Softwarové inženýrství

5. přednáška

Radek Janoštík

Univerzita Palackého v Olomouci

23. 10. 2025

Outline

- Architektura SW a její návrh
- Procesy návrhů architektury
- Návrhové vzory architektury
- Doporučená četba

Architektura – úvod

- = Rozdělení aplikace na subsystémy, hlavní komponenty
- Určení tvaru a stylu jejich komunikace
- „spojka“ mezi požadavky a implementací
- Je dobré mít v systému nějaký řád, pravidla, rozdělení zodpovědností
- Často opomíjená část – „ve skrytosti“, nedokumentovaná
- Proč architekturu řádně popsat a dokumentovat?
  - Poskytuje *vysokoúrovňový* pohled na systém $\Rightarrow$ lepší komunikace s *účastníky*
  - Usnadní *systémovou analýzu* – snáze splní kritické požadavky na systém
  - Odhalí možnosti pro *znovupoužití* a spolupráci komponent
- Výsledný návrh poskytne návod pro implementaci
  - Rozdělení práce do týmů
  - Způsob nasazení aplikace

Architektura – úvod

- (Dobře) navržená architektura ovlivní (téměř) všechny vlastnosti systému
- Navrhování architektury často probíhá zároveň se zjišťováním požadavků
  - Iterativně může ovlivňovat jedna fáze druhou
  - Neuspěchat návrh architektury
- Návrh architektury ovlivňuje i netechnické stránky projektu
- Volba architektury by měla vzejít z faktů a reálných potřeb
  - „Vždycky to děláme jako vždycky“ – architektura tu není pro architekturu
  - Má usnadnit práci, dodržet řád a **vyřešit problémy**(splnit požadavky)
- Např.: máme požadavek na
  - *výkon* $\Rightarrow$ kritické operace v málo subsystémech, malá komunikace
  - *bezpečnost* $\Rightarrow$ využijeme vrstvenou architekturu. Kritické funkce „uvnitř“
  - *dostupnost* $\Rightarrow$ redundantní komponenty
  - *snadnou údržbu* $\Rightarrow$ malé komponenty, snadno vyměnitelné

Návrh architektury „shora dolů“

- Návrh architektury od abstraktních struktur ke konkrétním
- Od nejvyšší úrovně dolů k nejdetailnějším komponentám
- Často prováděn iterativně, upřesňování detailů
- Vhodný, pokud dobře známe (celou, velkou část) doménu
- Oblíbená forma v korporátech, „enterprise“
- Výhody:
  - Systematické dělení funkcionality $\Rightarrow$ rozdělení práce do souvislých celků
  - Design subsystémů mohou dělat další architekti, snadnější odhady ceny, plánování
  - Vhodný pro velké projekty – snadnější údržba
- Nevýhody:
  - Problém „předesignování“ – *Big Design Up Front*
  - SW je komplexní, je obtížné nahlédnout na celý systém bez konkrétních problémů
  - Neodhalení drobných problémů (ovlivňující zbytek)
  - Těžko se pak dělají modifikace
  - Více týmů, horší sdílení znalostí a znovupoužitelnost kódu(redundance)

Návrh architektury „Zdola nahoru“

- Obrácený přístup – od konkrétních komponent po nejvyšší úroveň abstrakce
- Architektura (sub)systému často „vyvstane“ během práce na komponentách
- Nepotřebuje úplnou znalost domény – v jeden čas soustředění na jednu věc
- Výhody:
  - Tým se soustředí na jednu věc, důležitou v dané situaci
  - Funguje s agilním vývojem
  - Nehrozí „předesignování“
  - Může se brzy začít programovat (⇒ delší testování)
  - Snadnější znovupoužitelnost hotového kódu
- Nevýhody:
  - Nevyhneme se refaktorování kódu⇒ nákladnost změn
  - Architektura nemusí „být dobrá“ ⇒ horší udržitelnost
  - Těžší plánování (nevíme kolik nás toho ještě čeká)

Jaký postup tedy volit?

- Shora dolů:
  - Velké, komplexní projekty
  - Enterprise software (pro korporace, státní správa)
  - Velký tým, více týmů
  - Dobře známe doménu
- Zdola nahoru:
  - Malé, ne příliš komplexní projekty
  - Non-enteprise sw (zákazník)
  - Malý, jediný tým
  - Dobře neznáme doménu
- Každý extrém je špatný – někdy je vhodné udělat kombinaci
  - Začneme shora dolů (přemýšlet), po čase půjdeme zdola („programovