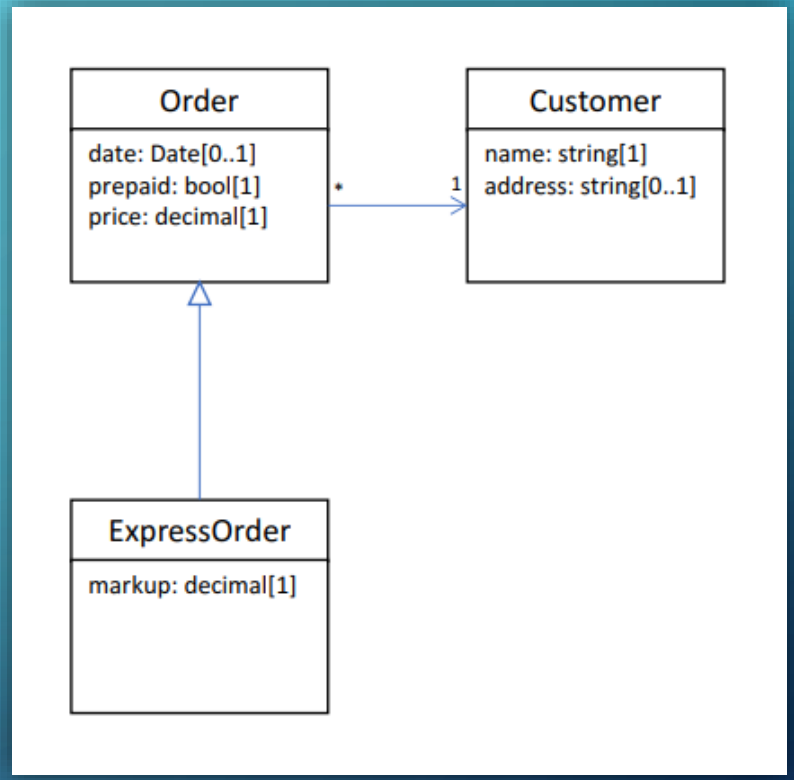
THE COMPONENT DIAGRAM

- Shows components
  - Modular building blocks
- Shows implemented and required interfaces
- Components can be nested
- If you decide to describe your architecture with a component diagram, you will have a very descriptive image



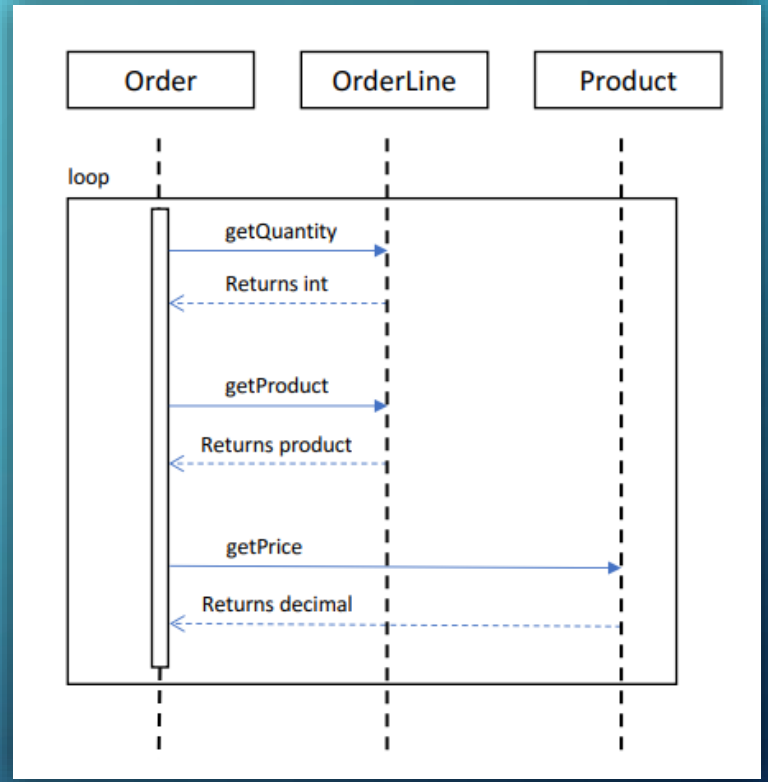
THE CLASS DIAGRAM

- Shows classes
- Shows methods and fields
- Shows associations, generalizations, and cardinality
- Quite detailed in terms of implementation



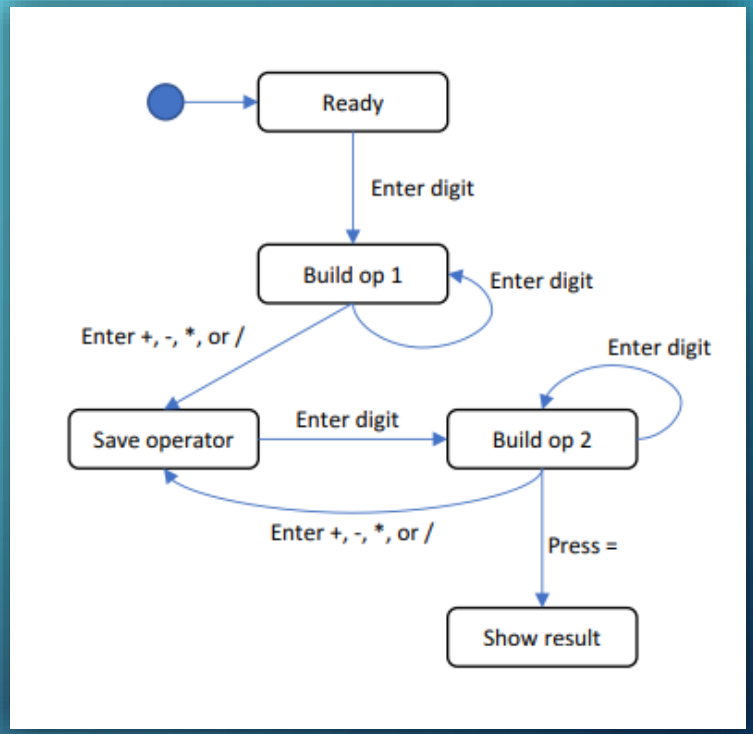
THE SEQUENCE DIAGRAM

- Shows call sequence
- Shows calling class, called method, and return data type
- Can depict loops



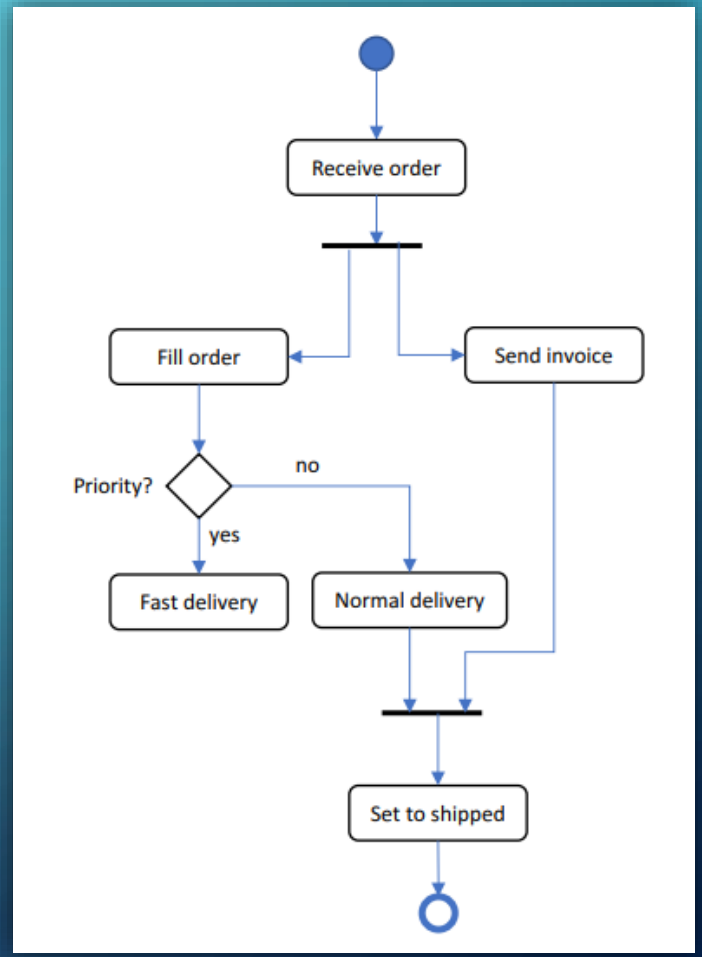
THE STATE DIAGRAM

- Shows states or activities
- Shows allowed transitions
- Can be nested
- Can depict internal activities



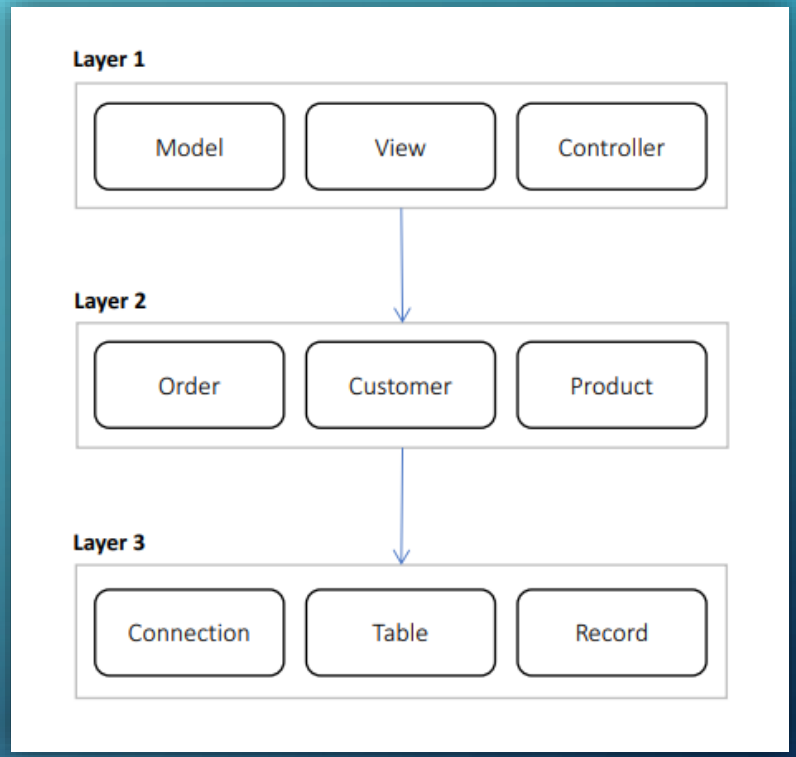
THE ACTIVITY DIAGRAM

- Shows process or workflow
  - Like the famous flow charts
- Can be nested
- Can show concurrent actions
- Can have swim lanes



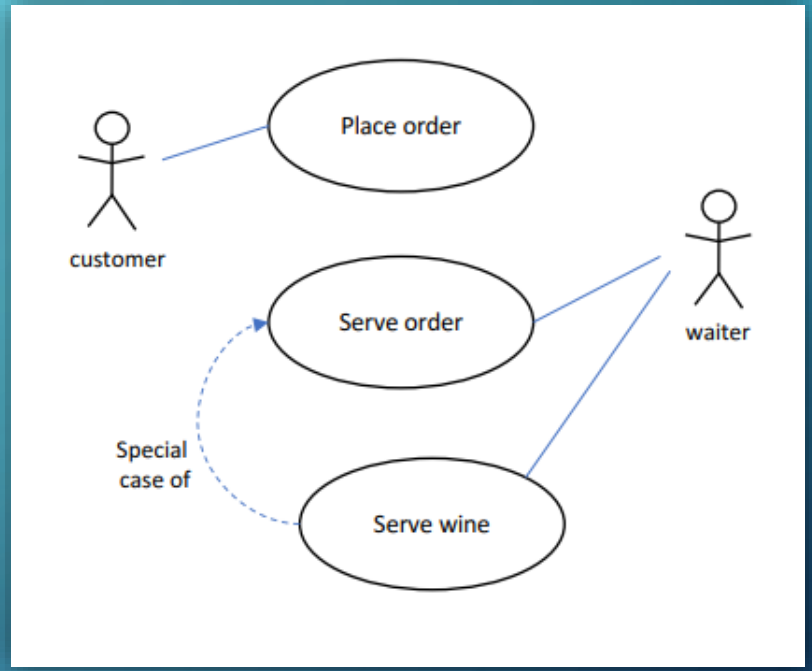
THE LAYER DIAGRAM

- Non-standard, invented by MS
- Shows areas of concern
- Shows references between areas
- Can be validated



THE USE CASE DIAGRAM

- Shows actors
- Shows use cases
- Binds actors to use cases
- Can depict generalizations



DESIGNING SOLUTION ARCHITECTURES

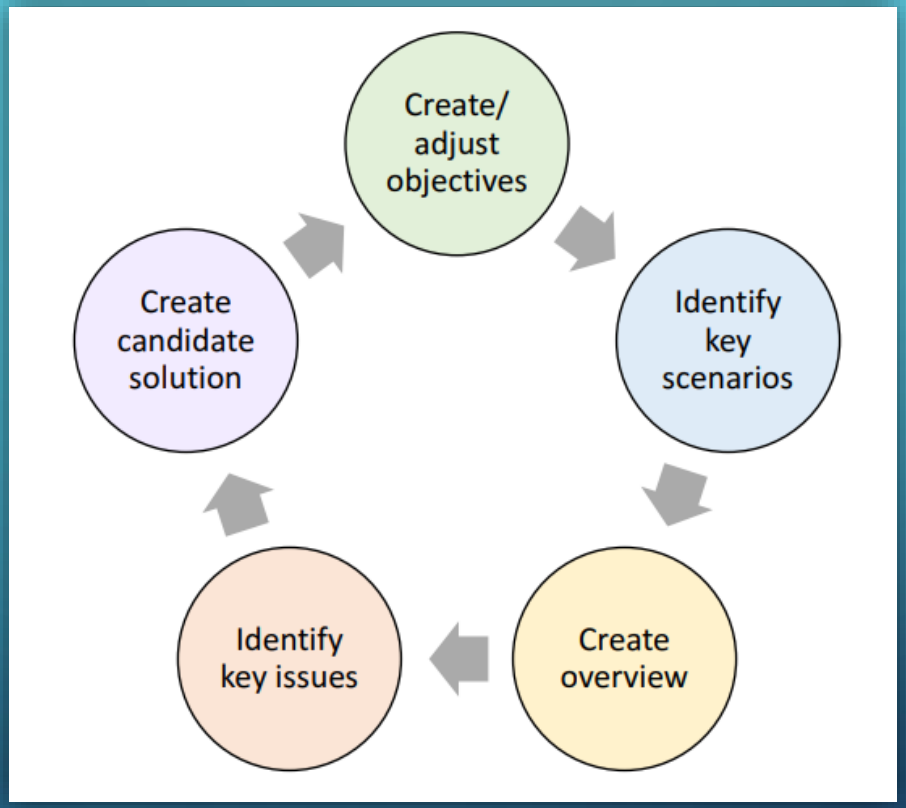
UML DIAGRAMS IN ARCHITECTURES

Solution Architecture Element	UML Diagram
Functional Requirement	Use Case Diagram
Structural Elements, Composition	Class Diagram Component Diagram
Structural Elements, Collaboration	Sequence Diagram Activity Diagram State Diagram
Areas Of Concern	Layer Diagram

UML DESIGN STRATEGIES

- UML as Sketch
  - Intended for brainstorming and general loose guidelines
- UML as Blueprint
  - Very detailed, you can write code based on the diagrams
  - Forward Engineering - use diagram to generate code
  - Reverse Engineering - build diagram from existing code
- UML as Validation
  - Validate implementation against diagram

THE PROCESS FOR DESIGNING ARCHITECTURES



CREATE OR ADJUST OBJECTIVES

- Identify scope of architecture
  - What are the high-level objectives and requirements?
  - New technologies? Server? Client? Proof of concept?
- Estimate time to spend
  - Do you have months? Do you have weeks?
- Identify audience
  - CEO? Developers? Functional Analyst?
- Identify technical, usage and deployment constraints
  - What are the available technologies?
  - How many people use the system simultaneously?
  - How many servers do you have? Cloud? On premise?
  - These requirements are super important!

IDENTIFY KEY SCENARIOS

- Key scenarios:
  - Significant unknown/risk – promo codes on scale
  - Significant use case – payments integration
  - Intersection of quality/function – clashing of these two attributes
  - Tradeoff between attributes – consider the trade offs
- Significant use cases:
  - Business-critical – part of the core business domain
  - High impact – very important for the end-users
- Create Use Case Diagrams

TWO TYPES OF REQUIREMENTS

- Functional
  - Business flows
  - What the user should do
  - User interfaces
  - And many more…
- Non-functional
  - Performance
  - Load
  - Data volume
  - Concurrent users
  - SLA
  - These are the most common ones

PERFORMANCE REQUIREMENTS

- "What is the required performance of the system?"
  - "SUPER FAST!"
- Always talk in numbers!
  - For example, if you have an end user – each task should complete in less than a second!
- Think about latency
  - How much time does it take to perform a single task?
  - How much time will it take to store a user in the database?
- Think about throughput
  - How many tasks can be performed for a given time unit?
  - How many users can be saved in the database in a minute?

PERFORMANCE NUMBERS EXAMPLE

- Let's say we have a task
  - Storing a user in database
- And its latency is 1 second
  - This is quite slow, but it is just a demonstrational number
- What is the throughput for 1 minute then?
  - In a well-designed system – more than 1000 users
  - In a badly designed system – around 60 users
- Both attributes are important!

LOAD REQUIREMENTS

- Load is quantity of work without crashing
  - In a Web API – how many concurrent requests could the server handle without crashing
- Difference with throughput:
  - Throughput – 100 requests/second
  - Load – 500 requests without crashing
- Users can tolerate a bit slow system
  - But they hate crashing ones!
- Best practice is to always plan for the most extreme cases!
  - For example – Black Friday in an e-commerce site!

DATA VOLUME REQUIREMENTS

- How much data the system will accumulate over time
- This requirement dictates:
  - The database type
  - Designing queries in the database
  - Storage planning
- Two aspects:
  - Data required on "day one"
    - For example – initially we need 1 GB of data
  - Data growth
    - For example – our database grows annually with 2 TB of data

CONCURRENT USERS REQUIREMENTS

- How many users will use the system simultaneously
- It is different than the load requirements
  - Concurrent users have "dead times"
  - They read the page, watch something on it, but do not make requests
  - Load requirements consider the actual requests
- The rule of thumb is to take the load requirements
  - And multiple them by 10!
  - But this depends on the system type

SERVICE LEVEL AGREEMENT REQUIREMENTS

- What is the required uptime for the system in percentage
  - For example, 99.99% uptime
  - This is translated to ~1 hour of downtime in a year
  - It would be quite impressive!
- It is the solution's architect to manage the clients' expectations
  - If the expected uptime is 99.999%
  - Just tell them that we will need at least five different data centers
    - On independent continents
    - With variety of power supplies
    - And multiple Internet providers
  - This answer usually brings them down to Earth ☺

WHO DEFINES THESE REQUIREMENTS?

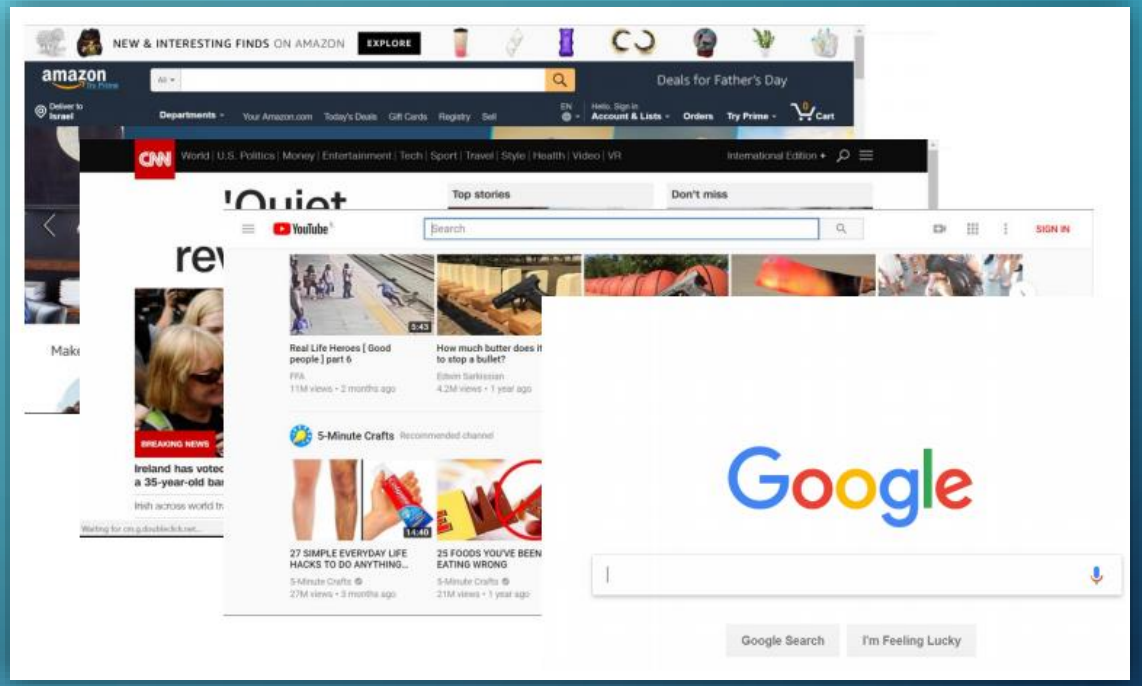
- The Functional Analysts and the CEO usually provide a well documented business and functional requirements of the system
- But who defines the non-functional ones?
  - It is the Solution's Architect job to frame them
    - By starting discussions
    - And asking the right questions
    - And keeping the expectations realistic and meaningful
    - There is no need to fight for every millisecond
    - The users will not notice that
- Never design a system without the non-functional requirements!

CREATE APPLICATION OVERVIEW

- Determine application type
  - Bot? Console? Service? Web? Desktop? Maybe Serverless?
- Identify deployment constraints
  - Where are you going do deploy the application?
- Identify architecture pattern
  - Layered? Component? Microservices?
- Determine technologies
  - What technologies are available? Libraries? Third-party tools?
  - A lot of factors play here… Choose wisely!
- Create a Layer Diagram

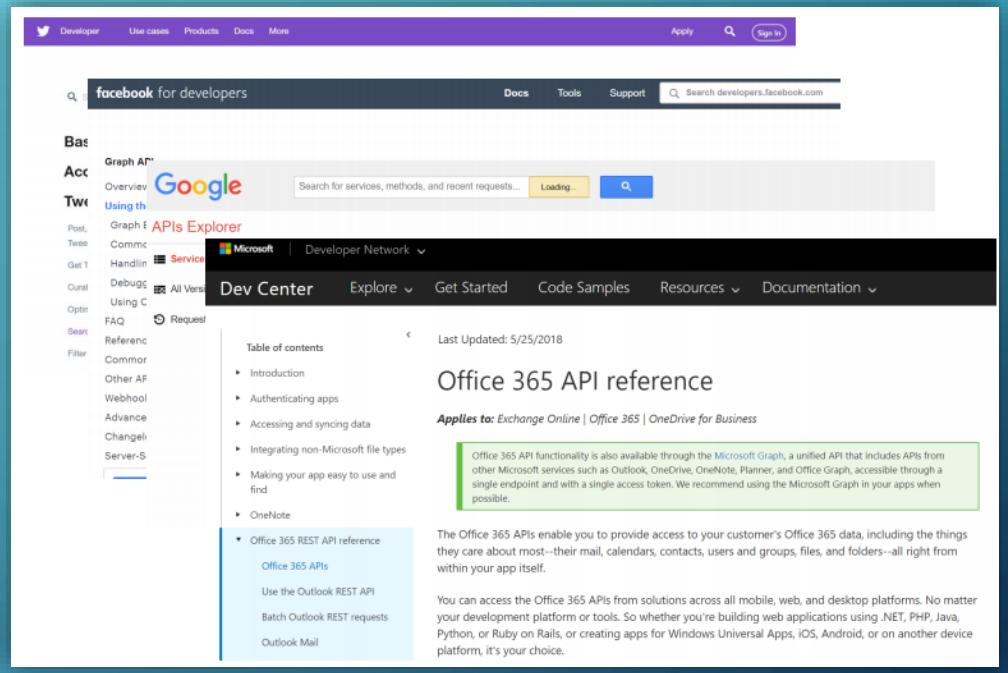
APPLICATION TYPES – WEB APPLICATION

- Best suited for:
  - User interface
  - User initiated actions
  - Large scale
  - Short, focused actions
- Request-Response based



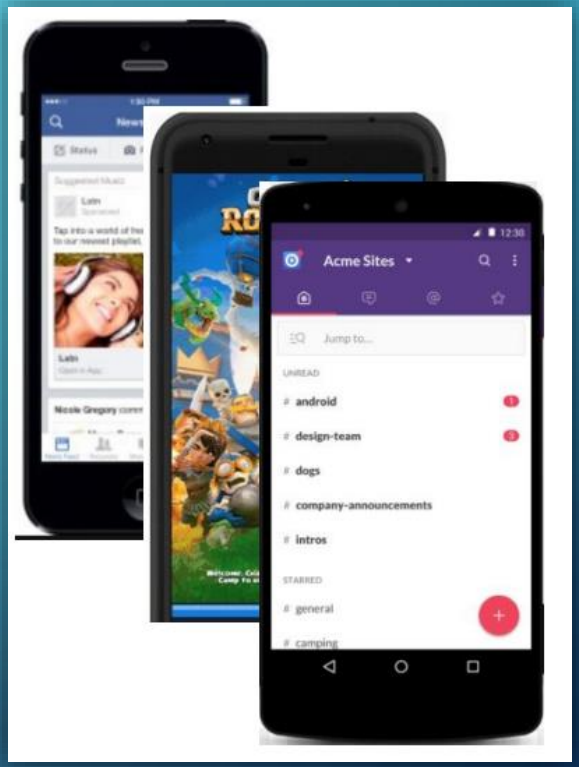
APPLICATION TYPES – WEB API

- Best suited for:
  - Data retrieval and storage
  - Client initiated actions
  - Large scale
  - Short, focused actions
- Usually REST based
- Returns data, not HTML
- Combination of:
  - URL (https://www.mysite.com/api/orders)
  - Parameters (date=10/10/2017)
  - HTTP Verb (GET)



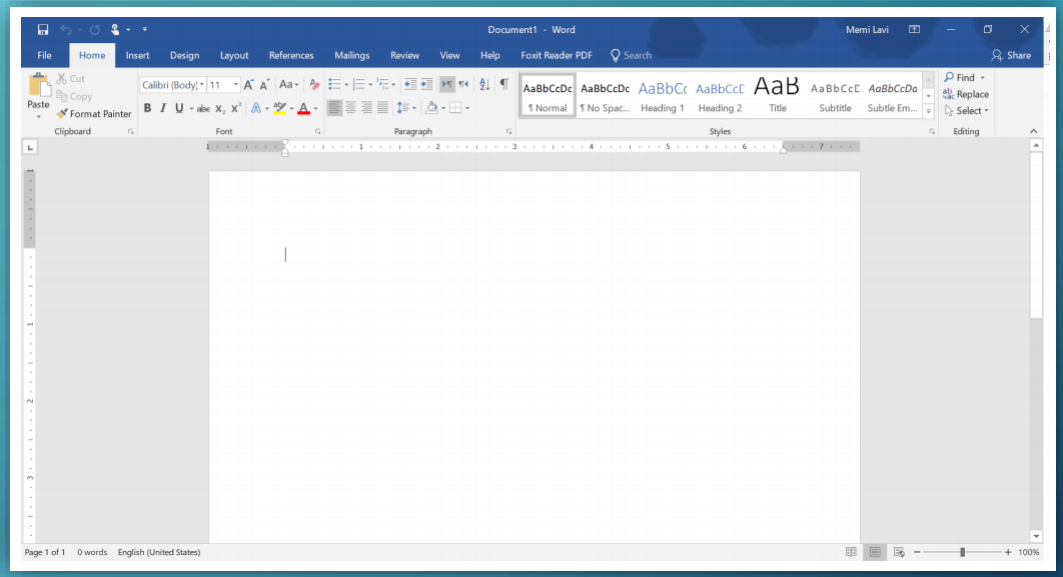
APPLICATION TYPES – MOBILE APPLICATION

- Best suited for:
  - Well, mobile devices ☺
  - User interaction (games, social apps)
  - Front end for Web API
  - Location & camera-based systems
- Usually work with a Web API



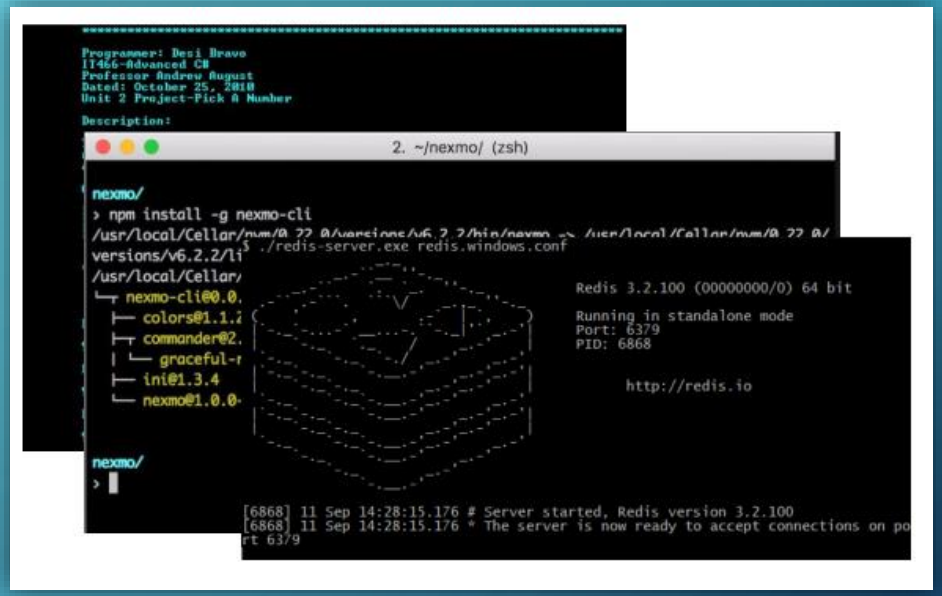
APPLICATION TYPES – DESKTOP APPLICATION

- Best suited for:
  - User centric actions
  - Gaming
- Has all its resources on the local PC
- Might connect to the web
- Great UI



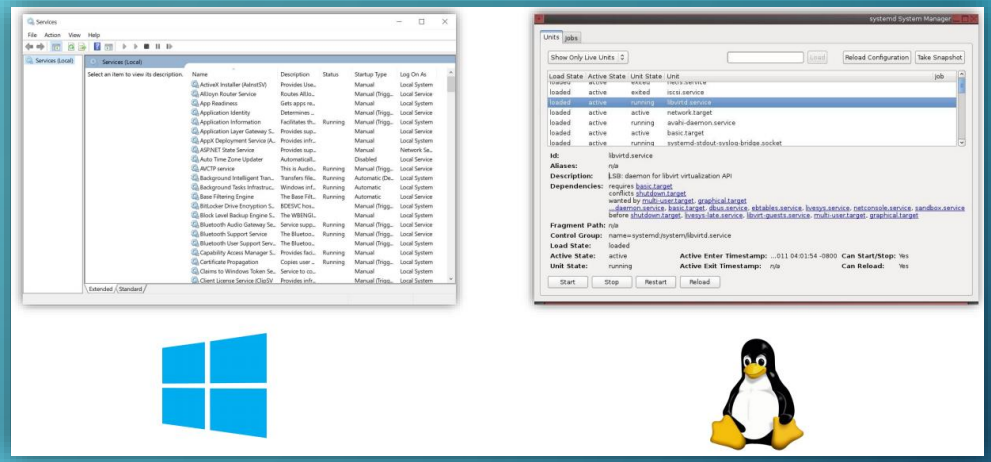
APPLICATION TYPES – CONSOLE APPLICATION

- Best suited for:
  - Long-running processes
  - Short actions by trained power-users
- No fancy UI
- Require technical knowledge
- Limited interaction
- Long or short-running Processes



APPLICATION TYPES – SERVICE

- Best suited for:
  - Long-running processes
- No UI at all
- Managed by the OS service manager



SELECTING TECHNOLOGY STACK

- This is a super important decision
  - Mostly because it is almost irreversible
  - And developers get emotional to their favorite tools
- The decision must be:
  - Made with a clear mind, heavily documented and based on a group effort
- Main considerations besides performing the task:
  - Community – check Stack Overflow
  - Popularity – check Google Trends
  - Current developer skills – unknown technologies produce delay and low quality
  - Deadline – advanced technologies take more time
  - Support – developers should develop, be careful with shiny new tools
  - Products – use external and existing tools but always estimate the cost

GENERAL TIPS AND TRICKS

- Do not choose on recommendation
  - Or marketing tricks
- Always do a pros/cons list
  - Which type system – static or dynamic?
  - What is our infrastructure?
  - Is the community using the tool?
  - Is performance critical?
  - Do we need to analyze the learning curve?
  - Can you easily hire a new team member?
- Stay rational and don't argue with your colleagues!
  - Use the right tool for the job!
  - And not vice-versa!

POPULAR TECHNOLOGIES

- Web applications – back-end
  - Mature - .NET, Java, PHP
  - Newer - Node.js, Python, .NET Core, Go
  - Serverless – Amazon Lambda, Azure Functions
- Web applications – front-end
  - HTML, CSS & JavaScript (or maybe Web Assembly, if you are adventurous)
  - Which JavaScript framework? Angular? React? Vue? Svelte?
- Mobile applications
  - Native? Hybrid? Cross-Platform?
  - The battle is development time versus capabilities
- Desktop applications
  - WinForms, WPF, UWP

BACK-END AND SERVICE TECHNOLOGIES

	App Types	Type System	Cross Platform	Community	Performance	Learning Curve
.NET	All	Static	No	Large	OK	Long
.NET Core	Web Apps, Web API, Console, Service	Static	Yes	Medium and growing rapidly	Great	Long
Java	All	Static	Yes	Huge	OK	Long
node.js	Web Apps, Web API	Dynamic	Yes	Large	Great	Medium
PHP	Web Apps, Web API	Dynamic	Yes	Large	OK -	Medium
Python	All	Dynamic	Yes	Huge	OK -	Short

MOBILE TECHNOLOGIES

	Native	Hybrid	Cross Platform
Development Language & IDE	**iOS** – Objective-C or Swift, with X-Code & iOS SDK **Android** – Java with Android Studio & Android SDK	Thin wrapper around HTML, JavaScript, CSS	Xamarin (C#, Visual Studio) React Native (JavaScript)
Access to Phone's Features	Full control, no limits	Very limited	Catch-up with latest versions
User Experience	Exceptional	Inferior	Good, with limitations

Keep an eye on PWA!

DESKTOP TECHNOLOGIES

	WinForms	WPF	UWP
Founded	2001	2006	2015
UI Flexibility	Limited	Unlimited	Unlimited, but runs in a sandbox
Learning Curve	Short	Long	Long
Runs on…	PCs	PCs	PCs, XBOX, IOT

DATABASE TECHNOLOGIES

- Key-value
  - Useful for cache, publish/subscribe, leaderboards
  - Not your primary database
  - Redis, Memcached
- Wide column
  - Each key has multiple columns without a schema
  - Cannot do joins and scales easily
  - Useful for time-series (IoT), historical records, high-write/low-read scenarios
  - Not your primary database
  - Cassandra, HBase

DATABASE TECHNOLOGIES

- Document
  - Each document is a container for key-value pairs
  - Fields can be indexed
  - Columns can be collection – query relational data without joins
  - Reading data is usually super fast, but writing data tends to be more complex
  - General purpose – applications, games, IoT, content-management
  - If you have unstructured data – a document database is a great start
  - No standard language for queries
  - Not suitable for huge graphs of data – social networks, for example
  - MongoDB, Firestore, DynamoDB, CouchDB

DATABASE TECHNOLOGIES

- Relational
  - Tables are connected through relationships
  - Requires a schema
  - ACID compliant – atomicity, consistency, isolation, durability
  - Good for stable data
  - More difficult to scale
  - Supports SQL – a query language
  - General purpose - perfect for most applications which does not have unstructured data
  - Some databases support JSON columns
  - MySQL, PostgreSQL, SQL Server
  - Modern databases like CockroachDB – designed for scale

DATABASE TECHNOLOGIES

- Graph
  - Data is represented as nodes
  - Relationships between the nodes are edges
  - Instead of many-to-many tables, you have direct connections
  - Languages like SQL
  - Better performance than relational databases when datasets are large
  - Useful for – recommendation engines and well… graphs
  - Neo4j, CosmosDB

DATABASE TECHNOLOGIES

- Search engine
  - Full-text search engine
  - Like document databases but under the hood all the text is indexed
  - Works like an index at the back of a book
  - Can easily rank results
  - Useful for search engines and typeahead scenarios
  - Elasticsearch, Algolia, MeiliSearch

DATABASE TECHNOLOGIES

- Multi-model
  - You do not think about data modeling, schemas, shards, replications
  - You describe how you want to use the data with GraphQL
  - It chooses the database type behind the scenes
  - ACID compliant, extremely fast and you do not care about provisioning infrastructure
  - Useful for… everything?
  - FaunaDB

IDENTIFY KEY ISSUES

- Quality attributes:
  - System, run-time, design, user
  - Performance, stability, manageability, configurability, etc.
- System wide concerns:
  - Authentication & authorization
  - Caching
  - Communication
  - Configuration management
  - Logging & exception management
  - Validation

CREATE CANDIDATE SOLUTION

- Create a Baseline architecture
- Create a Candidate architecture
- Develop Architectural Spikes
  - Proof of concept projects to validate unknown concepts
- Create Activity, Sequence, State, and Component Diagrams
- Create Class Diagram if needed
  - They are too low-level
  - Document only the most important classes
  - Everything else should be done by the Lead Developer

DURING EACH CYCLE

- Do not introduce new risk
  - Do not introduce untested technologies and concepts
- Mitigate more risk than baseline
  - Each cycle should reduce the risk
- Meet additional requirements
  - Cover more and more requirements until you meet them all
- Enable more key scenarios
- Address more key issues
- Start communicating architecture when you exceed 50% coverage
  - If you have the feeling you are half done, invite the developers
  - You don't want to be the bottleneck

A SAMPLE PROJECT

LEARNING SYSTEM

- Here are our business requirements from the Functional Analyst:

*A **student** visits the solution and **logs in**. He or she is presented with a **list of courses**. When a student clicks a course, he or she is taken directly to the last visited lecture in that course. The lecture detail page has 3 panels and shows the **curriculum** on the left, the **lecture contents** in the middle, and a **Q & A panel** on the right. The student can use the curriculum to navigate to different lectures and **submit questions** to the instructor in the Q & A panel.*

*An **instructor** visits the solution and logs in. He or she is presented with a **list of courses**. When the instructor clicks on a course, he or she is taken directly to a **course management** page. The page shows all **questions** in the course, with the unanswered questions highlighted. By clicking on a question, the instructor navigates to a new page where he or she can **answer the question**.*

LEARNING SYSTEM

- And here are our business requirements from the CTO:

> *Lectures need to load in 1 second or less. **Performance is key**, our USP is to be the fastest platform in the business, and our students and instructors will abandon our platform if we are too slow. We also cannot **afford to be offline**. Every hour we are offline would cost our business thousands of dollars in lost revenue. And we want to scale to **millions of students**, the platform must be able to accommodate for that with ease.*

- Basically, our key attributes are:
  - Performance
  - Availability
  - Scalability

INITIAL DESIGN

- We have not learned about architecture patterns yet
  - So, it's a bit premature to start creating a detailed design
  - But we know about the process and can get started with our initial diagrams
- Our tasks for now are in the first 3 steps of the design process:

  1a. What are our objectives? What are our scopes?

  1b. Who is the key audience? Any constraints?

  2a. What are the key business scenarios? Create a Use Case diagram.

  2b. Create an Activity diagrams for the student and the instructor.

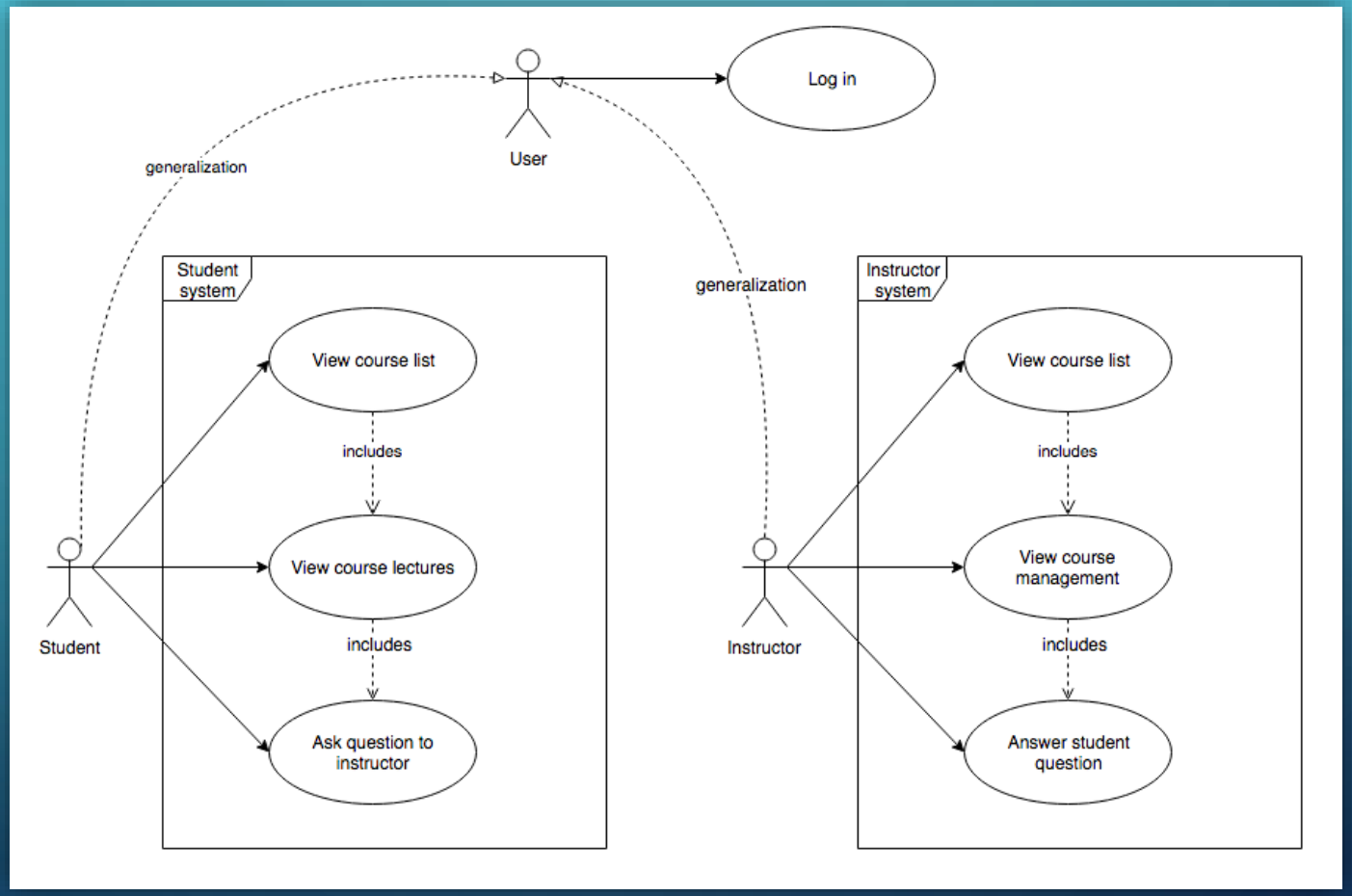
  3a. Think about the architectural overview. What is the application type?

  3b. Decide the technology stack. Back-end? Front-end?
- You can use https://draw.io or https://lucid.co/product/lucidchart for the diagrams

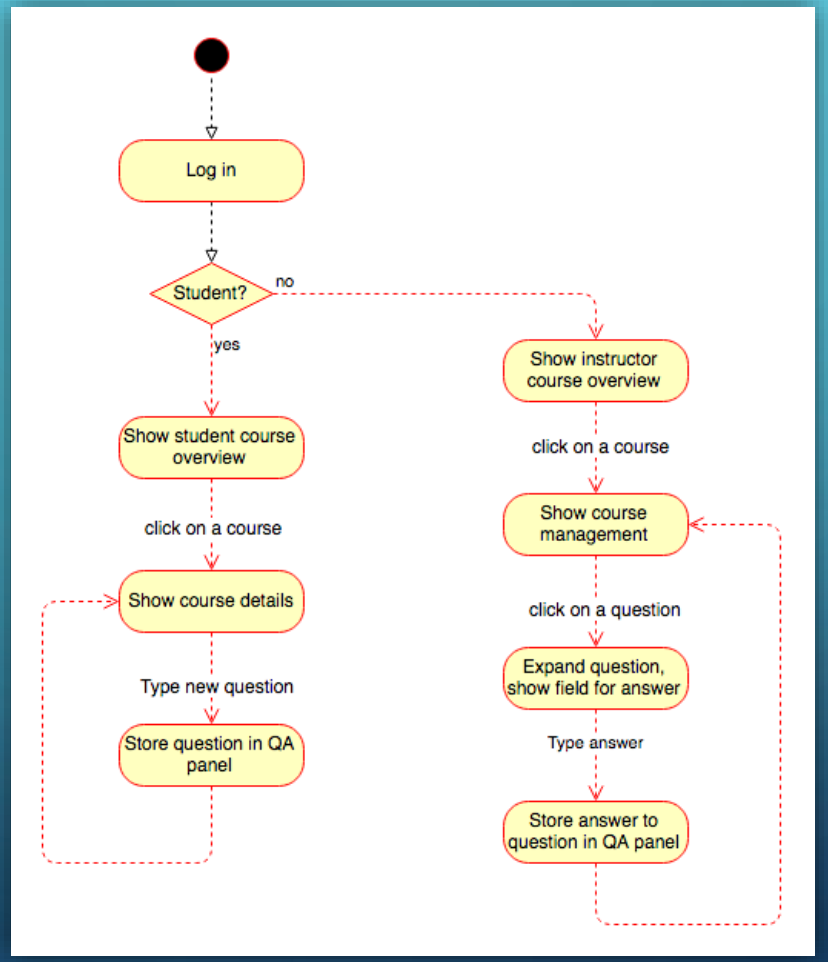
SCOPE AND OBJECTIVES

- The scope of this architecture contains:
  - Login screen for both student and instructor
  - Student interface for listing courses
  - Watching course lectures
  - Q & A panel for submitting questions
  - Instructor interface for listing courses
  - Course management page
  - Q & A panel for answering student questions
- Time schedule – 1 week
- Audience – CTO and Developers
  - We can easily include technical stuff

KEY SCENARIOS USE CASE DIAGRAM EXAMPLE



KEY SCENARIOS ACTIVITY DIAGRAM EXAMPLE



TECHNOLOGY CONSTRAINTS

- We are building the solution from scratch
  - We choose technologies based on our and the developer skills
  - There is no need to introduce an unfamiliar stack – it will increase the expenses of the project
  - If you are still building the team – you can experiment a bit more
- I am proficient with .NET and JavaScript
  - And my team is consisting of very skilled .NET developers
  - My technology stack will be Microsoft oriented
- We want millions of students and huge availability
  - On premises servers will require huge maintainability
  - We are going to use a public cloud
  - Azure is the perfect choice for Microsoft technology
  - But consider the other options as well – they may be cheaper

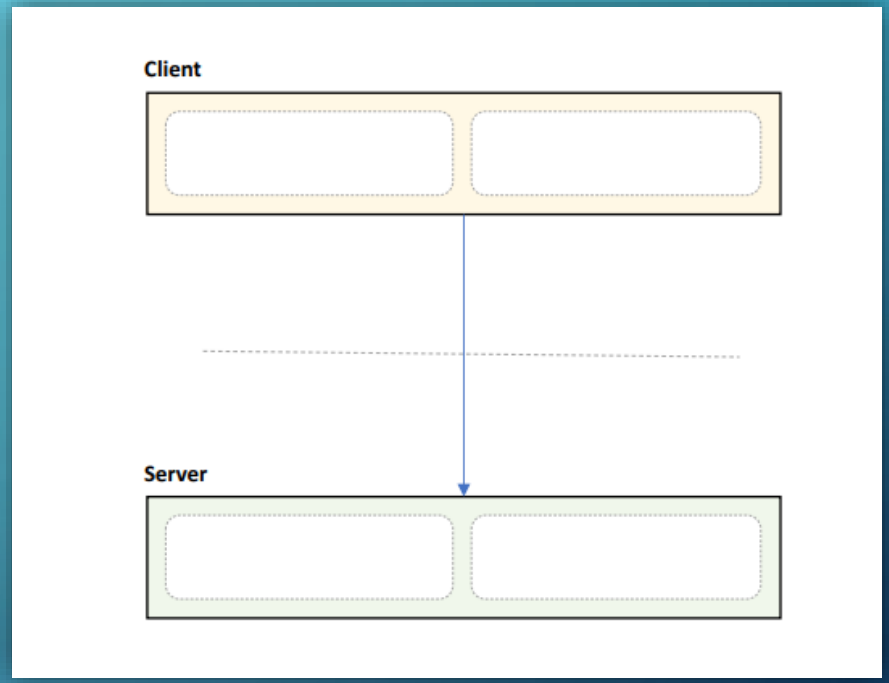
TECHNOLOGY OPTIONS

- Back-end technology options:
  - ASP.NET Core REST API – requires heavy server-side architecting
  - Serverless with Azure Functions – less architecture burden but less overall control
- Front-end technology options:
  - ASP.NET Core MVC
    - Server-side rendering is not very suitable for interactive applications
  - ASP.NET Core Razor Pages
    - Less overhead in terms of the client
  - Blazor
    - Way too new and experimental but the developers may want the bleeding edge
  - React or Vue
    - Depends on the knowledge of the developers

ARCHITECTURE DESIGN PATTERNS

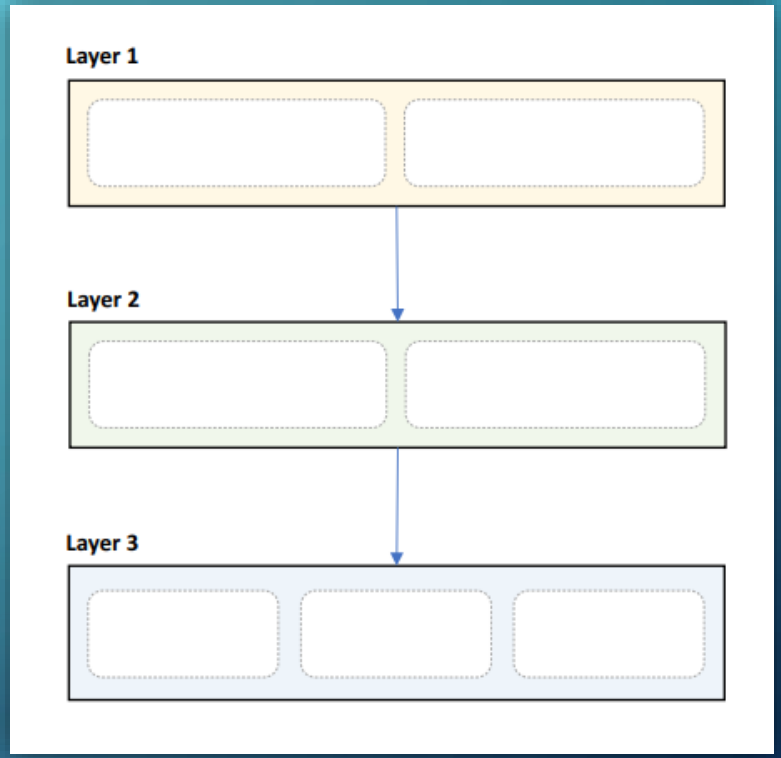
CLIENT/SERVER PATTERN – LAYERED

- Distinct client and server
- Separated by network
- Communication protocol
- Many clients, one server
- Pros:
  - Secure & Simple
  - Centralized Control
  - Easy to manage
- Cons:
  - Requires network, difficult to scale
  - Single point of failure



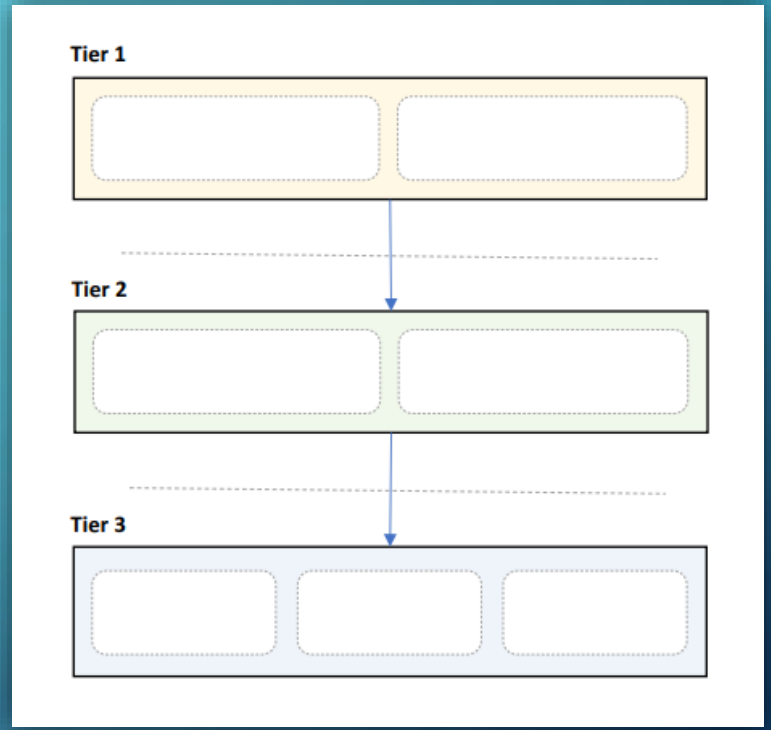
LAYERED PATTERN – LAYERED

- Strict areas of concern
- Layers may only communicate with peers above/below
- Pros:
  - High abstraction & High isolation
  - Structured communication
  - Easy to scale out
- Cons:
  - Deep call chains
  - Can hide complexity
  - May harm performance
  - Lowest layer must cover all use cases



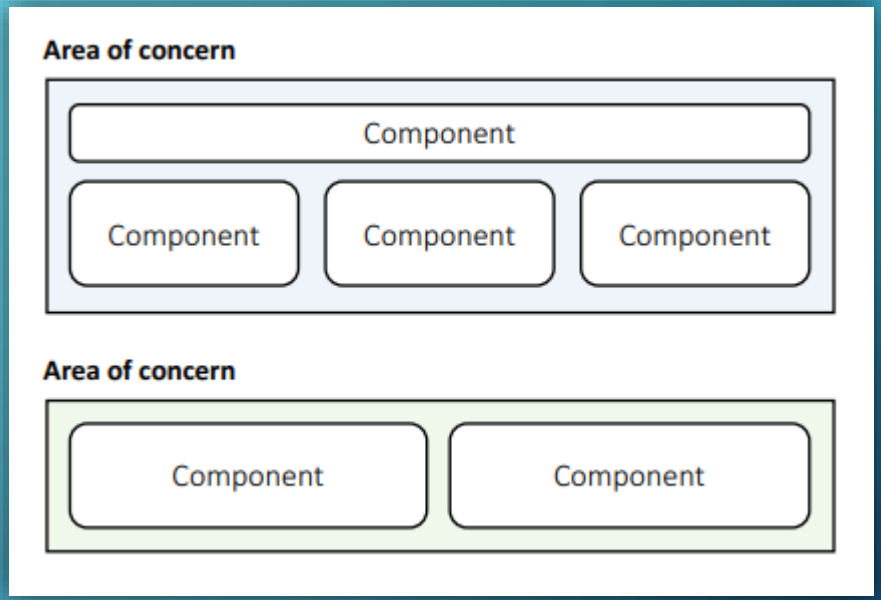
N-TIER PATTERN – LAYERED

- Layers deployed to servers
- Usually - 3 tiers, always start with them
- Communication between layers uses network
- Pros:
  - High abstraction & High isolation
  - Structured communication
  - Easy to scale out
- Cons:
  - Network = point of failure
  - Network may be slow
  - Coarse interfaces
  - Hard to debug



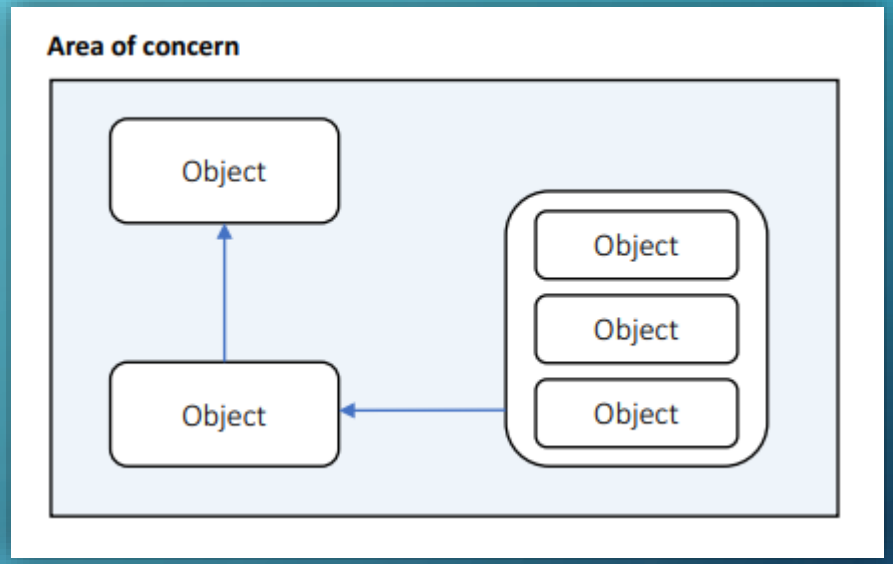
COMPONENT-BASED PATTERN - STRUCTURAL

- Components are modular building blocks of software
- Grouped into areas of concern
- Clearly described interfaces
- Containers provide additional services
- Pros:
  - Easy deployment & Allows 3rd party tools
  - Promotes modularity & Few unanticipated interactions
- Cons:
  - Coarse building blocks & Can be expensive
  - Initialization may be slow & Harder to develop & maintain



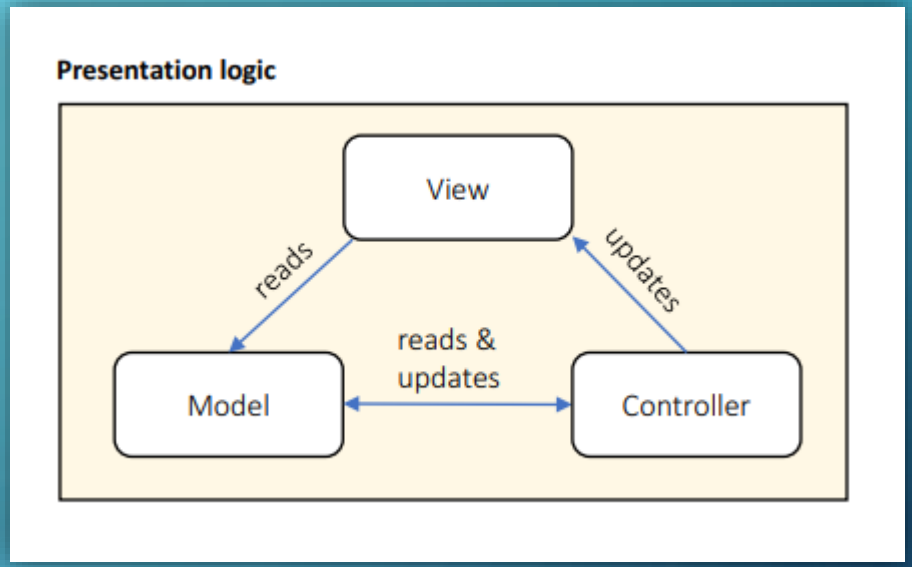
OBJECT-ORIENTED PATTERN - STRUCTURAL

- Uses classes
- Grouped into areas of concern
- Describes public members
- Uses inheritance, composition, aggregation, and associations
- Pros:
  - Easy to understand & Promotes reuse
  - Easy to test & debug & Highly cohesive
- Cons:
  - Inheritance hard to get right & Useful only if you do not have a Lead Developer
  - Many unanticipated communications & Too detailed



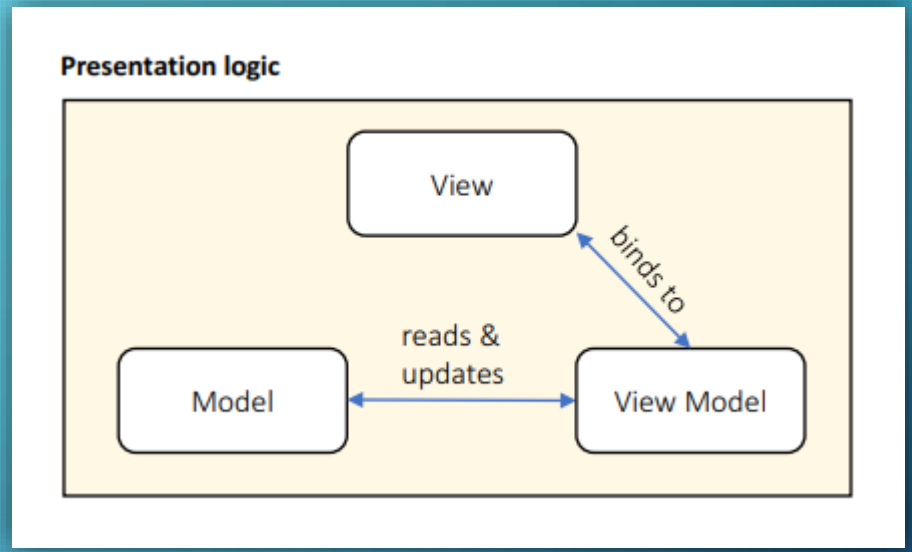
MVC PATTERN - PRESENTATION

- Designed for presentation layers
- View handles output
- Model handles data
- Controller handles interaction
- Pros:
  - Strict separation of concerns
  - Scales well
- Cons:
  - High overhead
  - Scattered code
  - Hard to data-bind



MVVM PATTERN - PRESENTATION

- Derived from MVC pattern
- View handles output
- Model handles data
- View Model is binding source
- Pros:
  - Strict separation of concerns & Scales well
  - Easy to data-bind
- Cons:
  - High overhead
  - Scattered code
  - Controller not separated

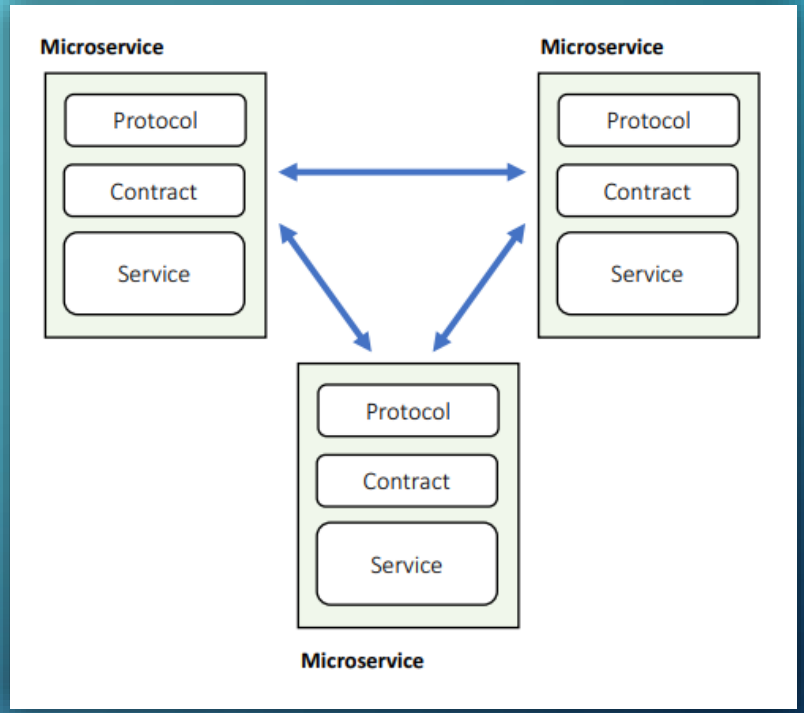


SERVICE-ORIENTED PATTERN - SERVICE

- Discrete business services
- Use network to communicate
- HTTP, XML, SOAP, Binary…
- Pros:
  - Business domain alignment & High abstraction
  - Discoverable, resilient, Allows 3rd party libraries & Cross-platform
- Cons:
  - Clients must handle slow, offline network
  - Coarse interfaces
  - May harm performance
  - Security issues

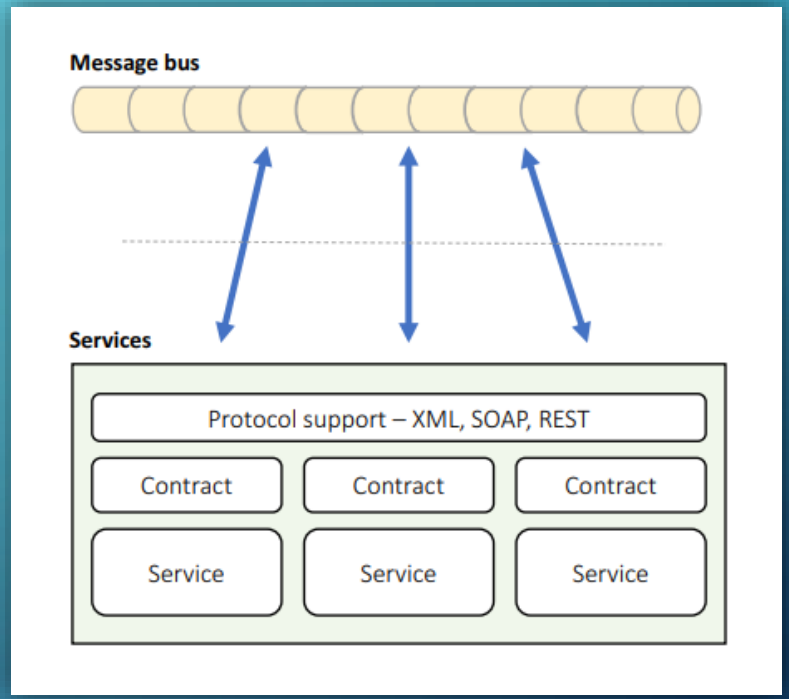
MICROSERVICE PATTERN - SERVICE

- Services calling services
- Can use fast private network and RPC
- Deployed on multiple servers
- Pros:
  - Modular & Reduced abstraction
  - Discoverable, resilient & Less coarse interfaces
  - Can use fast network
- Cons:
  - Must cope with slow, offline network
  - More unanticipated communications
  - Hard to do transactions & Hard to test, debug, deploy



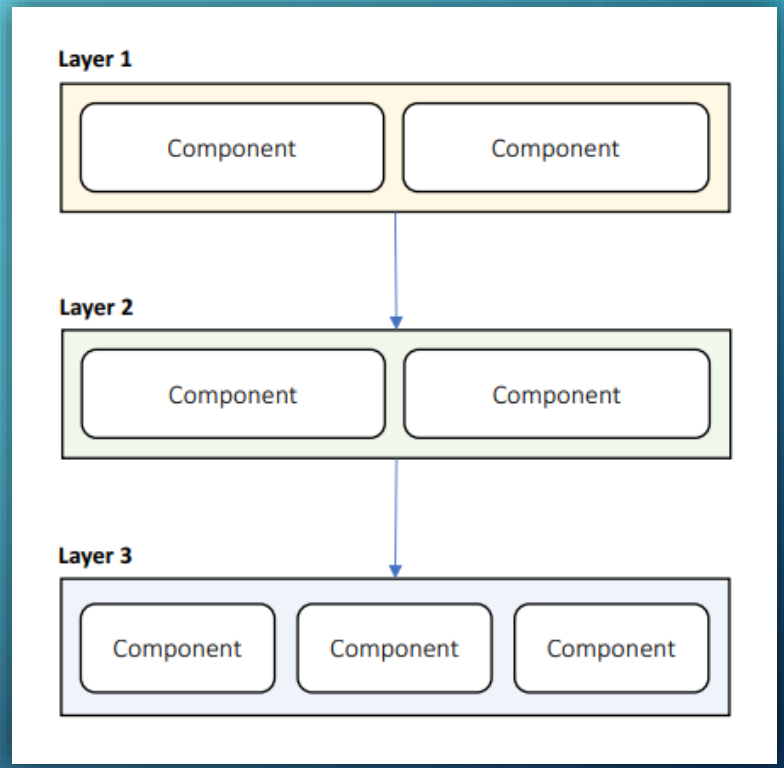
MESSAGE BUS PATTERN - SERVICE

- Services connected to shared data bus
- Uses messages for communication
- Supports discovery, failover
- Pros:
  - Easy to extend & Simple communication
  - Very flexible & Easy to scale
  - Easy discovery, failover
- Cons:
  - Bus = single point of failure
  - Coarse communications
  - Can be slow & Hard to test, debug



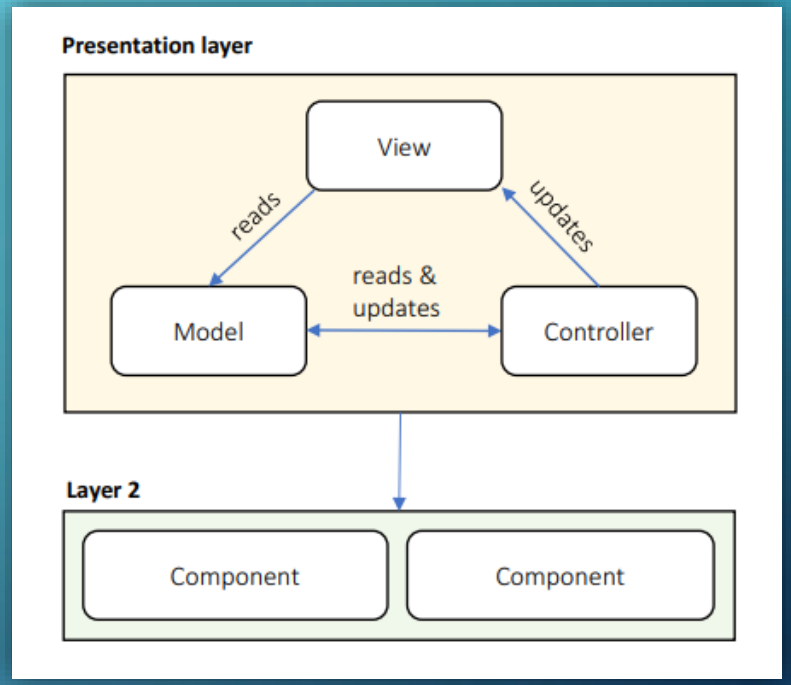
COMPONENTS IN LAYERS - HYBRID

- Layers containing components
- Benefits of components
- With structured communication, isolation, easy scaling & testing



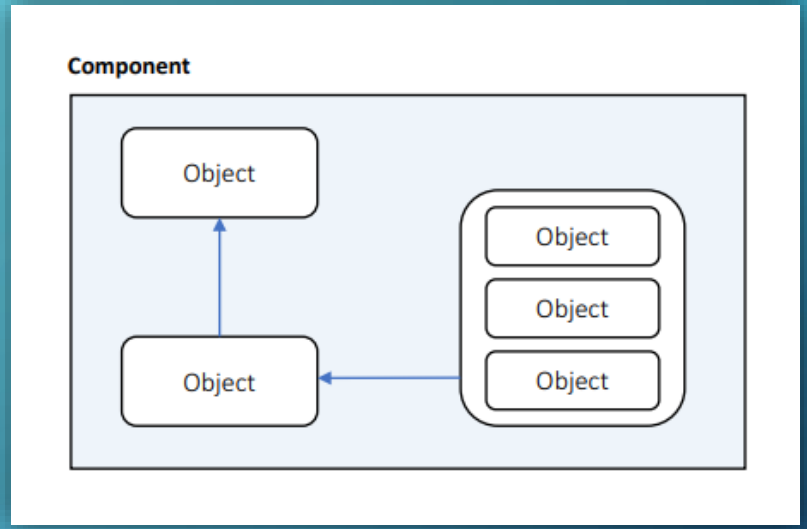
MVC IN PRESENTATION LAYER - HYBRID

- Presentation layer with MVC
- Benefits of layers
- With separation of concerns, presentation code scalability



OBJECTS IN COMPONENTS - HYBRID

- Components contain objects
- Benefits of components
- With cohesion & extensibility
- You are probably using this pattern already



MIXING OF PATTERNS

- A software architecture is just a mixture of patterns
- You can safely nest architectural design pattern inside of each other
- For example:
  - Microservices with components inside them
  - Microservices with a layered pattern inside them
- Create your own hybrid patterns and use them in your design
- If the pattern works for you, implement it!

QUESTIONS FOR THE LEARNING SYSTEM

- We already chosen an application type and the technology stack
- We need to pick architectural pattern and draw a diagram

1a. Choose an architectural design pattern. A great place to start is the layered pattern.

1b. Create a diagram with presentation, service, business, data and systemwide layers.

1c. Choose a pattern for the presentation layer. MVC? MVVM? Components?

2a. Consider authentication, caching, communication, configuration, logging, exceptions and validation.

2b. Consider all third-party tools to help you with these cross-cutting concerns.

3a. This is our first baseline architecture. Think about the components in the layers.

3b. Create diagrams for each layer and describe their public interfaces.

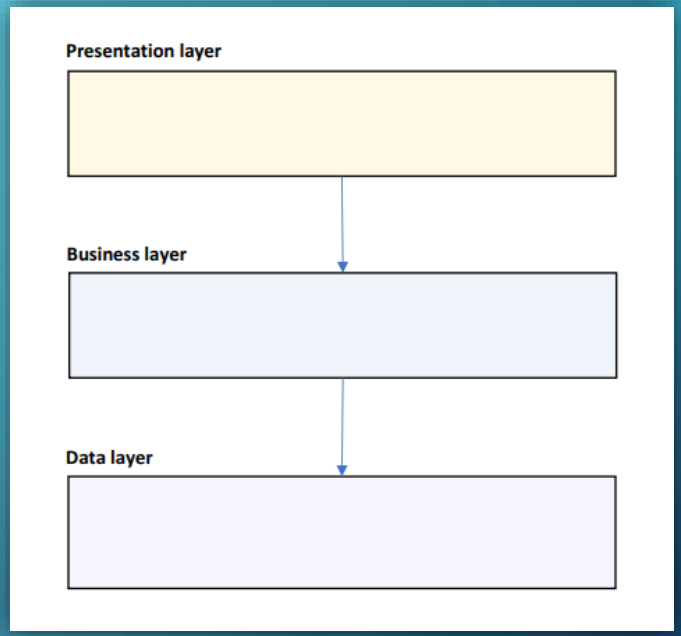
3c. Make sure you keep it simple at this stage. You can go into more detail in later cycles.

4. Design the business domain entities. Create a class diagram and define their public properties.

CHOOSING THE RIGHT PATTERNS

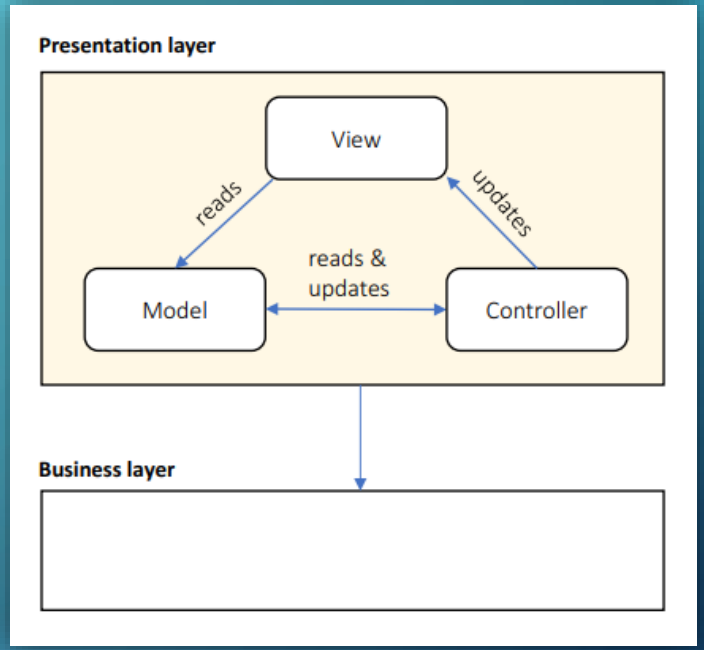
THE LAYERED PATTERN

- Good starting point
- Create 3 layers:
  - UI / Presentation – visualization concerns
  - Business – business logic and domain
  - Data – data access layer
- If you expect a long-term project, you may choose Domain-Driven Design with Clean Architecture
  - Presentation – visualization concerns
  - Application – business logic
  - Infrastructure – infrastructure details
  - Domain – business entities
- Add a Service layer if you plan to expose an API



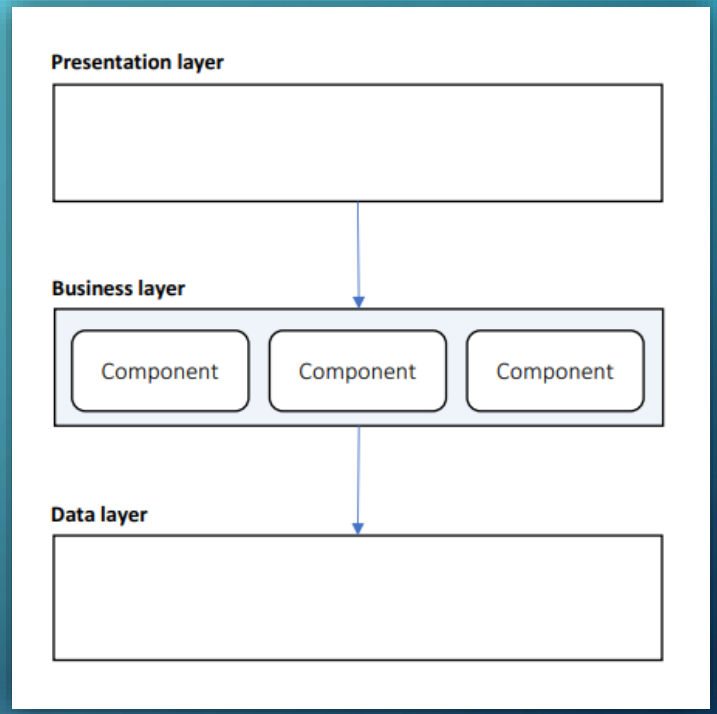
THE PRESENTATION LAYER

- Use Components for:
  - Containers
  - Reuse
  - 3rd parties
  - Declarative rendering
    - Like WordPress – with widgets
    - Or with visual designer
  - Lots of dynamic interactions
- Use MVC for lots of UI pages
  - Or MVVM if the technology support data binding
- Remove this layer, if you designing an API
  - Without any UI



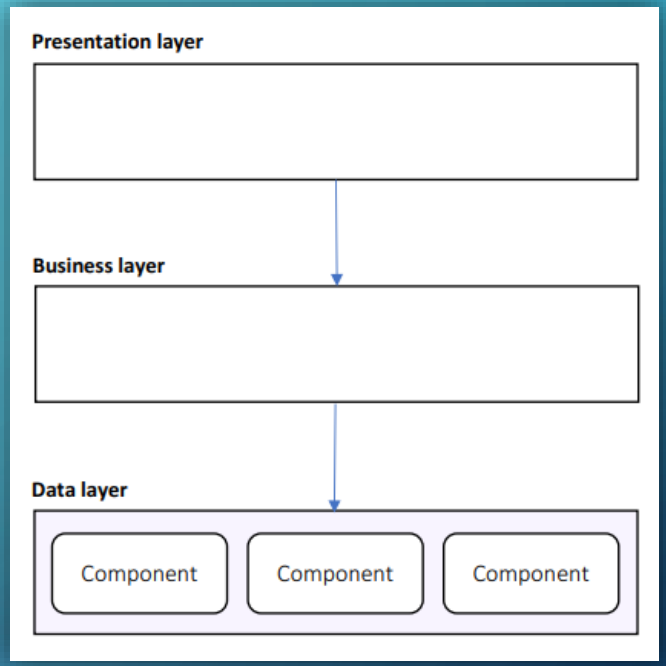
THE BUSINESS LAYER

- Use Components:
  - Modular functionality
  - Plugin support
  - Business entity abstraction
    - For each entity – create a component
  - Declarative configuration:
    - Visual workflows – like a flow chart for the business rules
    - Business rules – collection of if/then/else rules
- If you don't need the above, use an object-oriented architecture



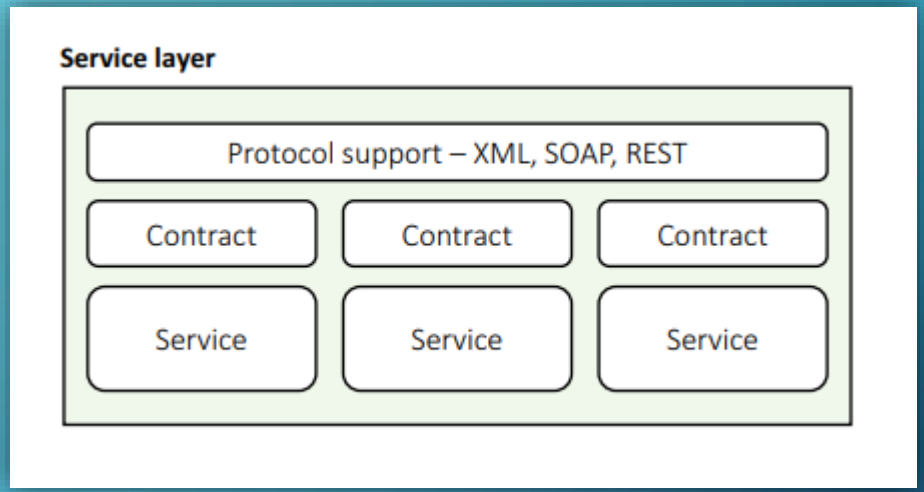
THE DATA LAYER

- Use Components:
  - Handling diverse data sources
    - SQL Server, Redis, MongoDB, etc.
  - Increased abstraction
  - Declarative configuration
    - Easily replace connection string, data storages and other mechanisms



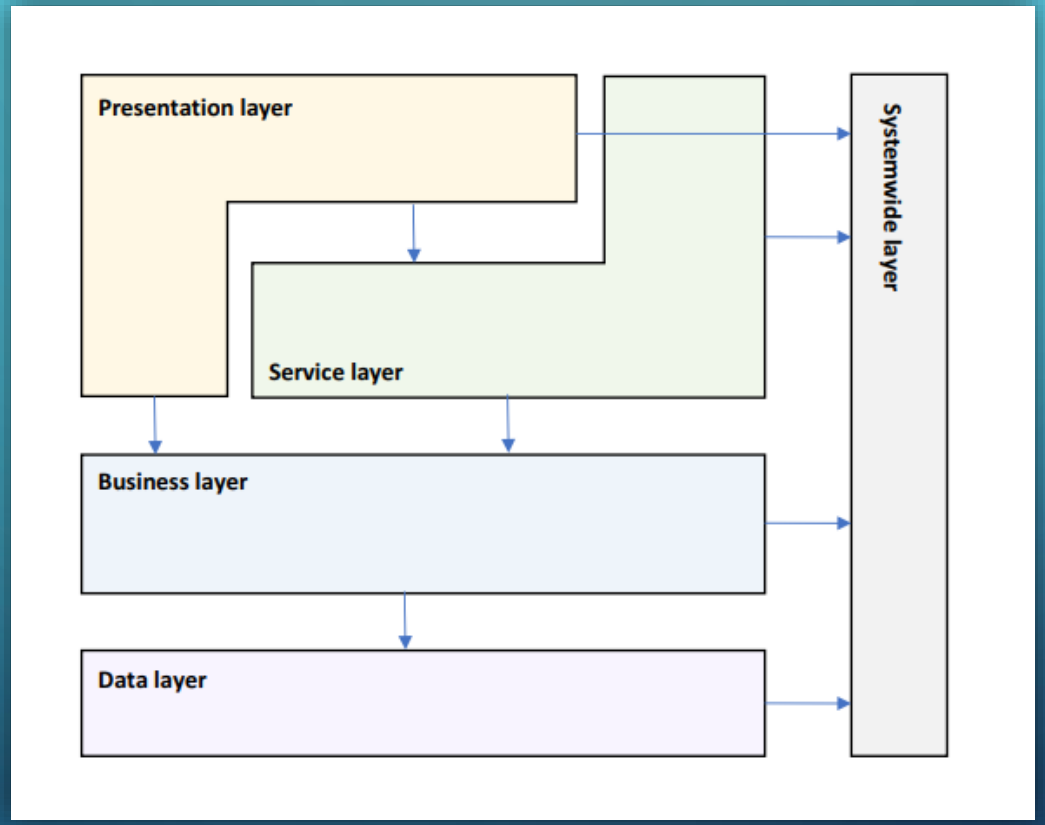
THE SERVICE LAYER

- Use Components:
  - Contract management
  - Containers
  - Declarative configuration
- Choose Microservices if your system is composed mostly of interlocking APIs
  - Never start with Microservices
  - Always choose a monolithic application with Microservices in mind
  - Use Domain-Driven Design and Clean Architecture to separate contexts
  - Extract microservices when necessary
- Choose Message Bus if all services alter state on a common message

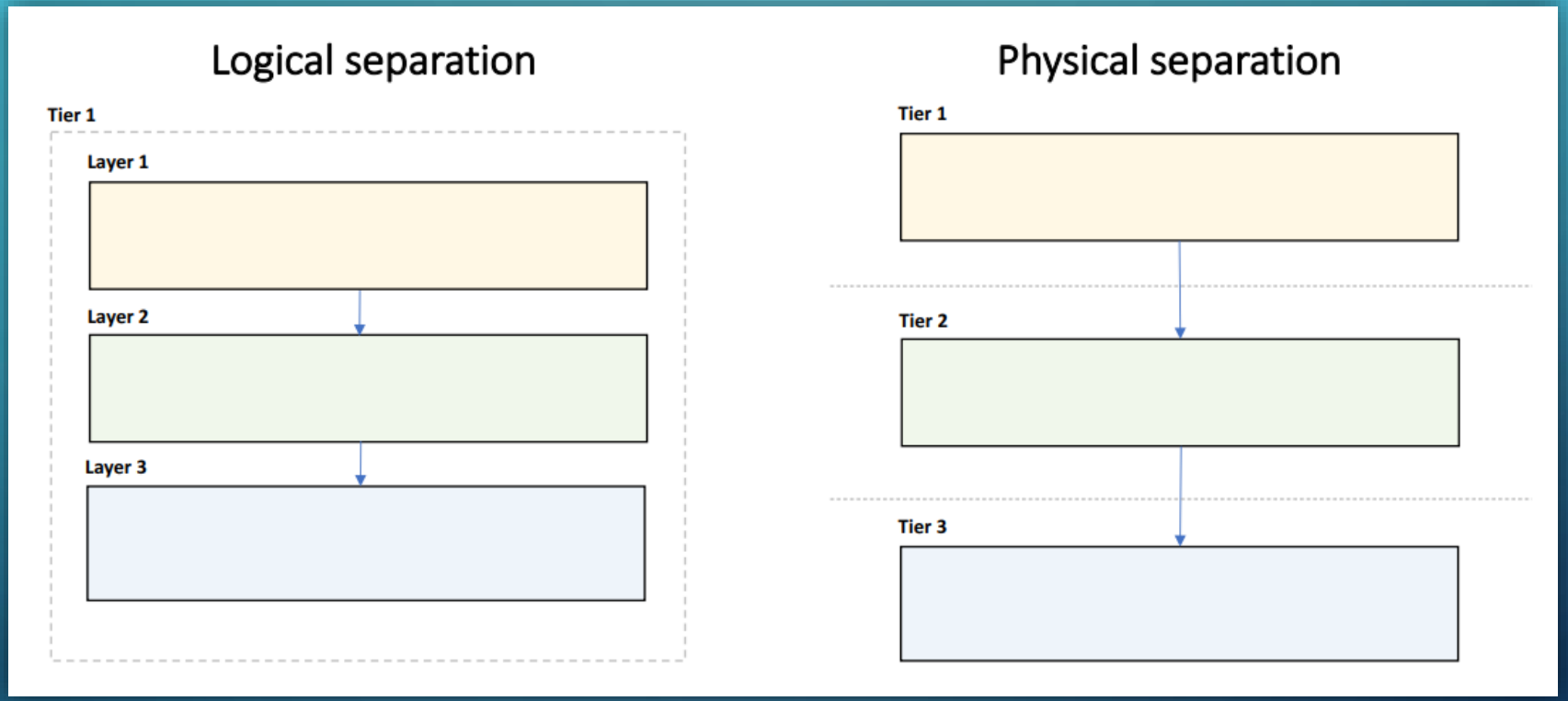


DESIGNING LAYERED ARCHITECTURES

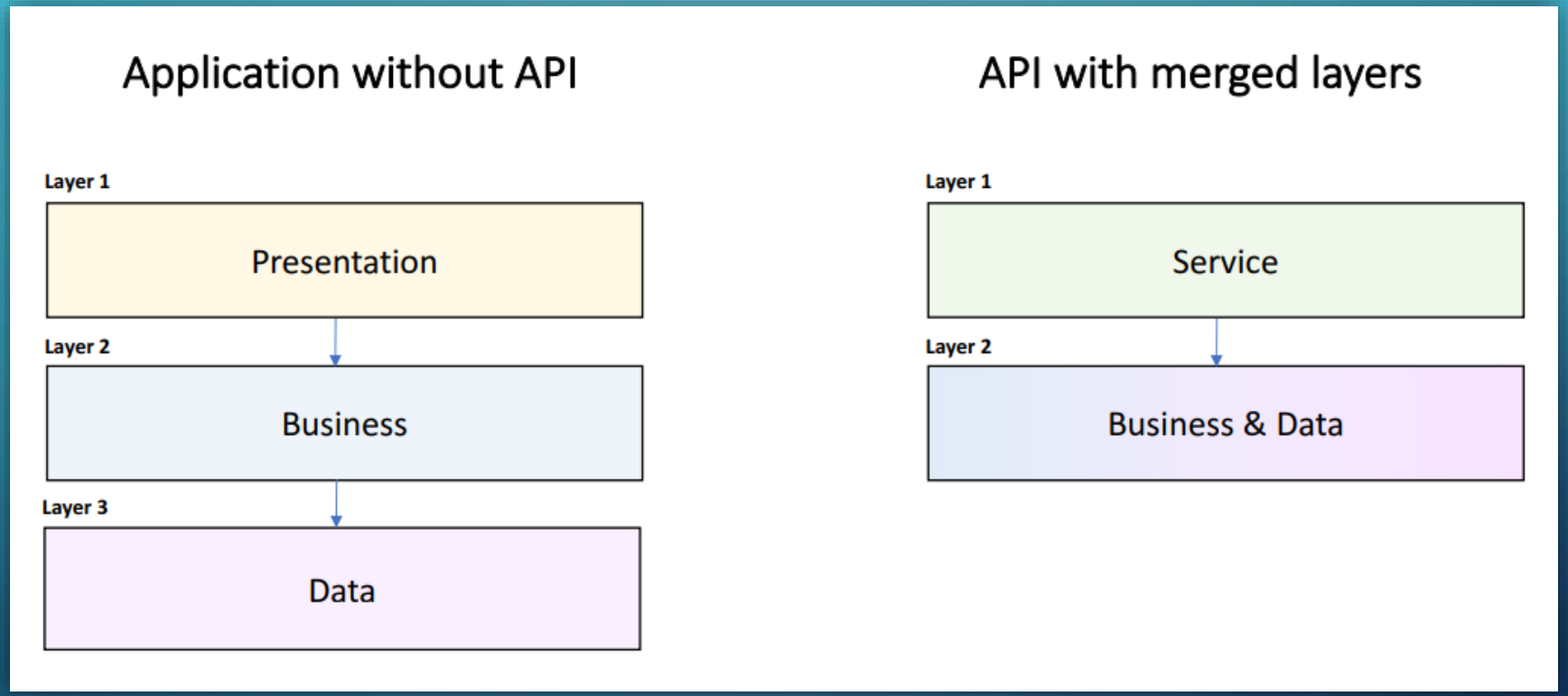
LOGICAL LAYERED DESIGN



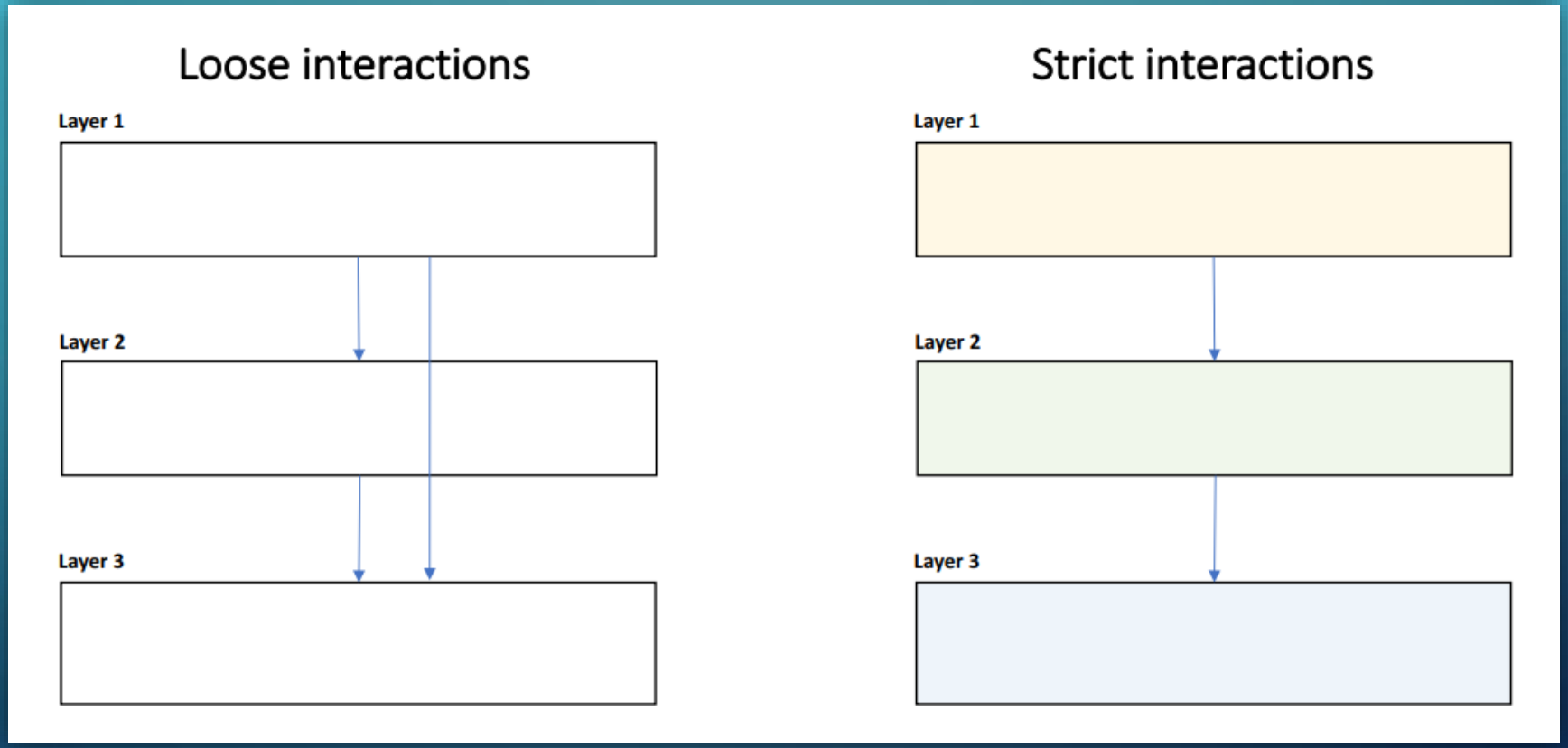
CHOOSE LAYERING STRATEGY



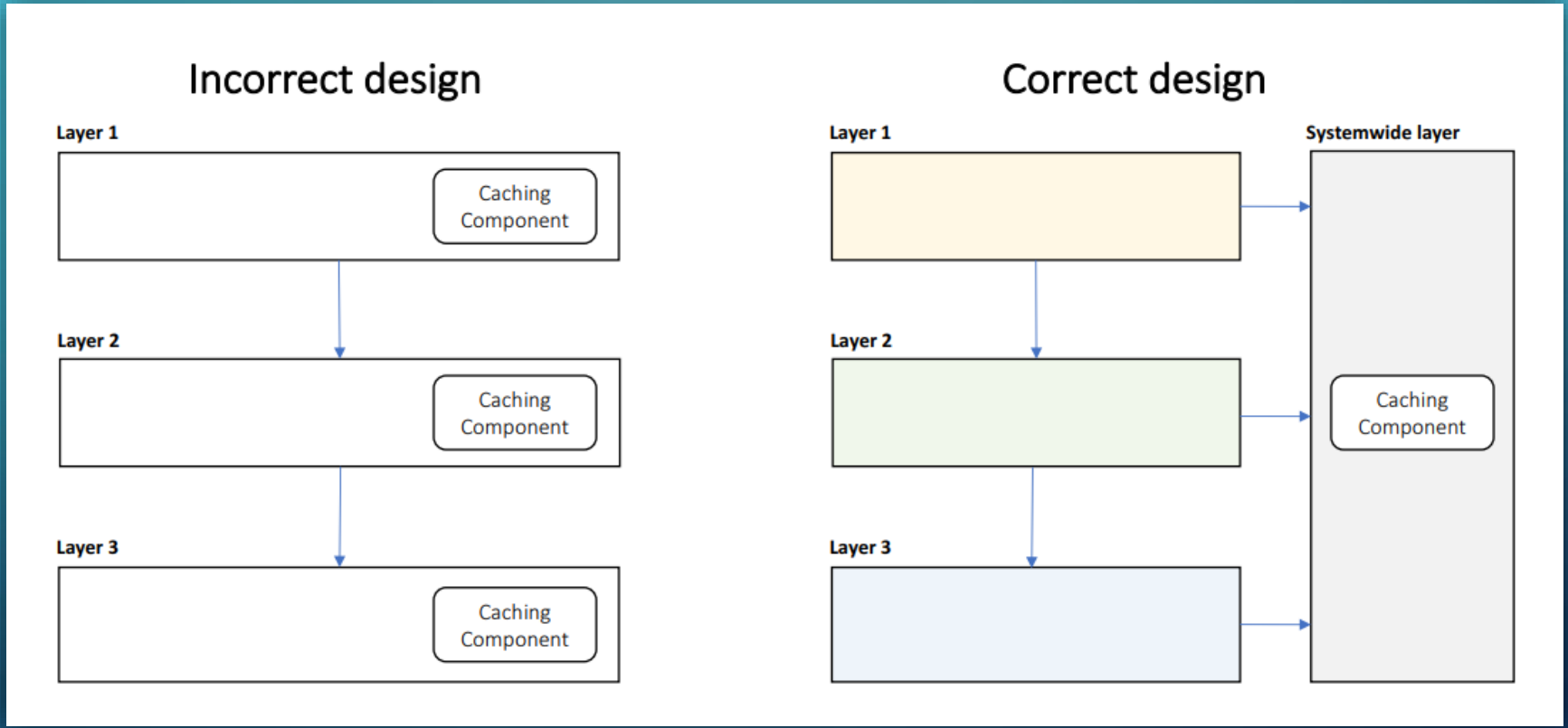
REMOVE & MERGE LAYERS



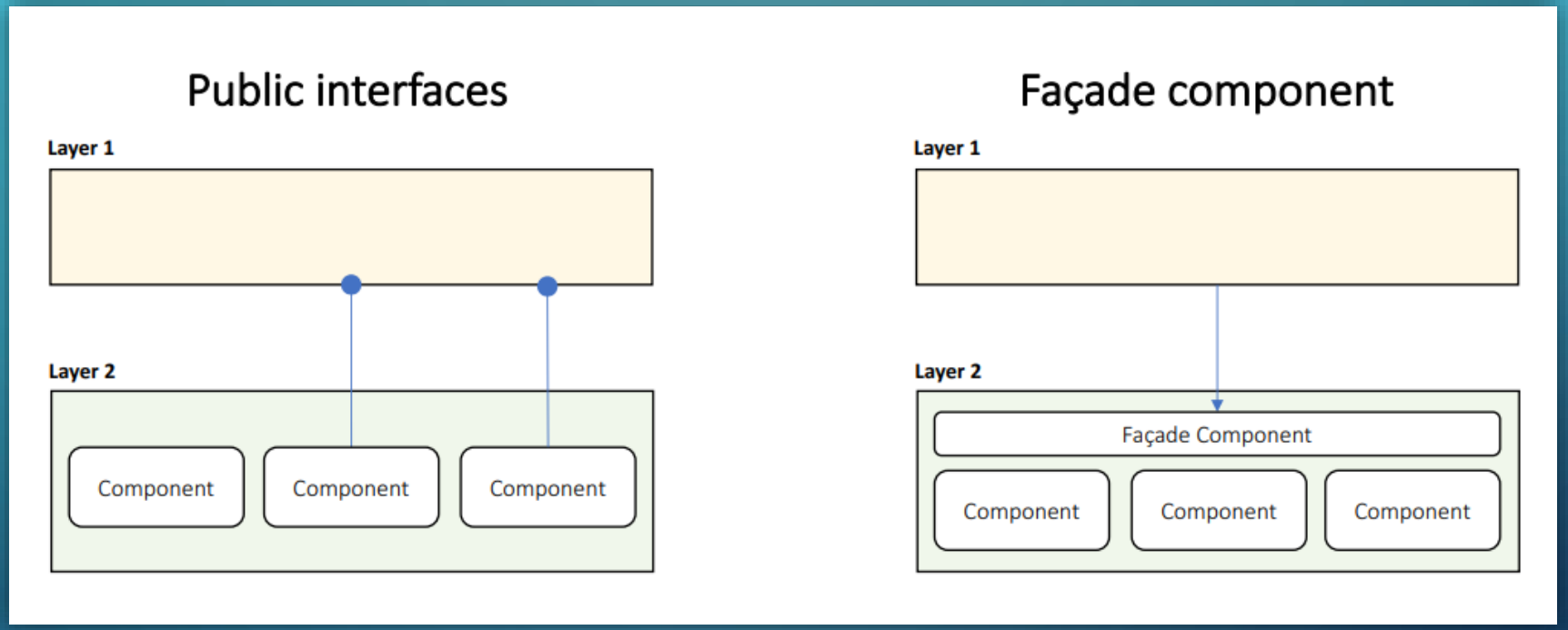
DETERMINE LAYER INTERACTIONS



IDENTIFY SYSTEMWIDE CONCERNS



DEFINE LAYER INTERFACES



DESIGNING COMPONENT ARCHITECTURES

GENERAL GUIDELINES

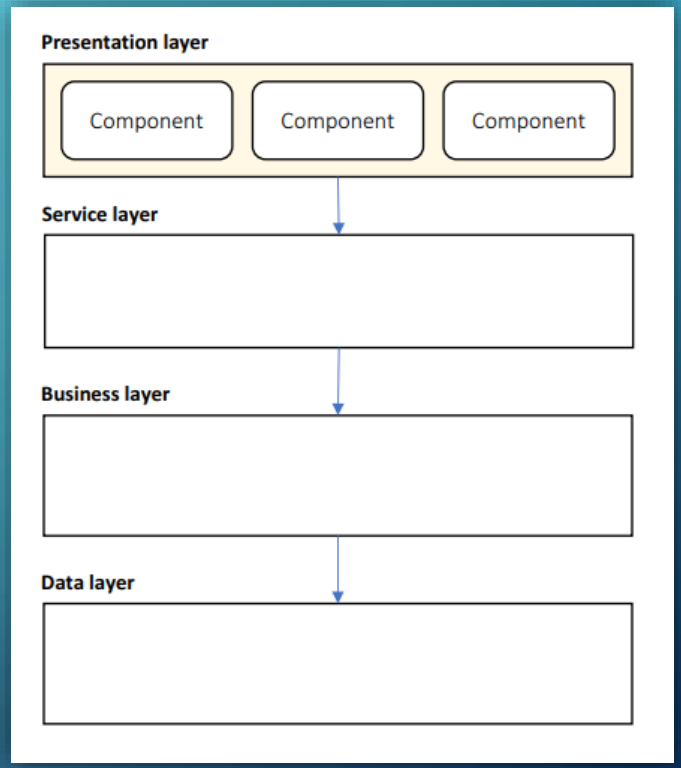
- Components are SOLID:
  - Single Responsibility
  - Open/Closed
  - Liskov Substitution
  - Interface Segregation
  - Dependency Inversion
- Highly cohesive
- Define clear naming rules
  - Classes, interfaces, methods, variable, constants, etc.
- No knowledge of internals of other components

DESIGN PATTERNS

- Design patterns are micro-architecture
  - Make sure your developers are familiar with them
- Commonly useful design patterns:
  - Factory – remove the "new is glue" syndrome from your code
  - Repository – only if you really need it, may serve as anti-corruption layer
  - Façade – hide complexity in your business logic
  - Command – encapsulate actions in objects
  - Strategy – supports open/close principle
- Learn all of them!
  - But use only the ones you need!

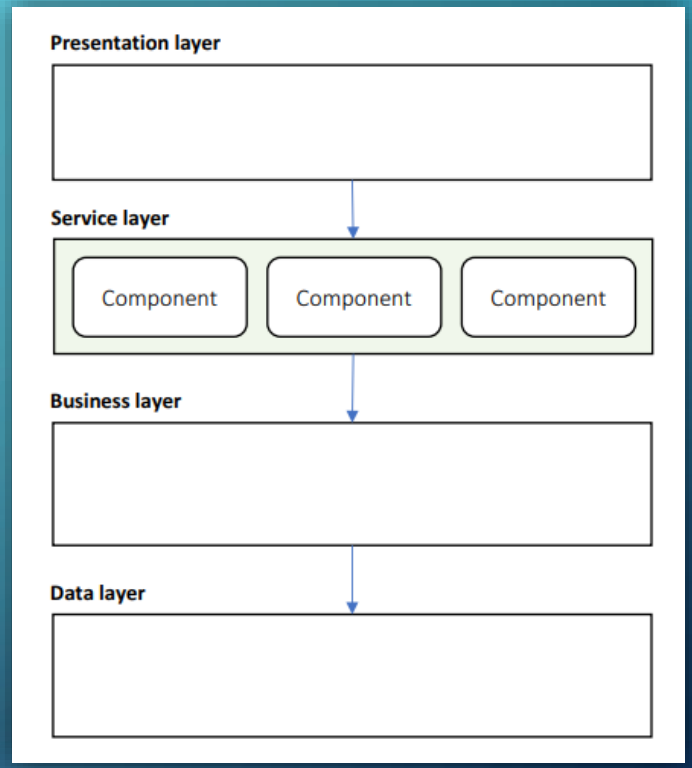
PRESENTATION LAYER COMPONENTS

- UI components
  - Presentation Logic
  - Views
  - Controllers
  - View Models
- Optional:
  - Presentation entities



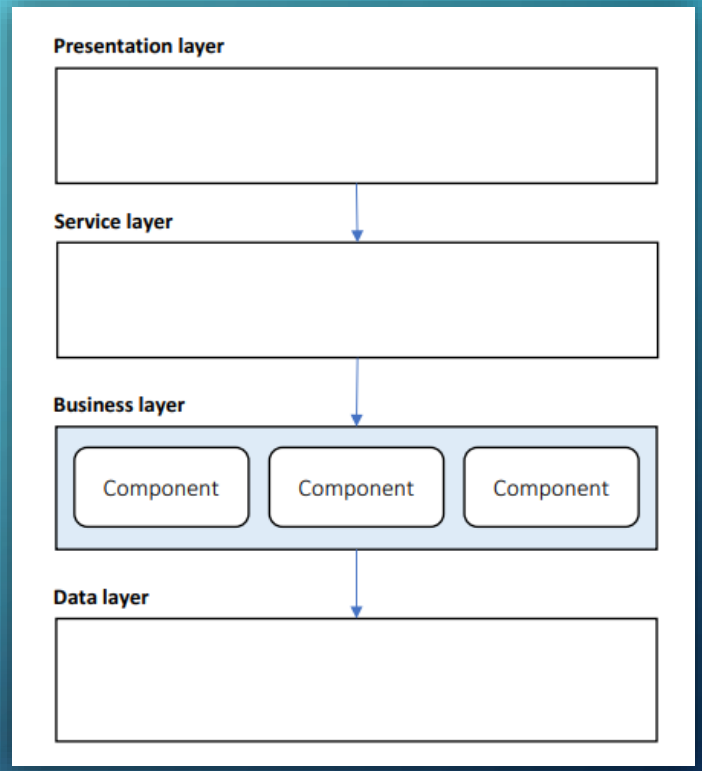
SERVICE LAYER COMPONENTS

- Services
- Service Interfaces
- Message Types
  - Data Transfer Objects
- Optional:
  - Message Broker



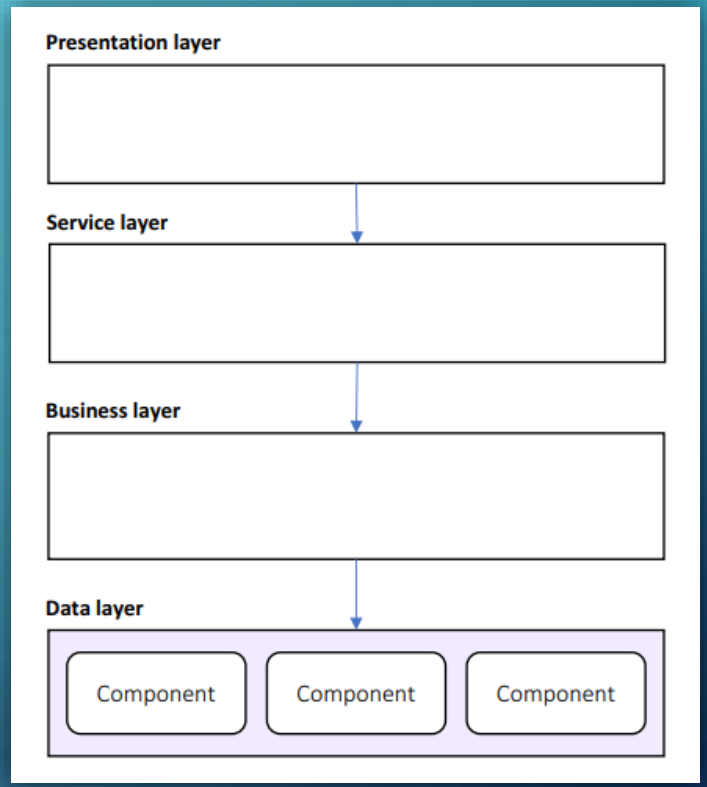
BUSINESS LAYER COMPONENTS

- Business Façade
- Business Logic
  - Components
  - Workflows
  - Business Rules
- Business Entities
  - Populated through ORM
  - Or data objects from data layer
- Business Events

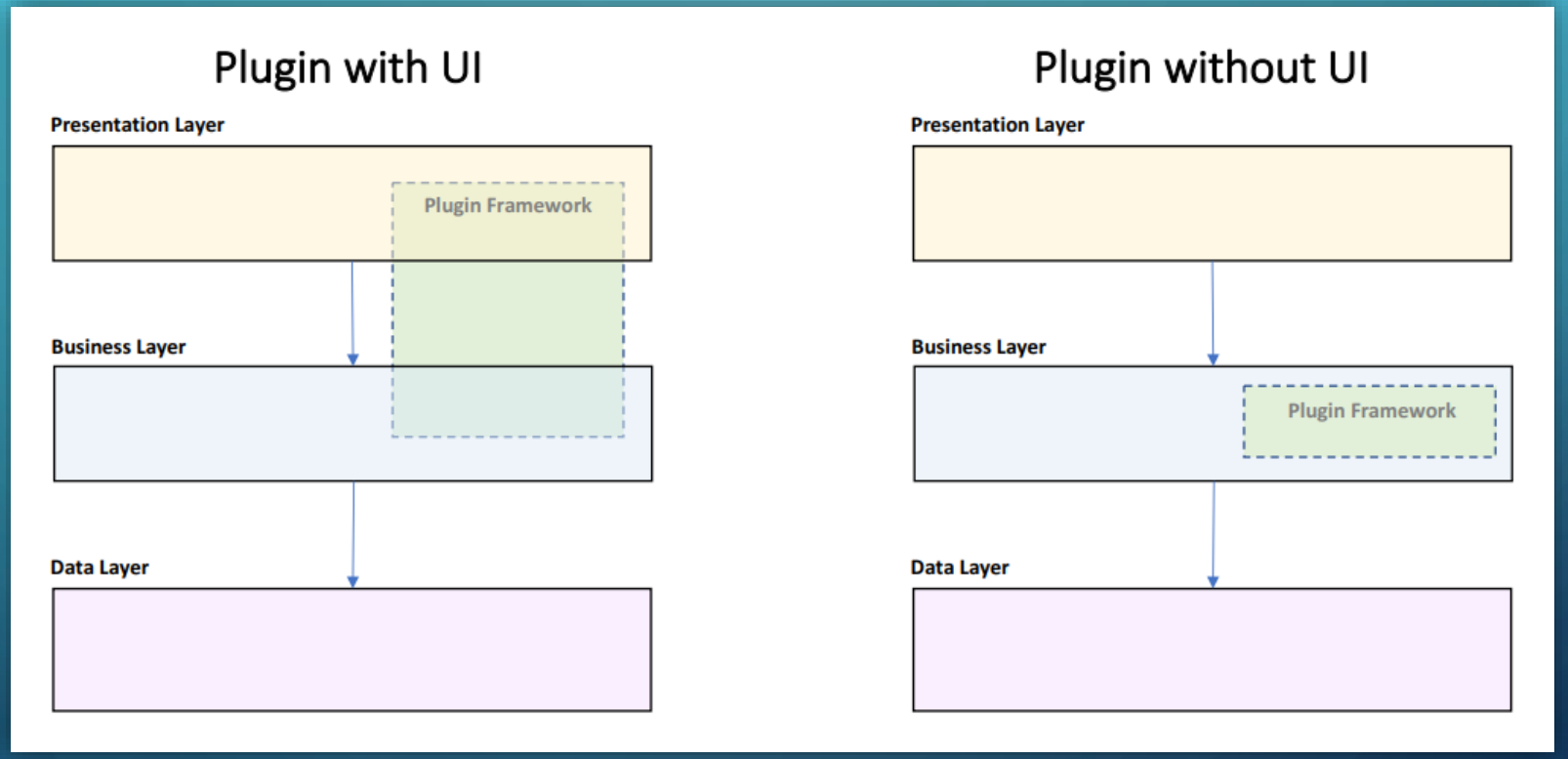


DATA LAYER COMPONENTS

- Data Façade
- Data Source Adapters
- Service Adapters
- Optional:
  - Data Objects
  - Command & Query objects



MODULAR ARCHITECTURES



DESIGNING SERVICE-ORIENTED ARCHITECTURES

REST SERVICE

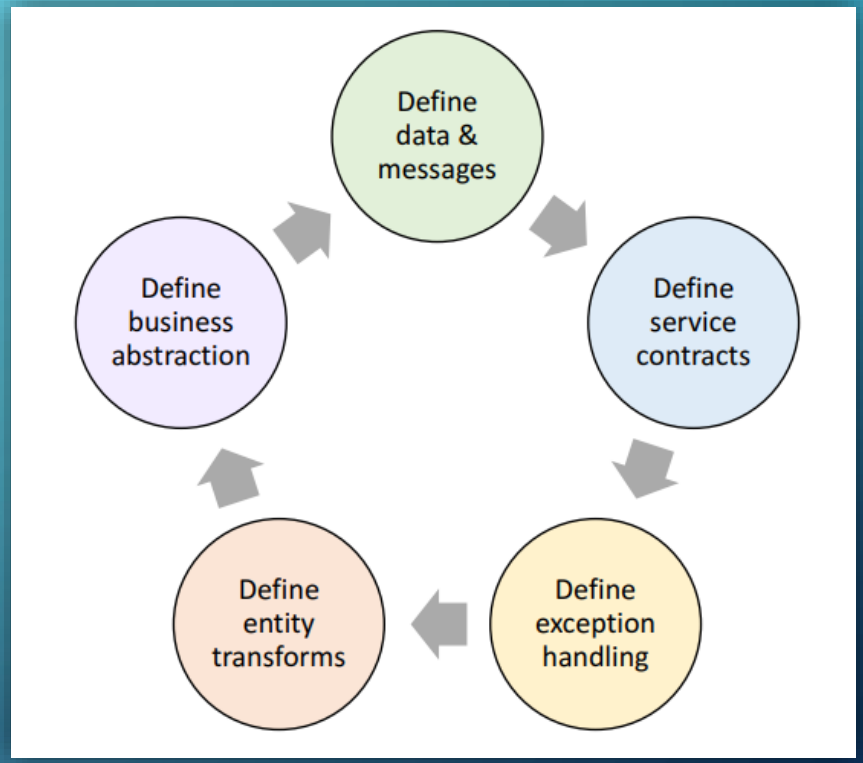
- CRUD operations
- Operates on Entities
- Content negotiation
  - JSON, XML, HTML, Markdown…
- Low overhead
  - Using web servers and HTTP
- Request/Response
- No discovery & routing
- Incomplete standard
- Fairly new

SOAP SERVICE

- Any operations
- Not limited to Entities
- XML
- High overhead
  - More interface layers
- Request/Response, Fire & Forget, Bi-Directional Calls
- Standard discovery & routing
- World standard
- You usually use it with older and legacy systems

THE SERVICE DESIGN PROCESS

- Define data & messages
- Define service contracts
- Plan exception handling
- Define how business entities are transformed to messages
- Define how business functions are abstracted to services



DEFINE SERVICE MESSAGES

- Request/Response, Fire & Forget, or Bi-Directional
- Command, Query, Document, Entity, Event, Message…
- Avoid large messages
- Add expiration & diagnostic info
- Define in a Class Diagram

DEFINE SERVICE CONTRACTS

- CRUD or RPC
- Stateful/Stateless
- Transaction Management
- Handle invalid calls:
  - Timeouts
  - Duplicate calls
  - Calls out of order
- Define in a Component Diagram

PLAN EXCEPTION HANDLING

- Only catch what you can handle
- Use meaningful messages:
  - Business explanation
  - Technical information
  - Retry instructions
- Return fault metadata
- Log everything
- Notify exception subscribers

DEFINE BUSINESS ENTITY TRANSFORMS

- Transform entities to messages:
  - Reference Business layer directly
  - Use an Object Mapper
  - Use an Object-Relational Mapper
  - Use a Transform Language
  - Custom-built
- Considerations:
  - Narrowing/Widening transforms
  - Flattening/Elevating transforms
  - Reversible transforms

DEFINE BUSINESS ABSTRACTION

- Business layer abstractions:
  - Call the Façade directly
  - Call Business Components
  - (Re)Start a Workflow
  - Log a Business Event
- Considerations:
  - Long-running workflows
  - State management
  - Transactions

QUESTIONS FOR THE LEARNING SYSTEM

- We can now move within a new cycle and create our first candidate solution
- Objectives and key scenarios stay the same but let's refine the architecture

1a. Are we happy with our layers? Remove or merge layers? Their communication?

1b. Revisit the key issues and the system-wide layer. Are we happy with them? Our third-party tools?

2a. Are we happy with our presentation layer? Do we want to add presentation entities?

2b. Are we happy with our business layer? Choose a mapping strategy.

2c. Are we happy with our data layer and the level of abstraction?

3a. Start filling the service layer. Choose REST or SOAP. Choose exception handling strategy.

3b. Describe service messages. Create a new UML class diagram describing public properties.

3c. Describe the service contracts and their public interfaces.

3d. Decide how the service layer will communicate with the business components.

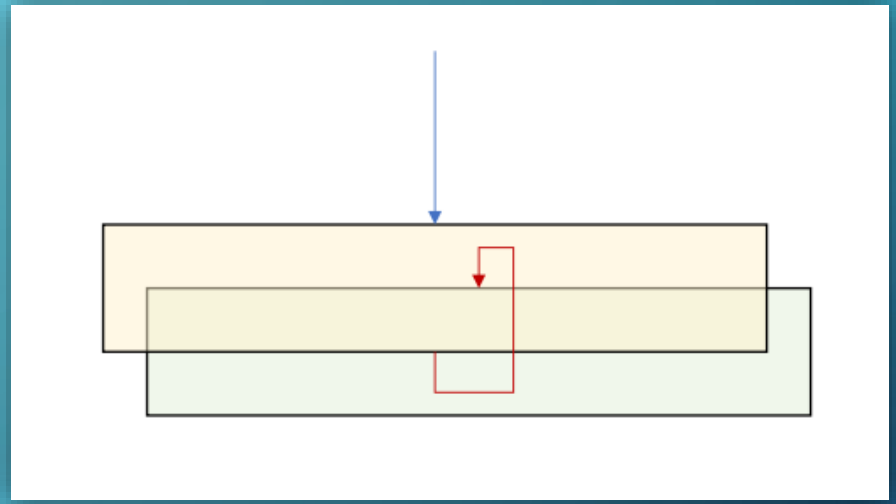
3e. Choose a transformation engine.

4. Bonus – forward engineer a skeleton solution.

ARCHITECTURE QUALITY ATTRIBUTES

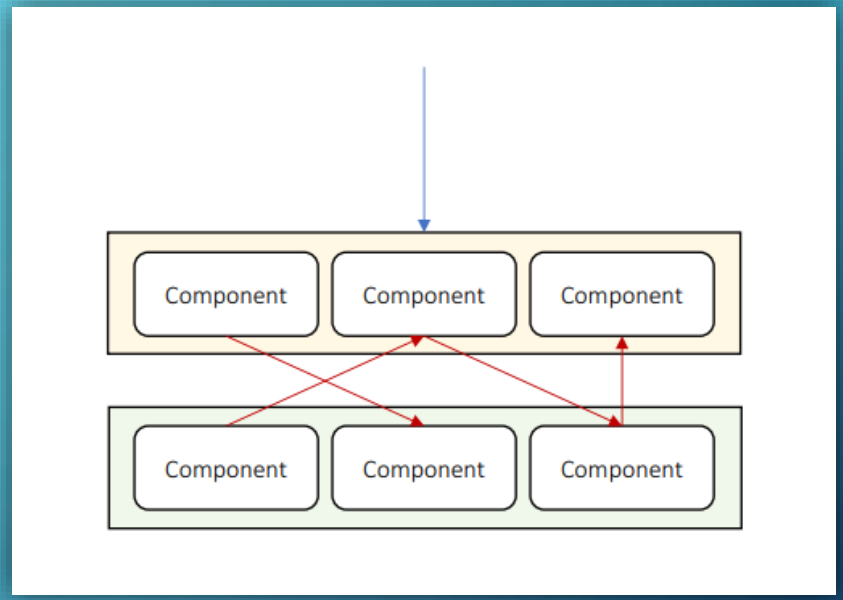
CONCEPTUAL INTEGRITY – DESIGN TIME

- Isolated layers and components
- Application lifecycle management
  - How to manage updates
- Healthy team collaboration
  - Awesome relationship with the Lead Developer
- Design and coding standards
  - Automate the in your code reviews
- Break away from legacy designs:
  - Façade pattern
  - Wrap as service
  - Rebuild from scratch



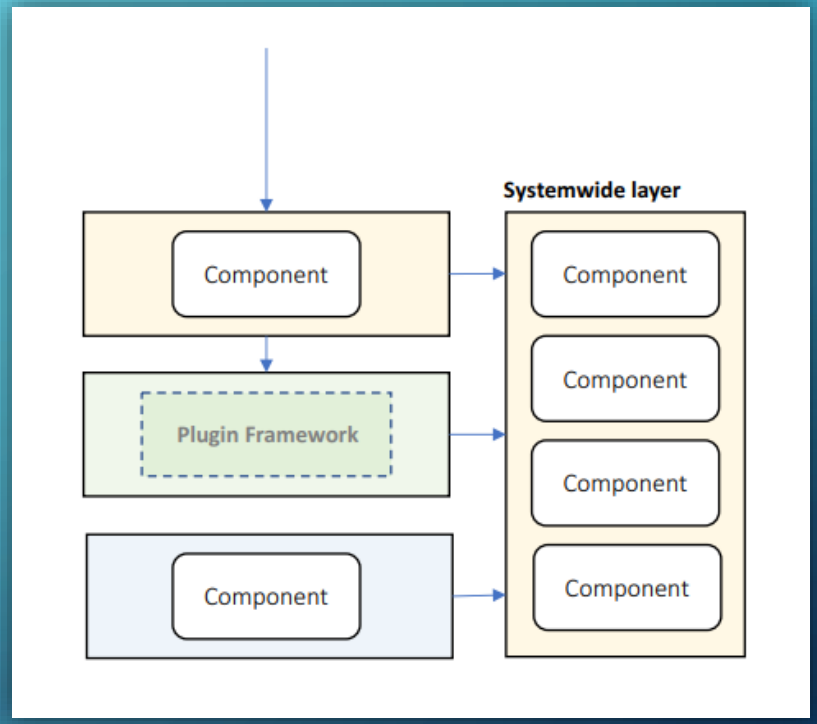
MAINTAINABILITY – DESIGN TIME

- Isolated layers and components
- Structured communication
  - Follow the same pattern for the whole solution
- Consider a plugin system
- Rely on platform features
  - Most modern languages have a lot to offer
- Use systemwide layer
- Add unit tests
- Documentation



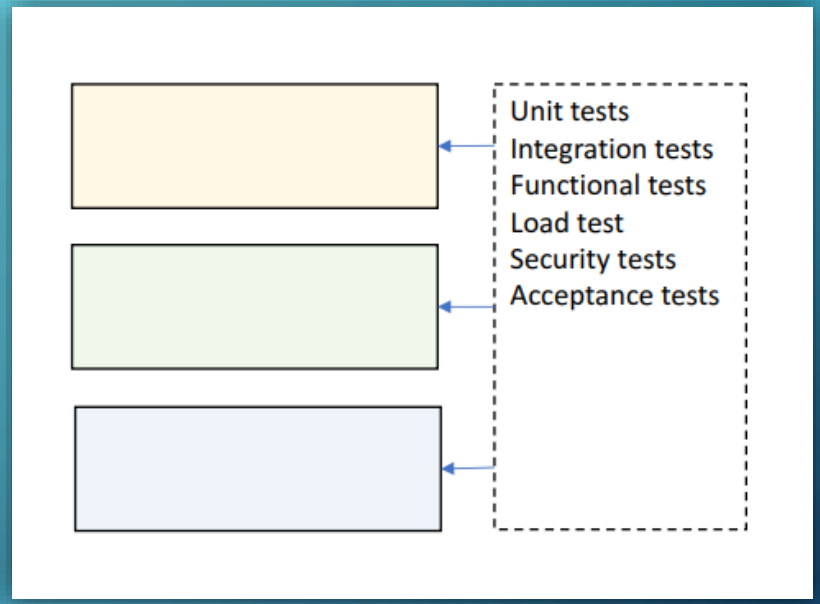
REUSABILITY – DESIGN TIME

- Component-based architecture
- Adhere to standards
- General-purpose code
  - Your components should not be specific
- Allow 3rd party software
  - You can buy tools to save time
- Use a plugin system
- Use systemwide layer



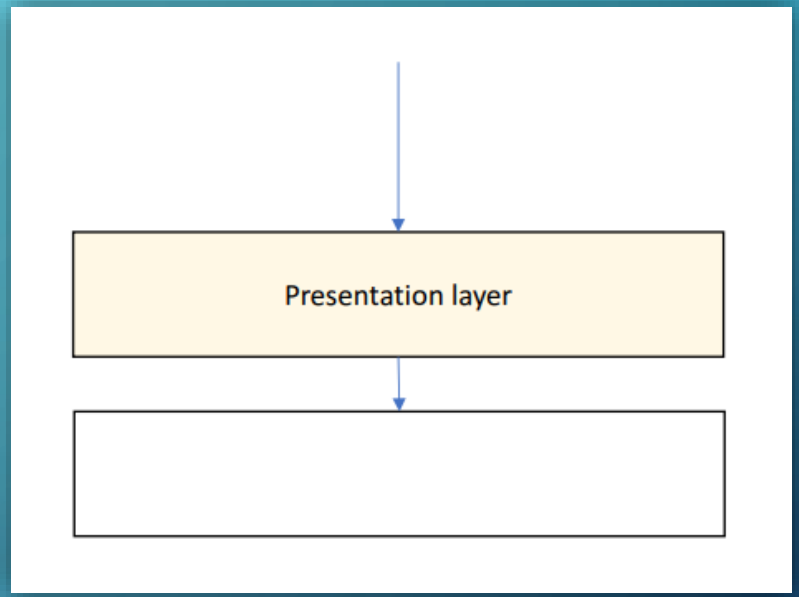
TESTABILITY – DESIGN TIME

- Design for testing
- Allow mocking
- Cover all layers
- Automate case studies
- Test:
  - Individual components
  - Entire layers
  - Collaboration between layers
  - Load, Security…



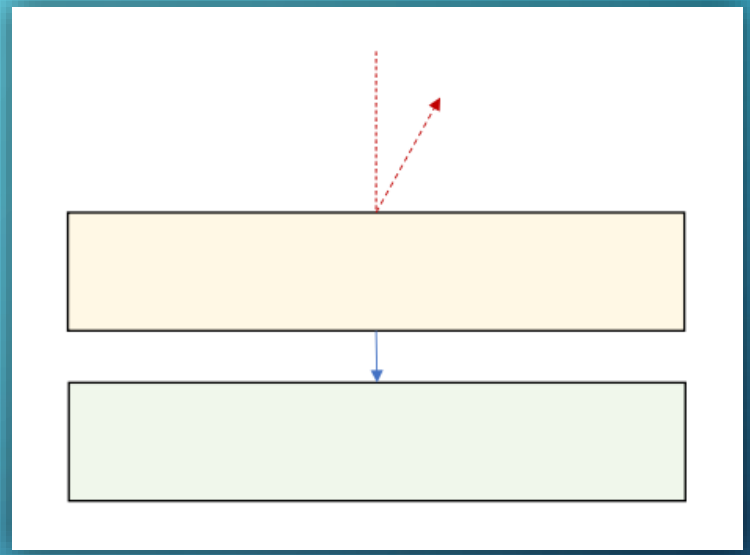
USABILITY – DESIGN TIME

- Elegant & simple UI
- Implements all case studies in minimal number of interactions
  - Minimize the user interface interactions
- Clear multi-step workflows
  - Ask for minimum number of information
- Intuitive feedback
  - Especially for exceptions
- Non-technical



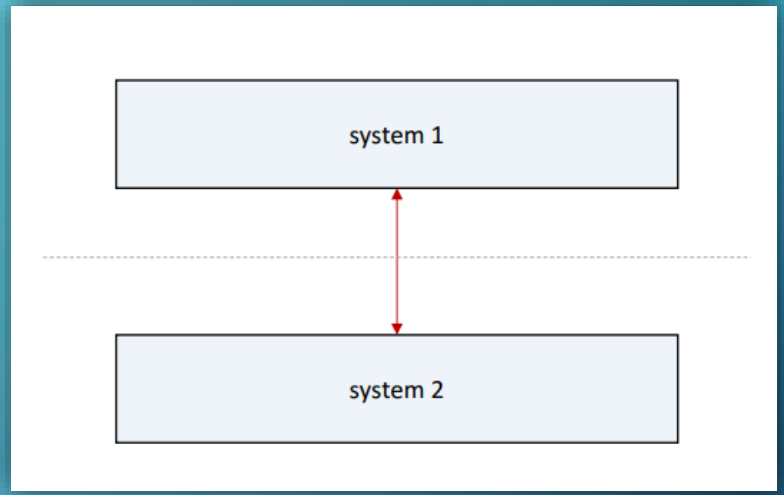
AVAILABILITY – RUNTIME

- Tier failover
- Use rate limiter
  - Limits the number of calls per second
- Short-lived resources locks
  - Or no locks at all, if that is possible
- Recover from exceptions
  - Do not kill an entire tier with a single exception
- Update-friendly architecture
- Handle network faults:
  - Offline support
  - Buffered proxy



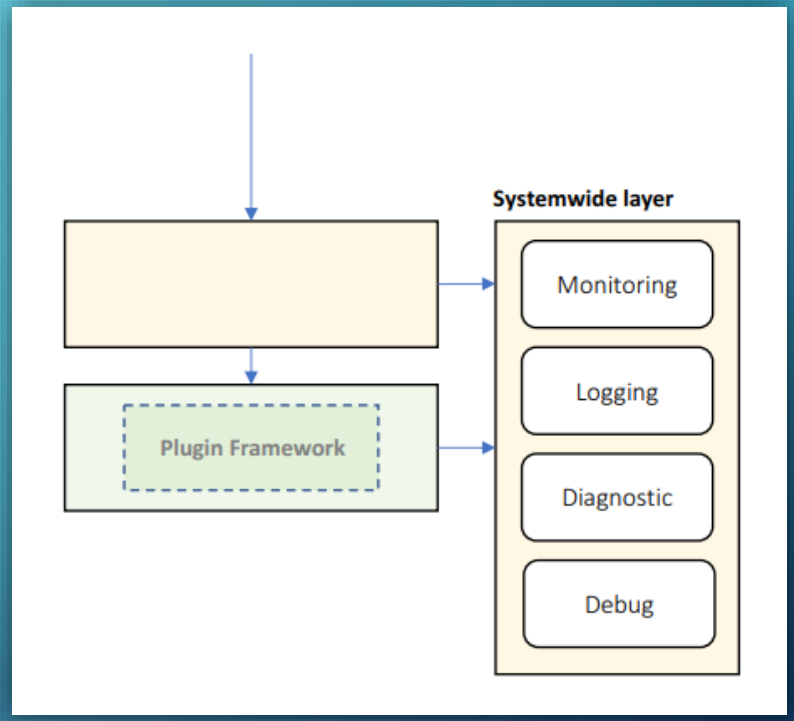
INTEROPERABILITY – RUNTIME

- Data transformation
- Keep systems separate
- Adhere to standard:
  - SOAP
  - REST
  - XML/JSON



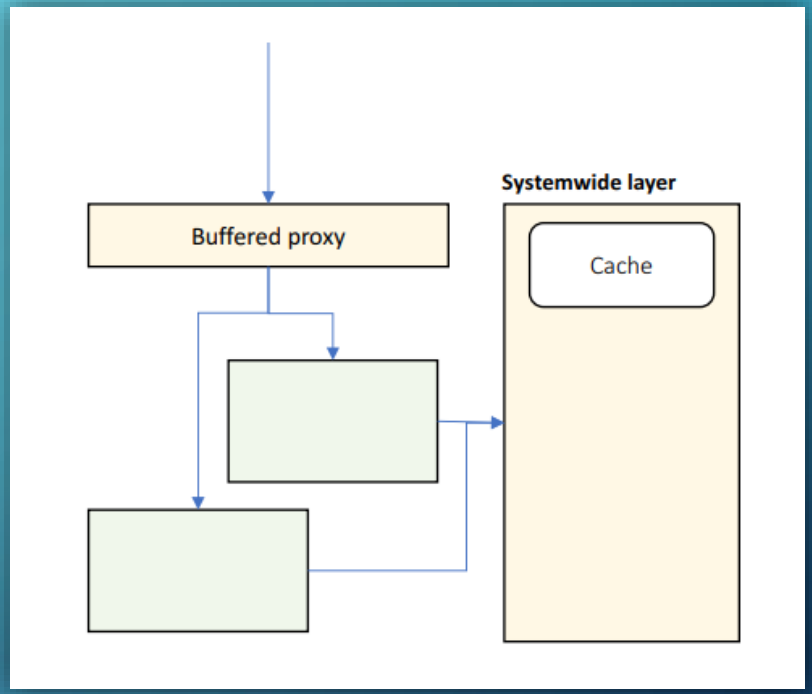
MANAGEABILITY – RUNTIME

- Health monitoring, logging, and diagnostic tracing
- Consider a plugin or modular system
  - Much more flexible development
- Declarative configuration
- Add diagnostic tools:
  - Live tracing
  - Diagnostic notifications
  - Runtime log inspection
  - Runtime debugging
    - Dangerous so use it only in critical situations



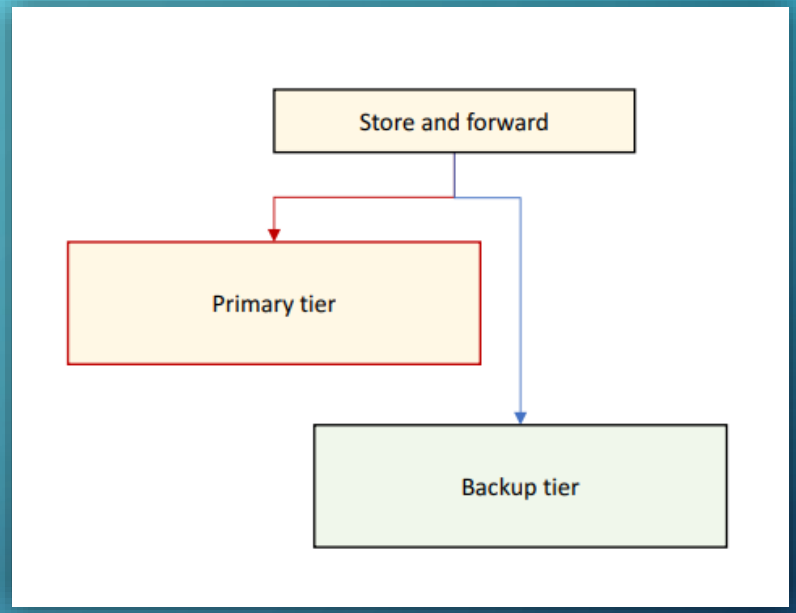
PERFORMANCE – RUNTIME

- Buffered proxy
- Async responses
  - The client gets immediate response with progress bar
- Load-balanced tiers
- Caching
- Load tests
- Minimize throughput:
  - Rate limiting
  - Design coarse interfaces
  - Minimize cache misses



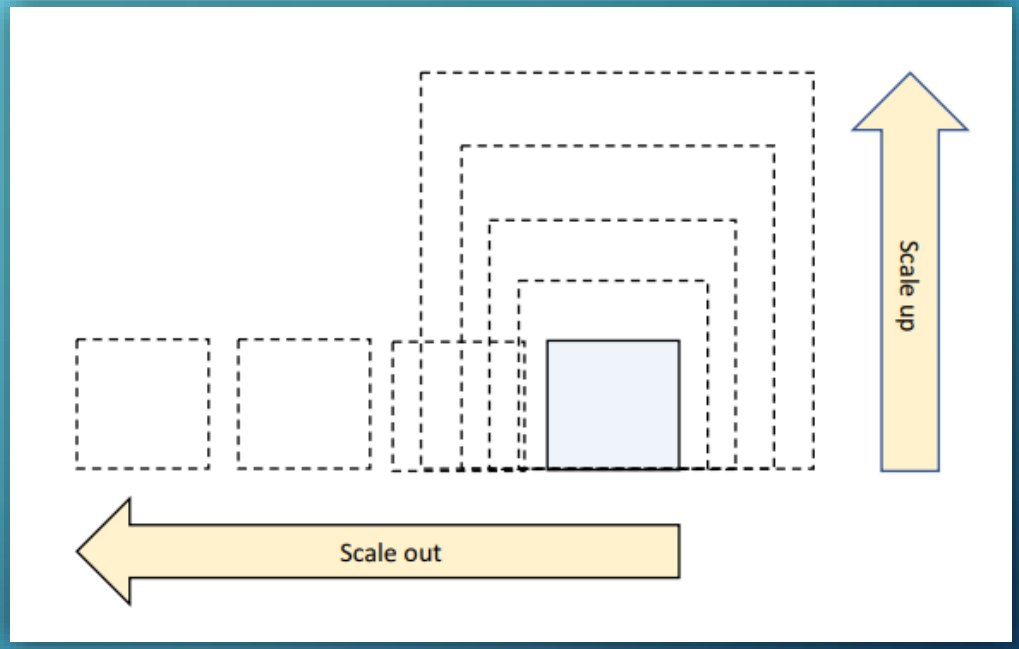
RELIABILITY – RUNTIME

- Self-healing architecture
- Use store and forward
- Use alternative system if:
  - Primary system is offline
  - Primary system is very slow
  - Primary output is invalid
- Replay messages when external resources come back online



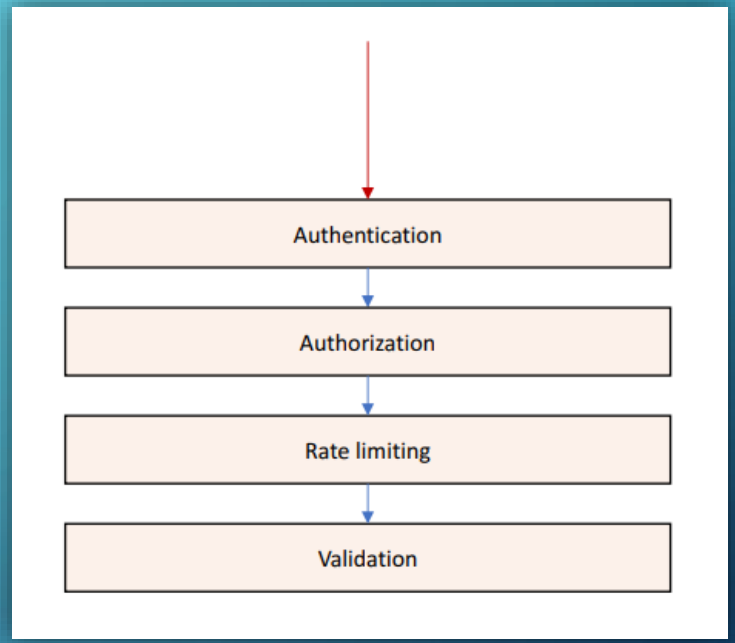
SCALABILITY – RUNTIME

- Adding computer resources without any interruption
- Tier scaling:
  - Scale up – increate resources
    - Single point of failure
  - Scale out – duplicate tiers
    - For stateless applications
    - The better approach
- Handle load spikes:
  - Async responses
  - Store and forward
  - Allow stale data



SECURITY – RUNTIME

- Authenticate & authorize clients
- Validate input & output
- Encrypt sensitive data
- Protect against:
  - Spoofing
  - Malicious input & output
  - Malicious use
  - Data theft
  - DDoS attacks



SYSTEM-WIDE CONSIDERATIONS

GENERAL GUIDELINES

- Always think about the big picture
- Consider the following questions:
  - How will the system work under heavy load?
  - What will happen if the system crashes at this exact moment in the business flow?
  - How complicated can be the update process?
  - There could be more depending on the specific requirements of the system
- Main concerns for the system architecture are:
  - Loose coupling
  - Stateless
  - Caching
  - Messaging
  - Error Handling & Logging

LOOSE COUPLING ON A SYSTEM LEVEL

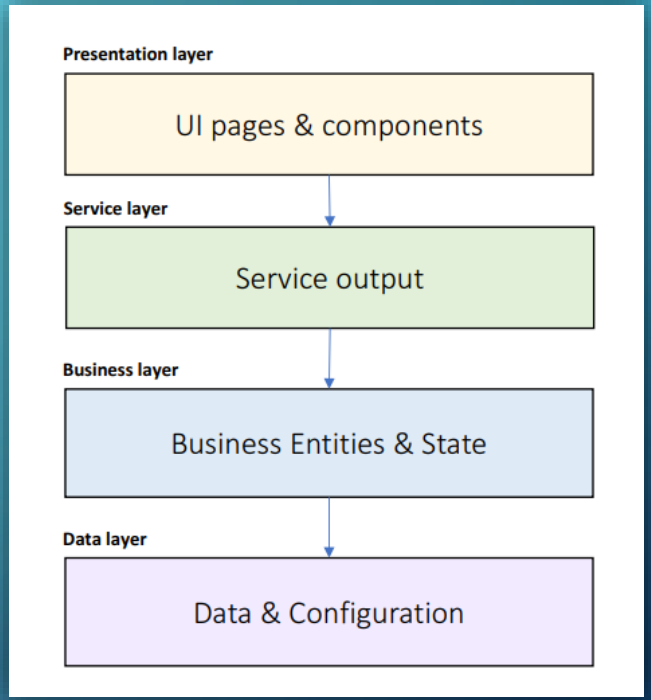
- Making sure our services are not tightly coupled to other services
- Every time we change one process, another needs to follow
- A good idea is to be platform independent
  - Do not use Java-specific communication, for example
  - Because the other service should also be written in Java
- Also, the services should be URL independent
  - Imagine multiple services like in a typical application
  - They all communicate with each other
  - If one URL changes – massive issues
  - Possible solution – yellow pages service used for discoverability
    - Consul is a great tool for that purpose
  - Another solution – a gateway service to route requests

STATELESS APPLICATIONS

- Super important for scalability and performance
- The application's state should be stored only in two places
  - The data storage
  - The user interface
  - No state is stored in application code
- Typically, the user-specific data is the application state
  - Make it stateless
- Why it is important?
  - Scalability - scaling out will be super easy
  - Redundancy – if one server fails, the other continue to work

CACHING – WHAT TO CACHE?

- UI pages
  - Mainly useful for seemingly static information
- UI components
  - These parts of the page which do not change often
- Service output
- Business Entities
- Business State
- Data query results
- Configuration data
- List data to be cached in each layer



CACHING – DEFINITION

- Caching is bringing the data closer to its consumer so that its retrieval will be faster
- Cache options:
  - Client-side cache – using headers to store data in the client's machine
  - CDNs – using a content-delivery network for static assets
  - Service cache – caching in the application code
- Cache types:
  - In-memory cache – size is limited but easily implemented (static concurrent collection)
  - Distributed cache – data is stored in separate process, slower performance

CACHING – WHERE TO CACHE?

- Local memory
  - The fastest
- State server
- File system
- Database
  - The slowest
- User standard solutions:
  - Redis
  - Memcached
  - Framework-specific caches

CACHING – HOW TO MANAGE THE CACHE?

- Expiration strategy:
  - Time-based
  - Event-based
- Flush strategy:
  - Manual
  - Automatic:
    - Least Recently Used
    - Least Frequently Used
    - Priority

CACHING – HOW TO FILL THE CACHE?

- Proactive loading
  - Static data
  - Known update frequency
  - Known size
- Reactive loading
  - Volatile data
  - Unknown lifetime
  - Large data volume
  - Fast caching medium

MESSAGING – DEFINITION

- Messaging is the communication between the various services
- Messaging methods are not exclusive
  - You can mix them easily
- Messaging criteria:
  - Performance – faster methods are preferred
  - Size – small or large messages
  - Execution model – blocking or asynchronous
  - Feedback – whether the message has failed
  - Reliability – a message will be received even if there was a problem
  - Complexity – development effort

MESSAGING – REST API

- The standard for HTTP-based systems
- Messaging criteria:
  - Performance – very fast
  - Size – same as HTTP limitations
  - Execution model – request/response – bad for long processes, blocking
  - Feedback – immediate feedback
  - Reliability – retry policy should be implemented, the circuit-breaker pattern is great
  - Complexity – extremely easy
- Useful for traditional web applications

MESSAGING – REAL-TIME COMMUNICATION

- Using HTTP Web Sockets
  - Publish/subscribe, push notifications
  - Socket.io, SignalR
- Messaging criteria:
  - Performance – very fast
  - Size – limited, no more than a few KB
  - Execution model – web socket connection or long polling
  - Feedback – fire & forget
  - Reliability – complex to implement proper reliable solution
  - Complexity – extremely easy
- Useful for chat, monitoring, and real-time data

MESSAGING – QUEUE

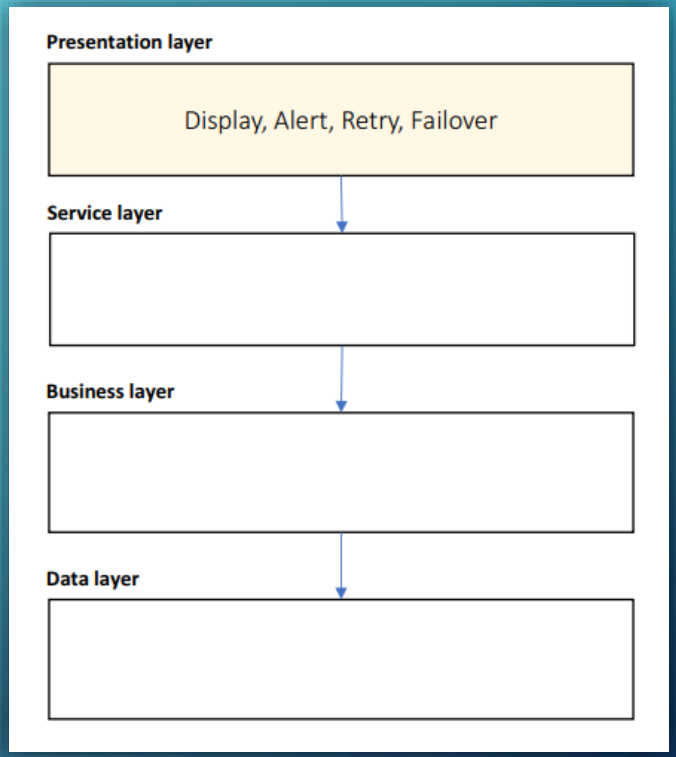
- Services do not communicate directly
  - The first service pushes to a message queue
  - The second service pulls from the message queue
  - Messages will be handled once and only once + in order
- Messaging criteria:
  - Performance – not so good, database persistence involved
  - Size – unlimited, but keep it small
  - Execution model – polling
  - Feedback – excellent feedback
  - Reliability – very reliable
  - Complexity – requires training and setup
- Useful for complex systems with lots of data

EXCEPTIONS – EXCEPTION STRATEGIES

- Allow to propagate
- Catch and Re-throw
  - Logging
  - Retains stack trace
- Catch, Wrap, and Throw
  - Add metadata
  - Expose consistent exception types
- Catch and Discard

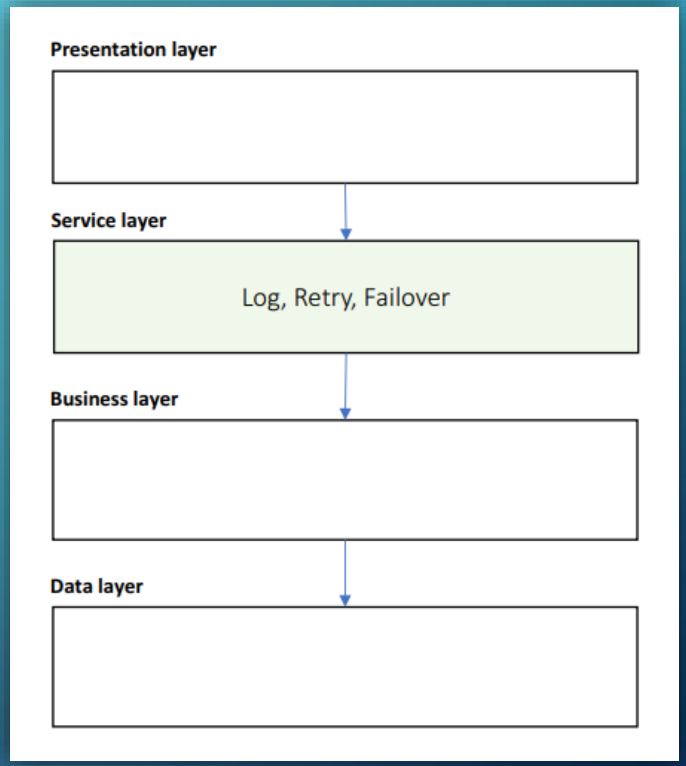
EXCEPTIONS – PRESENTATION LAYER

- Catch, display, and discard
  - Other layers should never discard exceptions
- Attempt to retry
  - Automatic 3 retries or manual
- Switch to secondary system
- Alert by Email, SMS, Slack…
- Use meaningful messages
  - Business explanation
  - Technical information
  - Steps to resolve



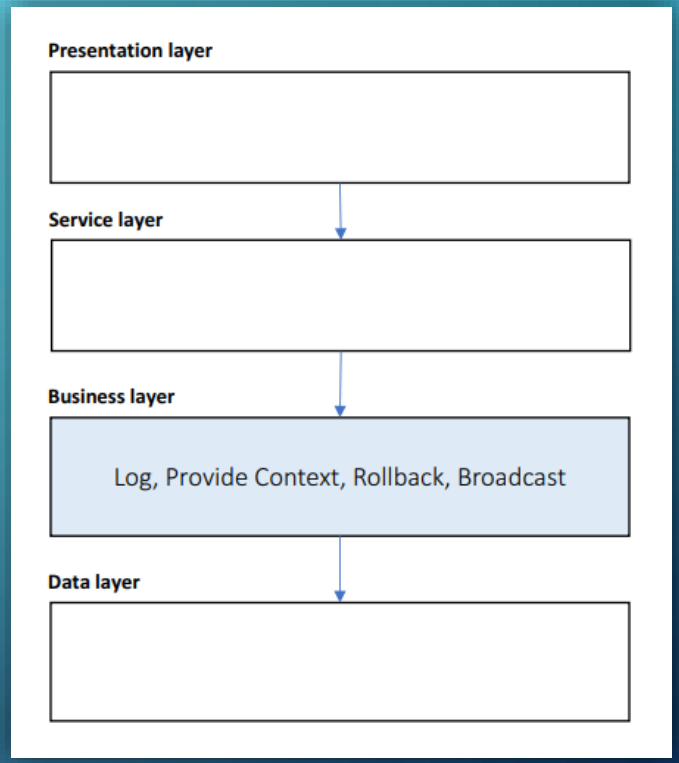
EXCEPTIONS – SERVICE LAYER

- Catch and Re-throw
- Attempt to retry
- Switch to secondary system
- Log exception and input message



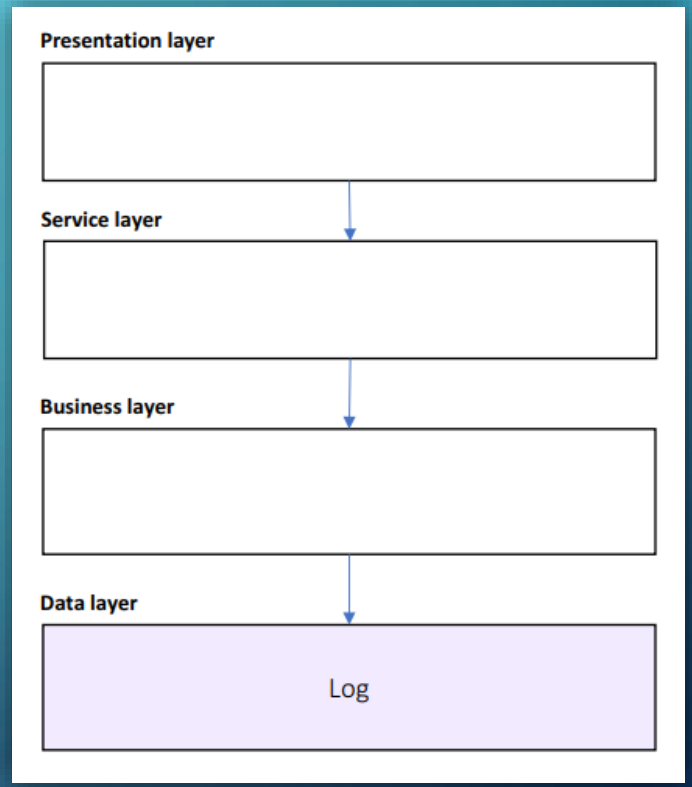
EXCEPTIONS – BUSINESS LAYER

- Catch, Wrap, and Throw
- Use custom exception types
- Provide business context
- Rollback transactions
- Log exception and input arguments
- Broadcast to subscribers



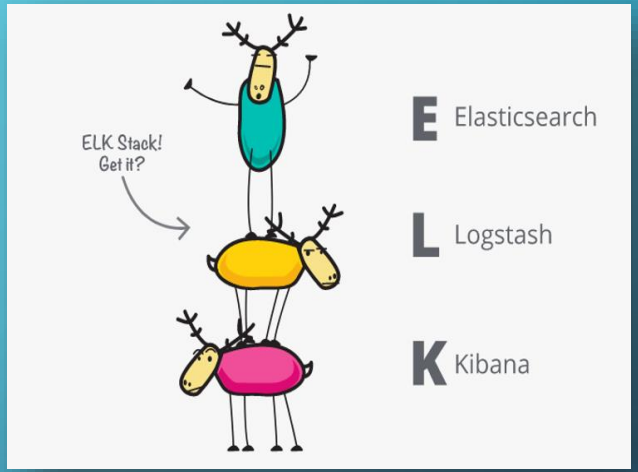
EXCEPTIONS – DATA LAYER

- Catch and Re-throw
- Log exception and input query



LOGGING - DEFINITION

- Logging has two purposes:
  - Track errors
  - Gather data
    - Which module is most visited
    - Performance scenarios
    - User's flow
- Log storage doesn't really matter
  - As long it is useful
  - Files
  - Database
  - Event log
- Good architectures always includes logging and monitoring!



LOGGING – BEST PRACTICES

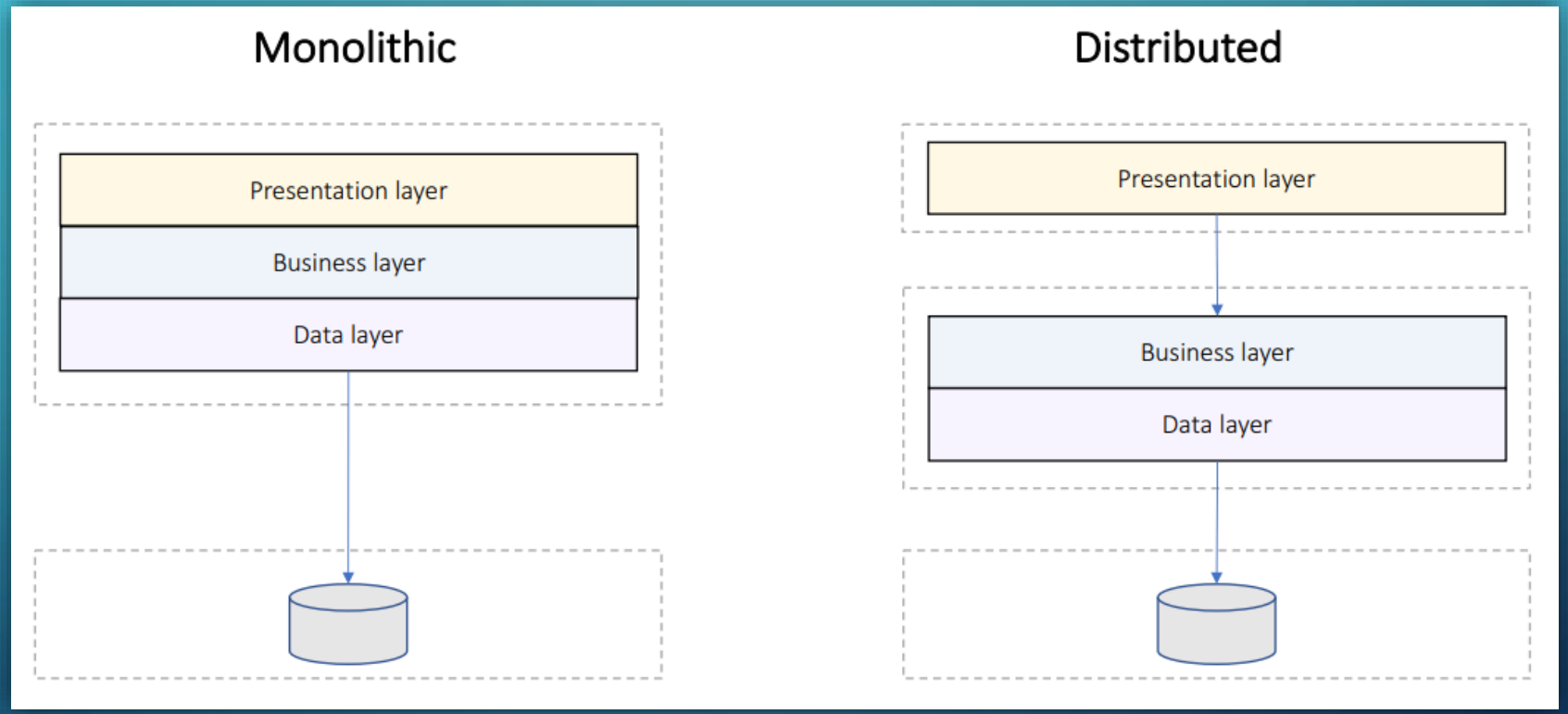
- Use a central logging service
  - Same log format
  - Same log data
  - Same log location
- Implementation:
  - Expose an API
  - Watch specific folders for log files and collect them
    - Tools like Logstash are perfect for that
- Use Correlation ID
  - If you have multiple services in a flow
  - Make sure you can link different log entries
  - You can easily track a complete flow

SYSTEM-WIDE ATTRIBUTES

- The concepts we covered in this section are super important
  - They should be part of every single system out there
  - They allow you to have a fast, secure, reliable, and maintainable solution
- Make all choices as informative and early as possible
  - Many concepts are difficult to replace once made
  - Changing a REST API with a message queue is not an easy and cheap task
- These concepts are not exclusive
  - Although they are the most important ones
  - A lot of research should be done for every single system
  - No tool is a golden hammer

DEPLOYMENT CONSIDERATIONS

DEPLOYMENT MODELS

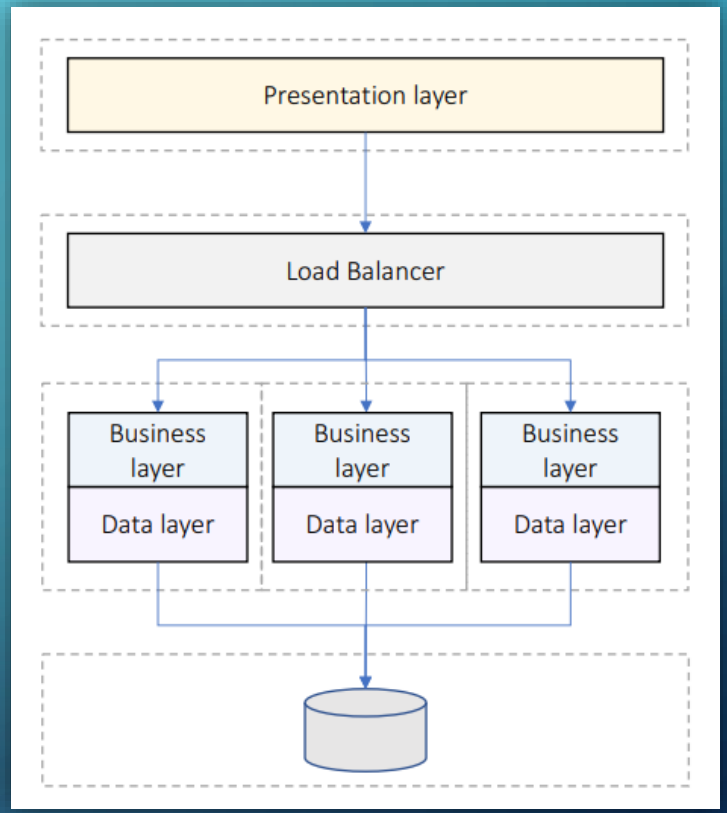


DISTRIBUTED DEPLOYMENT GUIDELINES

- Minimize blocking calls:
  - Async calls
  - One-way calls
  - Buffering
- Use distributed transactions
- Use coarse-grained interfaces
- Manage state:
  - Stateless design – highly scalable
  - Stateful design – supports workflows but doesn't scale
  - Shared state server

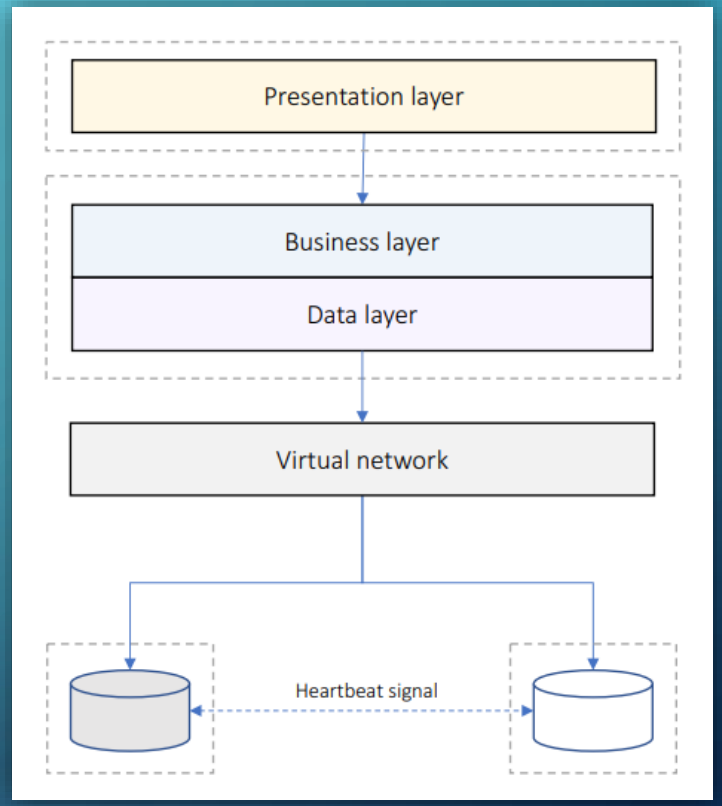
DEPLOY FOR PERFORMANCE

- Business/data layers scale out
- Can detect failed tiers
- Stateless design preferred
- Stateful design requirements:
  - Shared state server
  - Session affinity



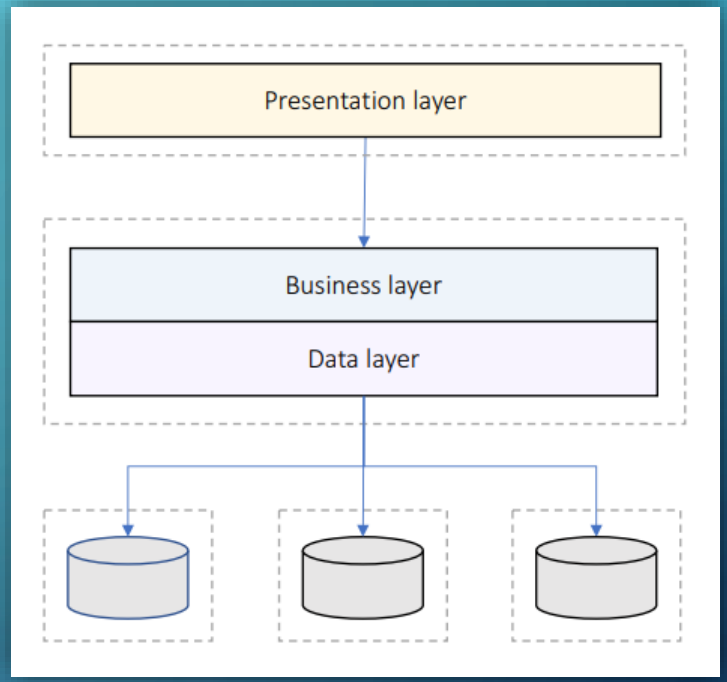
DEPLOY FOR RELIABILITY

- Secondary tier takes over when primary tier fails
- Requires way more hardware
- Synchronization considerations:
  - Sync when secondary tier activates
  - Or allow stale data



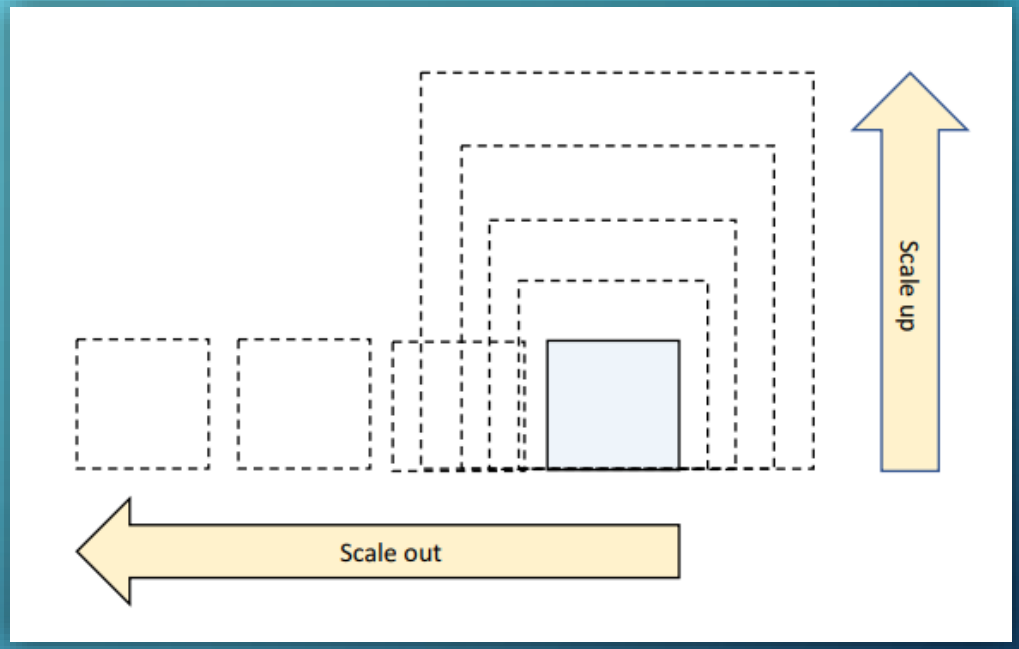
DEPLOY FOR SCALABILITY

- Data replicated on multiple tiers
- Replication breaks consistency and atomicity
- Consistency considerations:
  - Delayed sync in background
  - Or allow stale data
  - Or partition data



SCALE UP AND SCALE OUT

- Scale Up
  - Easy with VMs or containers
  - But limited results
- Scale Out
  - Requires layered design
  - Requires partitioned data
  - Potentially unlimited



QUESTIONS FOR THE LEARNING SYSTEM

- We still haven't thought about performance, availability, and scalability

  1a. What is our tiering strategy? Scaling up or scaling out?

  1b. What is our redundancy and failover strategy?

  1c. Write down a load test and validate the scalability? What can we do to improve it?

  2a. Is our business layer stateless? What is our state management policy?

  2b. Think about all the data in our system. Where can we introduce caching?

  2c. Where are we going to store the cache? How are we going to warm it up?

  2d. Think about the exception policy for each layer. Consider a third-party logging tool?

  3. Do you think you can easily debug problems in your solution? If no, redesign it.

THE ARCHITECTURE DOCUMENT

DEFINITION

- Table of contents
  - The requirements of the system
    - Functional and non-functional
  - The technology stack
  - The architecture diagrams
  - And many more sections
- Do not start development without this document!
- Audience – almost everyone involved
  - Project manager
  - CTO / CEO
  - The Developers
  - The QAs

AUDIENCE

- Management
  - The requirements reflecting the essence of the system
  - Executive summary describing best practices and modern patterns
  - Architecture should be geared towards business goals
  - Management's sections appear first
- Development team
  - Technology stack
  - Modules, services, communication
  - Other technical details
- QA team
  - Testing infrastructure
  - Servers, testing tools, coding

FORMAT OF THE DOCUMENT

- The format of the document is subject to hot debates
- There are standards but nothing is set in stone
  - You can use UML, but it is not necessary
  - Sometimes you may skip it, if the audience is not familiar with it
- Keep it as simple as possible
- Use plain and simple English
  - Keep the technical details for the developers
- Get into the minds of your readers
- Visualize using software you are comfortable with

MAIN STRUCTURE OF THE DOCUMENT

- Background
  - The role of system from a business perspective
- Requirements
  - These dictate the whole architecture
- Executive Summary
  - High-level overview of the solution – for non-technical readers
- Architecture Overview
  - The technical big picture of the solution
- Module Drill-Down
  - The core of the document
  - Detailed and practical instructions for implementing the architecture

BACKGROUND SECTION

- Describes the system from a business point of view
  - System's role
    - For example – solution for the HR team
  - Reasons for replacing an old system
    - For example – too much maintenance and old technologies
  - Expected business impact
    - For example – increasing HR productivity by at least 20%
- Validates the architect's point of view of the system
  - Maximum 1 page
  - If there is an error, you can easily correct it
  - All next sections build upon this one
- Boosts your confidence in front of the management
  - Therefore, you do not use any technical words here

REQUIREMENTS SECTION

- This section should again be maximum 1 page
  - Keep it brief
- Use bulleted lists for describing the requirements
  - Functional – what should the system do
  - Non-functional – what should the system deal with
- Validates your understanding of the requirements
  - Whatever you design, it will solve the actual problem of the customer
  - The architecture is designed against well-defined requirements
- It is a high-level overview of the requirements
  - Do not take the job of a functional analyst ☺

REQUIREMENTS SECTION EXAMPLE

Functional Requirements

1. The system should allow adding, removing and updating employees' data
2. The system will have a sophisticated authentication / authorization mechanism that will ensure only authorized persons will be able to view respective employees' data
3. The system will have a comprehensive reporting mechanism, allowing authorized end users to view various reports about the employees

Non-Functional Requirements

1. The system will have 150 users, with expected load of 20 concurrent user
2. On day one the system will have 100GB migrated from the previous system, and is expected to grow by 400GB annually
3. SLA: The system is allowed a downtime of 5 days annually (planned and unplanned combined)

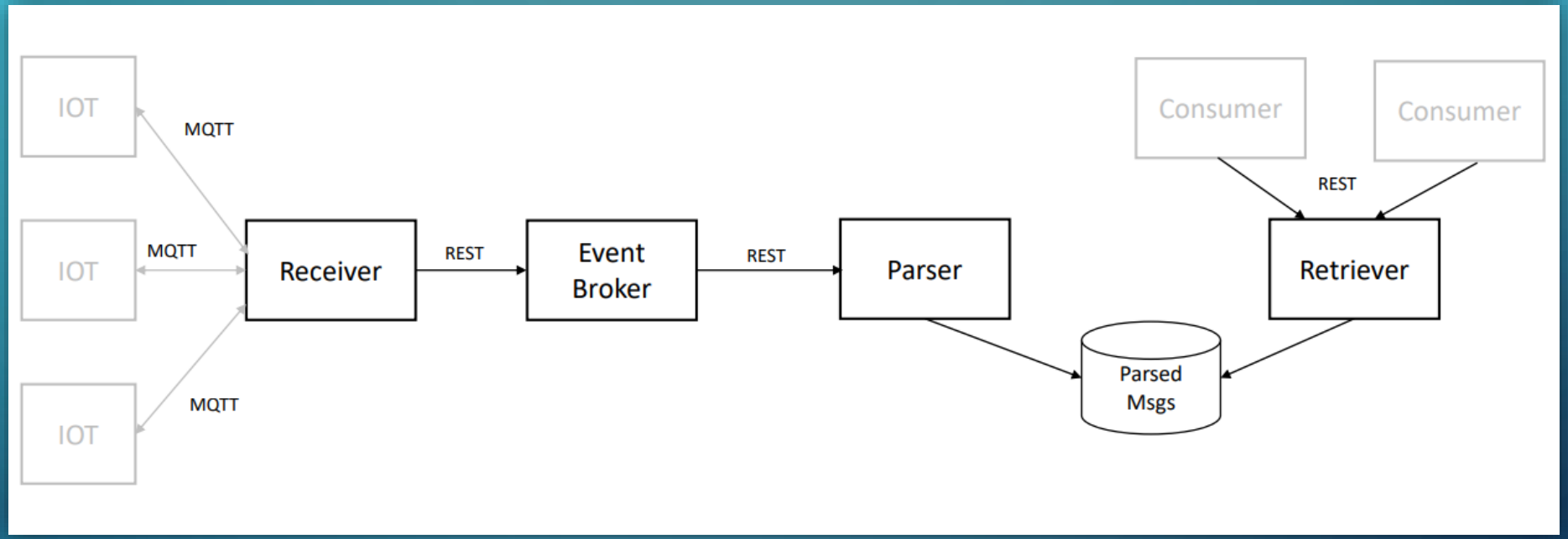
EXECUTIVE SUMMARY SECTION

- This section should be around 3 pages
- This section is again for the management team
  - They will not read your whole document because usually they do not have the time
  - So, you need to impress them and present yourself as good solution architect
  - You need to make them believe that their system is in good hands
- You need to provide a high-level nontechnical view of the architecture
- Get into your readers' mind!
  - A person who does not have a lot of time available
  - He or she should be satisfied as quickly as possible with the presented solution
- Use charts and diagrams (+ well-known technical terms)
- Write this section after you write the rest of the document!

ARCHITECTURE OVERVIEW SECTION

- This section can reach up to 10 pages
- Presents the architecture from a technical point of view
- This section should not deep dive into independent modules
  - It just layers the foundation for these modules
- Include these three subsections:
  - General description - type of the application and major non-functional requirements
  - High-level diagram – show the separate modules and their connections logically
    - Do not mix physical hardware here
  - Diagram walkthrough – describe various parts of the architecture and their roles verbally
    - Use simple words and include the most relevant details
    - Include technology stack here only if you use the same stack in each module

ARCHITECTURE OVERVIEW DIAGRAM EXAMPLE



MODULES DRILL-DOWN SECTION

- This section can have "unlimited" number of pages
- You should add for each module in huge detail:
  - Component's role
  - Technology stack
    - Data store, back-end, front-end, etc.
    - Be extremely detailed here and always include rationale!
    - But only for the first module of a particular technology!
  - Module's architecture
    - The inner architecture of the module
    - What exactly and the module should do and how
    - Include layers, diagrams, design patterns, etc.
    - Describe the API and method names
    - Development instructions – keep it brief

DESIGNING A SOLUTION

OUR MAIN TASKS

- Here are our main tasks as solution architects in a nutshell:
  - Understand the system's requirements
  - Understand the non-functional requirements
  - Identify key scenarios and map baseline modules and layers
  - Select the technology stack
  - Design the architecture
  - Write the architecture document
  - Support the Lead Developer and the team ☺

REAL-WORLD PROJECT

- In our previous sample project, we analyzed the process in a huge detail
  - And it was more of a demo project
- Let us analyze a real-world application
- Our project works with lots of IoT devices
  - Home cameras and thermostats, for example
- Each of these devices has a separate application to control it
- But we want to have a unified view of all our registered ones
- We should collect status information and format the data to visually pleasing dashboard
- This way the customer will know what is going on with all his/her devices

FUNCTIONAL REQUIREMENTS

- For our first version the data is read-only and just visualized
  - Customers cannot update data directly from the application
- Customers and their devices are pre-validated because of security protocols
  - Customers do not need to register in the application
- Functional requirements
  - The Functional Analyst did a good job
  - We should understand the concept of the system
    - Receive status updates from IoT devices
    - Store the updates for future usage
    - Allow the users to query the updates

NON-FUNCTIONAL REQUIREMENTS

- Think for a minute – what are the non-functional requirements of this system?
- What information can influence our architecture?
- What kind of questions we need to ask our customer?
- What we know:
  - Messages are from IoT devices – there should be a huge amount of messages
- What we should ask:
  - Load - how many concurrent messages should the system expect at peak times?
  - Volume - what is the total number of expected message per month?
  - Size – what is the average size of a message?

CLIENT ANSWERS

- After a few days of thinking, the client answers:
  - Maximum 500 concurrent messages at peak times
  - 15 000 000 total number of message per month
  - 300 bytes is the average message size
- Let's do some data volume calculations:
  - 15 000 000 x 300 bytes = 4 500 MB / month
  - 4 500 MB x 12 months = 54 GB / year
  - Almost every database can handle this volume of data easily
  - Usually in data extensive applications – the data can expire
    - But we do not think about data archiving or deleting here
- Data volume doesn't seem to be a problem here

BUT WHAT ABOUT LOAD?

- 500 concurrent messages is a super busy system by any standard
- We can easily add lots of servers and scale out
  - But such solution costs money
- It is a better solution to design the software so that it can handle such load
- There is one more concept we need to think about
- Do we care about losing messages?
- If we think about it – no. If a message is lost, a new one will be send in a couple of seconds…
  - Our system is quite tolerant for message lost
  - Of course, we are not talking about system-wide catastrophe
  - 1 lost message out of 1000 is completely acceptable (99.9%)

USERS

- The next requirement is about users
- How many users will the system have?
  - Total of 2 000 000 users
- How many concurrent users should we expect?
  - No more than 40 concurrent users
  - Users who are actively accessing the server
- Total load calculated – 540 concurrent requests

SERVICE LEVEL AGREEMENT

- Of course, the client expects 100% uptime
- But that is not possible in even the most advanced data centers
- There are a lot of factors and we do not have control over them
  - Hardware, virtualization, network, database servers
  - In such cases – make sure you communicate with the proper people
- When talking with the client, you can define three levels of software SLA
  - Silver, Gold, Platinum
  - You decide the differences but the best one is
    - Fully stateless
    - Easily scaled out
    - Logging & monitoring
  - We have live data here, so we choose the Platinum level

REQUIREMENTS CONCLUSION

- Functional:
  - Receive status updates from IoT devices
  - Store the updates for future usage
  - Allow the users to query the updates
- Non-functional:
  - Data volume: 54 GB annually
  - Load: 540 concurrent requests
  - Message loss: 0.1%
  - Total users: 2 000 000
  - SLA: Platinum level

MAPPING BASELINE MODULES

- Modules are based on the requirements
- We have two separate tasks working with separate entities
  - Receiving and storing status updates
  - Querying these status updates
- We can distinguish two different modules
  - Receiver – to receive the messages
    - It will work on heavy load, so we need to make sure it will not have a thread starvation
  - Info Provider – to provide information to the end-users
- But there is a question here…
  - Do we directly store the messages in a raw format?
  - Or do we validate them first?

MAPPING BASELINE MODULES

- We ask the client, and the answer is:
  - There are 4 types of devices and formats
  - 3 of them are JSON-based and one of them is a plain string (needs parsing)
  - Validation is required
- We now know the receiver has the following tasks:
  - Receive the message
  - Validate the message
  - Parse the message and convert it to a unified format
  - Store the message
- But the receiver is under heavy load, its request processing should be as fast as possible

MAPPING BASELINE MODULES

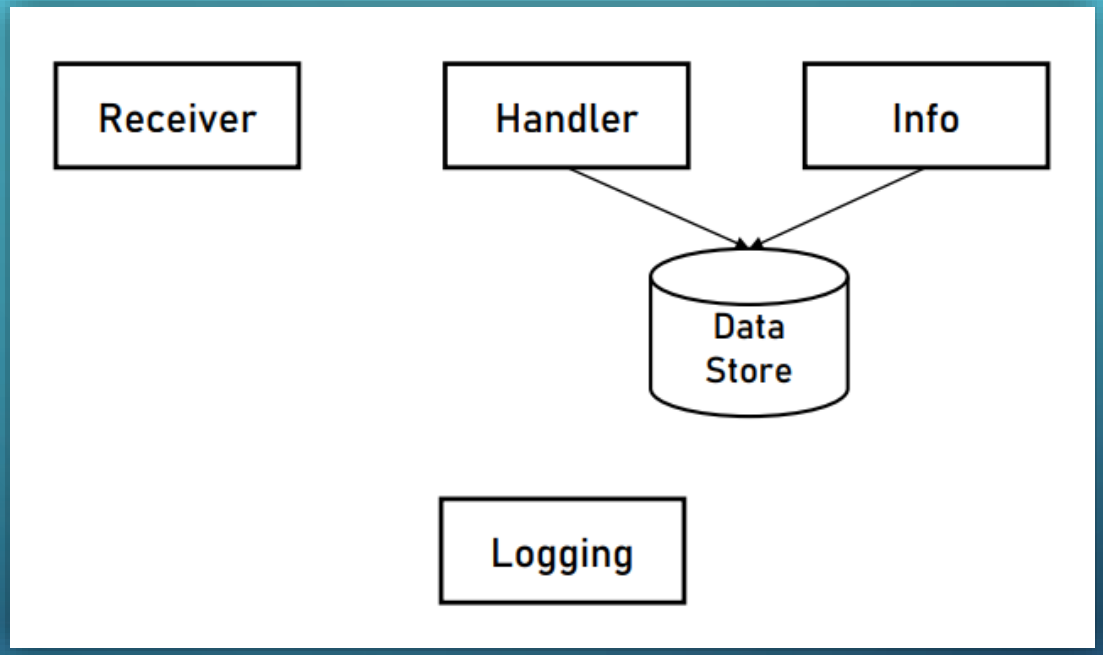
- The parsing task is super important
  - It will make our data independent from its source
  - We will have easier queries for the dashboard
  - Having unified format is a good solution when data is from multiple sources
- So, we decide to leave the receiver do just that – receive
- For the other tasks – we need additional modules
- We can add a Validation module to our architecture
  - But who should do the parsing? It is a good question…
  - Since validation and parsing always go hand in hand, we can do them in a single module
  - There is no justification to split the tasks in different modules for this scenario
  - Because they will require maintenance

MAPPING BASELINE MODULES

- To summarize, we defined the following modules:
  - Receiver – receives messages and dispatches them
  - Handler – validates, parses and stores messages
  - Info Provider – gives us the option to query the messages
- We can now add one more system-wide module:
  - Logging – the central logging service for all the other ones
- Finally, we need a Data Store module
  - It should be shared between the Handler and the Info Provider

CHOOSING MESSAGING METHODS

- Our service modules look like this:



- What kind of messaging is the right one between each one of them?

CHOOSING MESSAGING METHODS

- Let us begin with the Receiver:
  - It receives messages from the IoT devices but how?
  - We ask the client for device specification and we see that the IoT devices use HTTP POST requests – REST API then
- But here is the tricky part. How to send the data to the Handler?
  - It should be done as quickly as possible to relieve the load of the receiver
  - The obvious mechanism here is a message queue
    - Provides us order of execution and reliability
  - If we use REST, we will block the working thread – it will be waiting for a response
    - Additionally, we will have to handle errors and glitches
    - Hurting the performance by a lot!
  - The fire and forget approach is perfect!

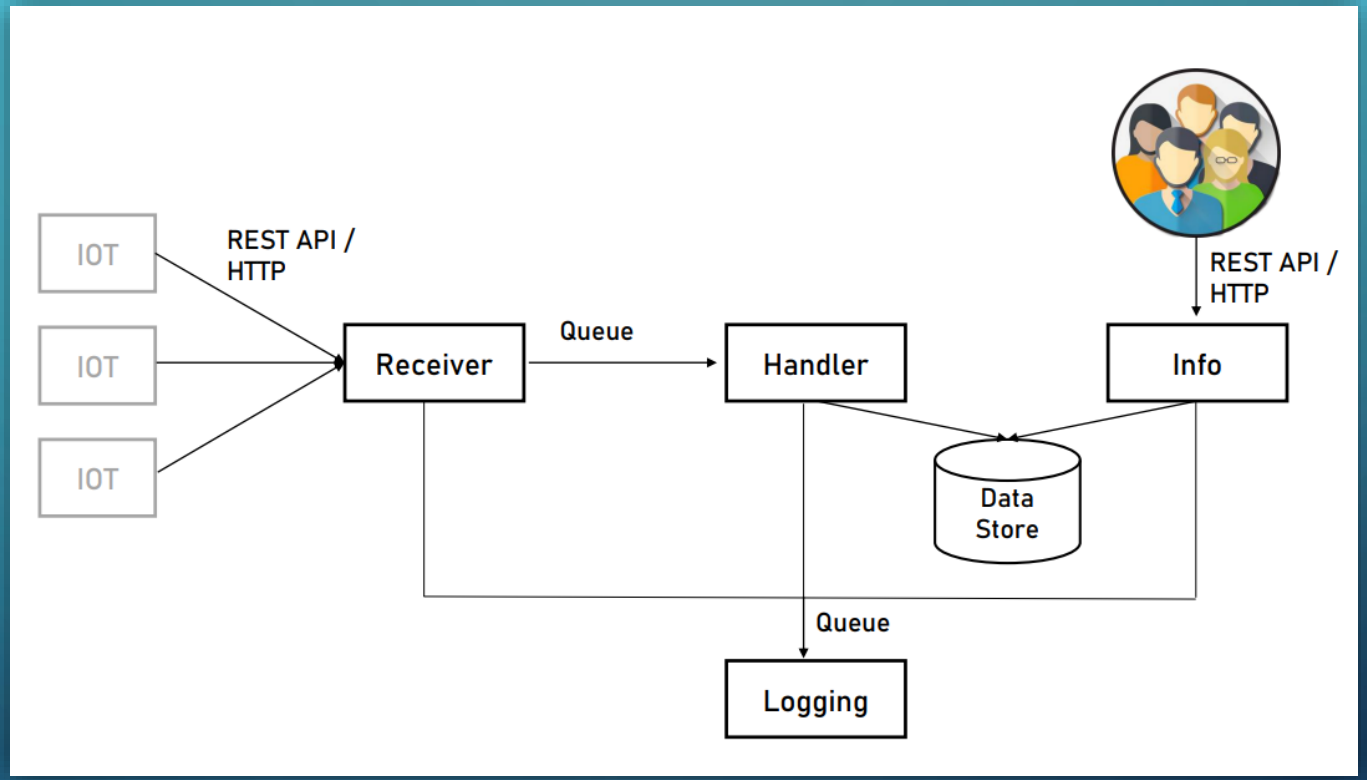
CHOOSING MESSAGING METHODS

- Popular queue options:
  - Self developed
    - Never developer your own message queue except in single-process applications
  - RabbitMQ
    - General purpose message broker
    - Easy to setup
    - Easy to use
  - Kafka
    - Stream processing platform
    - Perfect for data-intensive applications
    - Very complex installation and setup

CHOOSING MESSAGING METHODS

- What about Info Provider?
  - That one is easy – the answer is lies within the end-user requirements
  - How are they using the system?
    - Via a web browser
    - And web browsers use HTTP
  - The Info Provider will implement a REST API for the end-users
- Last, but not least – the Logging service
  - We have large amounts of records – REST API will hurt the performance a lot
  - We could use file polling, but this solution is not cloud compliant and it is hard to control
  - We can use a database or a queue. Which one is better?
  - In this situation – a queue
  - Just because the database will be used as a queue in this scenario

CHOOSING MESSAGING METHODS



DESIGNING THE LOGGING SERVICE

- Our architecture is now ready!
- Just make sure the client's IT support understands the queue mechanisms!
- Our next task is to design the Logging service
  - It is super important, and it should be treated as a first-class citizen
  - Besides that – the other services need it
- These are the steps:
  - Design the application type
  - Design the technology stack
  - Design the architecture

DESIGNING THE LOGGING SERVICE

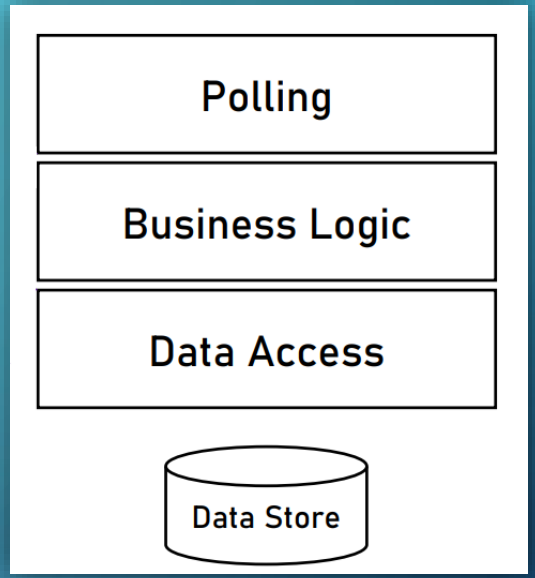
- What are the tasks of the Logging service?
  - Read log records from the queue
  - Handle the log records
  - Save in log data store
- The Logging service is not based on HTTP, so we do not need a web application
- It is not a mobile or desktop application… But is it a console one?
  - Console is a good fit for long-running applications with limited UI
- What about a service? Well, service is also a good fit
- The choice is between a console or a service application
- It depends on personal taste, so both are fine

DESIGNING THE LOGGING SERVICE

- Let us design the technology stack
- Module's code and the data store
  - Access queue's API
  - Store data
  - All technologies provide a solution to these stacks, so we consider our developers' skills - .NET Core and SQL Server, for example
  - Any combination will work here – Java and MySQL, Python and PostgreSQL, etc.
  - For operations with lots of writes and less updates, we may consider a wide column or document database here

DESIGNING THE LOGGING SERVICE

- Let us design the service layers
- Does the classical 3-tier architecture fit?
  - UI/Service interface
  - Business logic
  - Data access
- It fits, but we do not have UI here, so let's change it a bit:
  - Polling
  - Business logic
  - Data access



DESIGNING THE LOGGING SERVICE

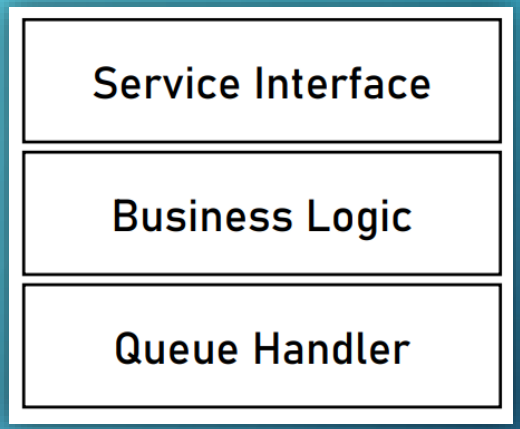
- We need to specify the responsibilities:
  - Polling – polls the queue every few seconds for new log records
  - Business logic – all new log records are sent here for validation
  - Data access – if everything is ok with the log records, this layer saves them to the database
- Other developer instructions:
  - Use dependency injection – Microsoft.Extensions.DependencyInjection for our .NET case
  - Use ORM for data access – Entity Framework Core for our .NET case

DESIGNING THE RECEIVER SERVICE

- What are the tasks of the Receiver service?
  - Receives messages from the IoT devices
  - Sends messages to the queue for the Handler
- What kind of application is this service? It is an easy one – web application
- What is the technology stack?
  - If we chose .NET Core for the Logging service, we need a very good reason to use another stack as it can create a lot of headaches
  - The question here is - does .NET Core support REST API?
    - Yes, it was built for that!
    - As a bonus point – we receive superb performance!
    - There is no reason to change the technology!

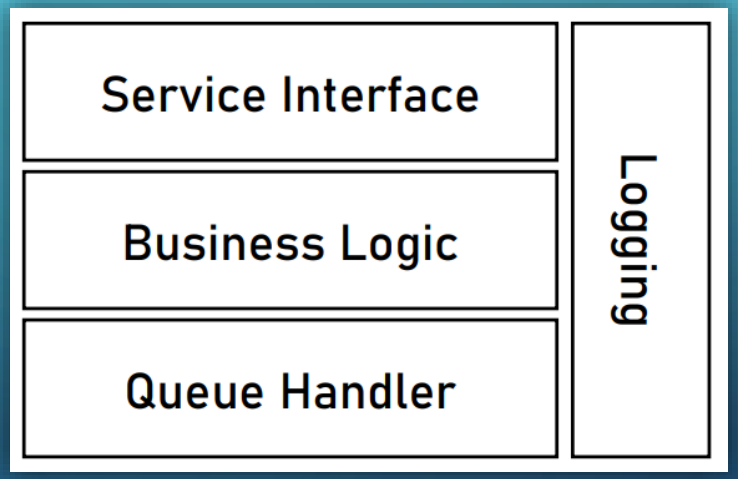
DESIGNING THE RECEIVER SERVICE

- Let us design the service layers
- Does the classical 3-tier architecture fit?
  - UI/Service interface
  - Business logic
  - Data access
- It fits, but we do not have a data store, so let's change it a bit:
  - Service interface
  - Business logic
  - Queue handler
- With these two examples you can see why the layered architecture is so flexible!
  - Just work with interface contracts and you are great!



DESIGNING THE RECEIVER SERVICE

- There is one more thing here
- We need a cross-cutting concern – logging!
  - It should receive logs from every single layer
  - Remember exception handling for the various separations?



DESIGNING THE RECEIVER SERVICE

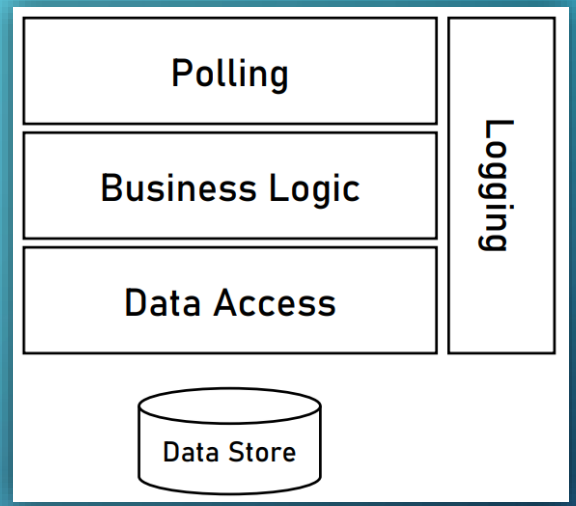
- Let us look back at our non-functional requirements for validation
- Two of the non-functional requirements are relevant here
- Load is 500 concurrent messages. Are we compliant?
  - Yes! Our service is stateless, simple, and easily scaled out in front of a load balancer
- Message lost is 0.1%. Are we compliant?
  - Yes! Our service is super simple and uses a reliable REST API protocol
  - As a side note – for a better uptime, we should consider a cloud technology
- This service is done
  - We can add additional developer instructions about library usage, for example

DESIGNING THE HANDLER SERVICE

- What are the tasks of the Handler service?
  - Polls messages from a queue
  - Validates messages
  - Parses messages
  - Stores messages in data store
- What is the application type?
  - No HTTP and UI are required
- A good choice is a service application
- There are no special requirements so we can use the same technologies
  - .NET Core and SQL Server in our example

DESIGNING THE HANDLER SERVICE

- Let us design the service layers
- As with the Logging service, we do not need a service interface
  - So, our first layer will be named Polling
  - Because it polls data from the queue
- For the Business Logic layer, we can add a plugin mechanism
  - To easily add validation and parsing logic
  - If new devices with new formats are added in the future
- Of course, we need a cross-cutting concern - logging

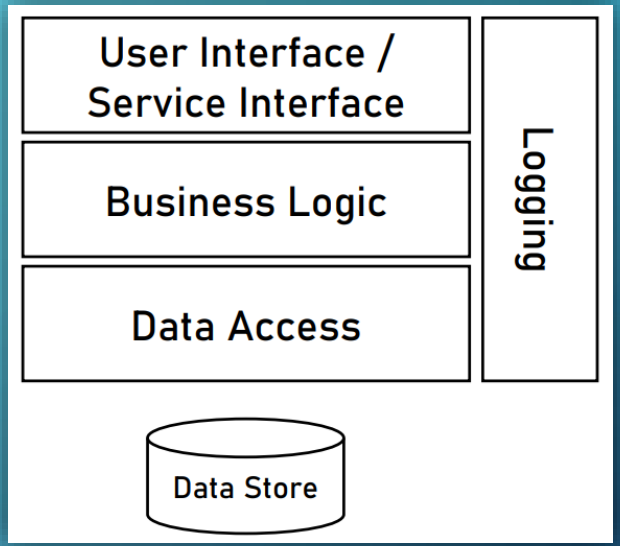


DESIGNING THE INFO PROVIDER SERVICE

- What are the tasks of the Info Provider service?
  - Allows end-users to query the data store
  - The service is responsible only for data retrieval (it does not display the data)
- The application type is easy
  - A web application with REST API
- There are no special requirements so we can use the same technologies
  - .NET Core in our example

DESIGNING THE INFO PROVIDER SERVICE

- Let us design the service layers
- Does the classical 3-tier architecture fit?
  - UI/Service interface
  - Business logic
  - Data access
- Yes, it fits perfectly, just add logging!
- But that is not all, we need to design the API
  - We did not do it in the Receiver service, because the method was dictated by the devices



DESIGNING THE INFO PROVIDER SERVICE

- After discussing with the client, the end-users need the following:
  - Current status of their entire house and for specific devices
  - Past events for their entire house and for specific devices
- Required functionality:
  - Get all the updates for a specific house's devices for a given time range
  - Get the updates for a specific device for a given time range
  - Get the current status of all the devices in a specific house
  - Get the current status of a specific device
- The three main factors for our API are:
  - Path
  - Status code
  - Content

DESIGNING THE INFO PROVIDER SERVICE

- A sample functionality will be:
  - Get all the **updates** for a specific **house**'s devices for a given **time range**

```
GET /api/house/houseId/devices/updates?from=from&to=to
```

  - The time range is not part of the path because it is not an entity
- The status codes should be:
  - 200 OK, if the data is successfully returned
  - 404 Not Found, if the *"houseId"* could not be found
- The response content should contain all the necessary information for the clients

DESIGNING THE INFO PROVIDER SERVICE

Functionality	Path	Return Codes
Get all the updates for a specific house's devices for a given time range	GET /api/house/*houseId*/devices/updates?from=*from*&to=*to*	200 OK 404 Not Found
Get the updates for a specific device for a given time range	GET /api/device/*deviceId*/updates?from=*from*&to=*to*	200 OK 404 Not Found
Get the current status of all the devices in a specific house	GET /api/house/*houseId*/devices/status/current	200 OK 404 Not Found
Get the current status of a specific device	GET /api/device/*deviceId*/status/current	200 OK 404 Not Found

WRITING THE ARCHITECTURE DOCUMENT

- We need to write an architecture document
  - Background
  - Requirements
  - Overall architecture
  - Module drill down
  - Executive summary
- There is one provided in this workshop
  - Contains everything we discussed
  - You may use it as a template, if you like
- We are done with this project! Great job!

WHAT'S NEXT?

THIS WORKSHOP COVERS A LOT!

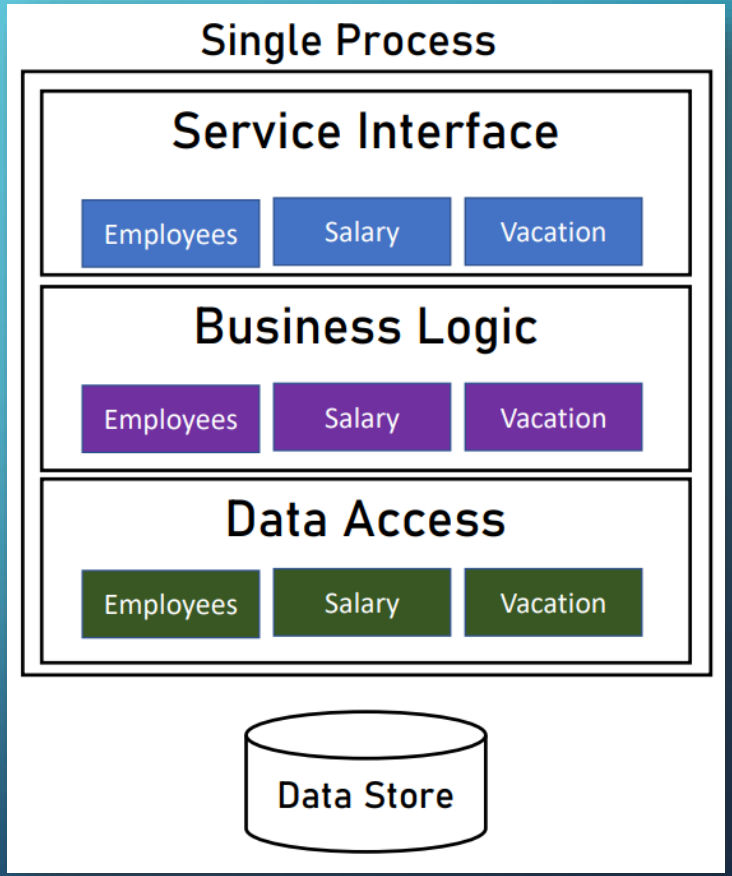
- We now know a lot of patterns
- But the software development world is an ever-changing one
- I am going to mention some other currently used patterns
- You will not use them in every system but just keep them in mind
  - Use them when you think they will be relevant
- Most of these patterns require a separate workshop
  - And I will do one in the future ☺

MICROSERVICES

- The microservices pattern is quite popular today
- Basically, microservices mean logically distributed data
- It comes with a lot of challenges and technical difficulties
- It is a huge trend but do not board that train in a hurry
- The big companies use microservices because they need them
- Consider various other solutions first
- And the best approach is to start with a domain-driven designed solution and extract microservices later when you need them
- There is a bonus video in this workshop about microservices

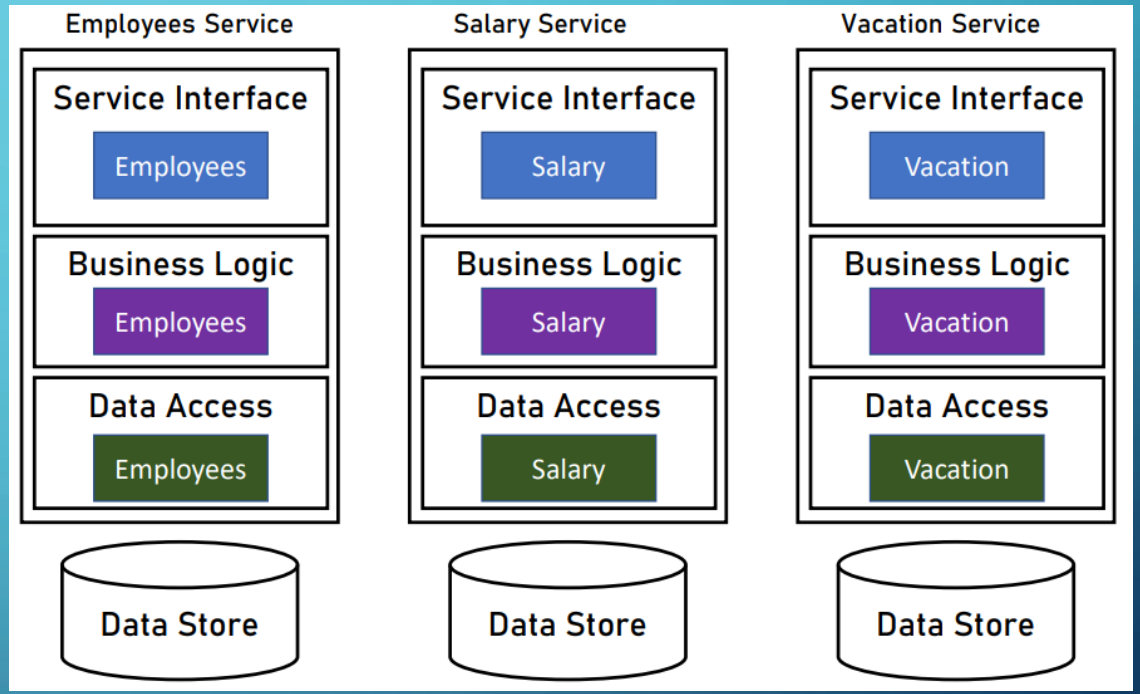
MICROSERVICES

- In monolithic architecture:
  - A single exception may crash the whole process
  - Updates impact all the components
  - Limited to one dev platform
  - Unoptimized compute resources
- With microservices:
  - Each service runs in its own process and does not impact other services
  - Each service can be updated separately
  - Each service can be implemented using different platform
  - Each service can be optimized separately



MICROSERVICES

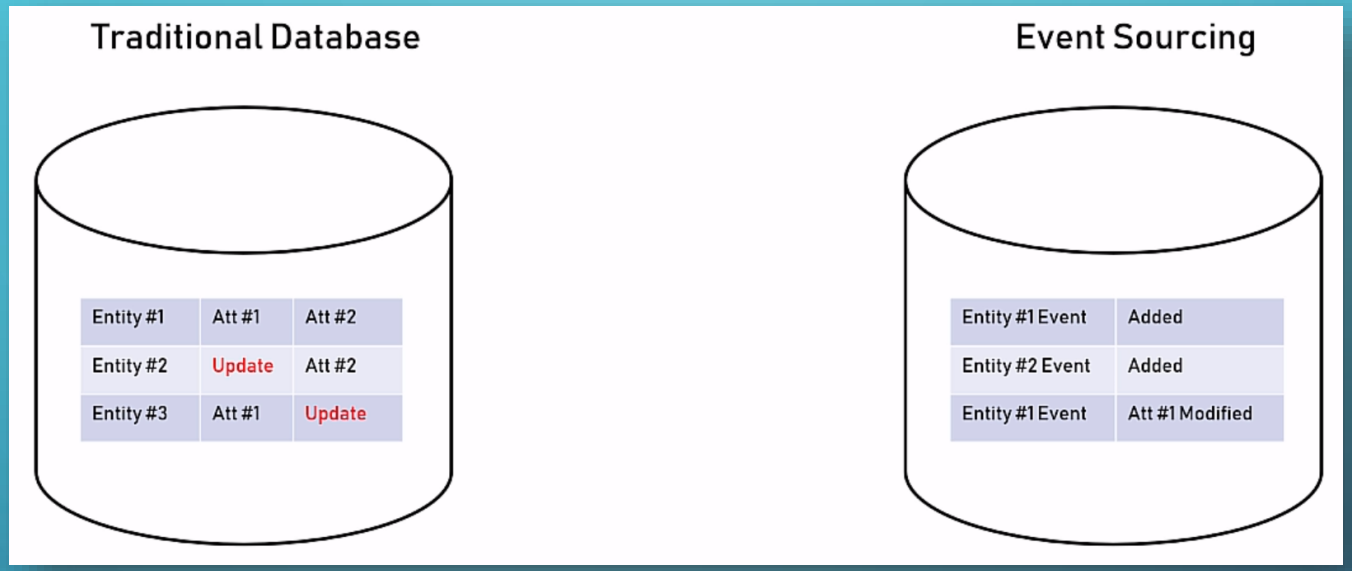
- Additional complexity:
  - Complex monitoring
  - Complex architecture
  - Complex testing
  - Complex deployment
  - Complex data consistency
- You need eventual consistency patterns
  - Like the Outbox pattern
  - There is a bonus video about it



EVENT SOURCING

- Instead of updating entities, you save the update event and what was changed
- Pros
  - Tracing – you can track everything quite easily
  - Data Model – super simple, no complex entities
  - Performance – changing state is add a small record
  - Reporting – a lot of business reports require data history
- Cons
  - No unified view of entities – it takes time to build some visualizations
  - Storage – a lot of storage is required for all the data

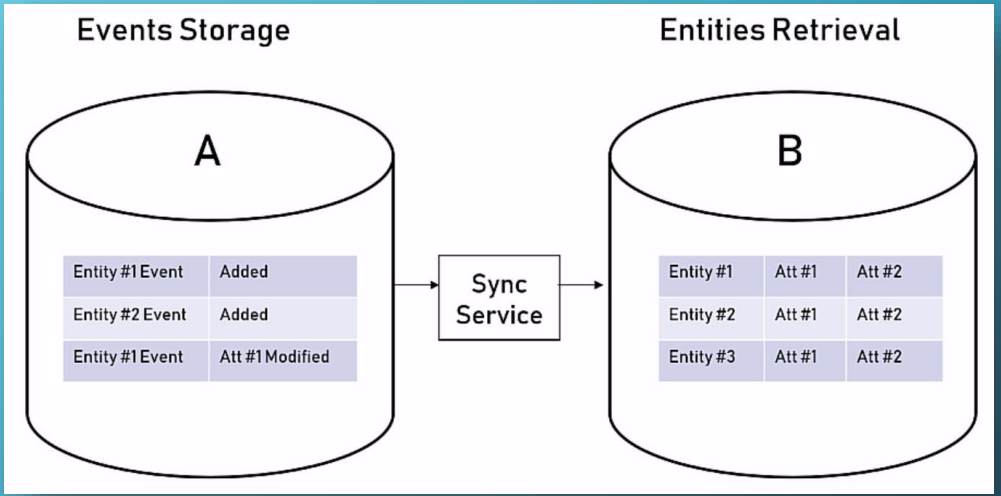
EVENT SOURCING



- Mostly implemented where data history matters
- Think about your bank account statement – you see events, not entities
- Learn more about event sourcing
  - https://martinfowler.com/eaaDev/EventSourcing.html

CQRS

- Stands for Command Query Responsibility Segregation
- It means that we write data to one database
- We read data from another
- And there is a process to synchronize the two databases
- Sounds strange at first, but it complements very well the Event Sourcing pattern
- Useful with high frequency updates that require near real-time query capabilities for the price of complexity



CONTAINERS

- Another great pattern is using containers
  - Docker is the most popular approach
- You wrap your application in a read-only image
- It gives you environment consistency and easier deployment
- Containers often need orchestration
  - Docker Swarm or Kubernetes are great tools
- There is a video lecture about Docker

SERVERLESS COMPUTING

- Emerged with the clouds
- You may exclude the back-end development
- And rely on deployed functions in the cloud
- Allows easier scaling and implementation without an architecture
- It is more difficult to debug
- Azure Functions and AWS Lambda are great providers

SECURITY

- Customer data should be protected
  - And should not be shared with external services or users
- Sensitive data should be protected
  - Payment cards, for example
- Anti-breach mechanisms
- There are a lot of factors regarding security
- It is a huge topic and I plan to create a workshop for it ☺

LEARN BY YOURSELF

- We covered a lot!
- But the topic is endless!
- Google how to become a good software architect!
- Improve yourself daily!
- Work on your soft skills!
  - Excellent communication is as valuable as computer knowledge!
- Keep in mind this field changes very quickly!
  - New design patterns come and go!
  - Frameworks are changing all the time!
  - New libraries are released every day!
  - The whole IT landscape is constantly evolving!

FINAL LECTURE WORDS

EXCLUSIVE WORKSHOP PARTNER – LAB08

- Lab08 offers product development services
- The company works with clients from the Scandinavian market, developing web platforms in various domains
  - Video-based user testing
  - Issuance and verification of digital credentials
  - Marketing through gamification and influencer-based social media marketing
- When a Lab08 employee seeks development - the person can move on to a new product or domain within the company, being able to explore the product area in depth
- Careers: https://lab08.com/careers/

CODE IT UP MENTORSHIP PROGRAM ON PATREON

- Target – junior to regular programmers with 0 to 2 years practical work
- Help with:
  - Becoming better software developers
  - Code improvement in terms of quality and logic
  - Career choices and advancement
  - Interview preparation
  - And more
- Approach – private groups, exclusive lessons, live classes, one-on-one meetings
- From 2021 – project mentorship – DDD, Microservices, CI/CD, Kubernetes, and more
- More information on Patreon: **https://www.patreon.com/ivaylokenov**

LECTURE SUMMARY

- About Code It Up
- About This Workshop
- Why Software Architecture
- The Architect And The Team
- What Makes A Great Architect
- What Is Software Architecture?
- Unified Modeling Language
- Designing Solution Architectures
- Common Technology Stacks
- Architecture Design Patterns
- Choosing The Right Patterns
- Architecture Quality Attributes
- System-Wide Considerations
- Deployment Considerations
- The Architecture Document
- Designing A Real-Life Solution

ANY QUESTIONS?

YOUR TURN NOW!

DOWNLOAD THE RESOURCES FROM HERE AND START THE BOOK:

HTTPS://BIT.LY/SOFTWARE-ARCHITECTURES

ASK QUESTIONS ON SLI.DO:

HTTPS://SLI.DO **#SOFTARCH**

USE THE FACEBOOK GROUP FOR QUESTIONS AFTER THE EVENT:

HTTPS://WWW.FACEBOOK.COM/GROUPS/CODEITUP/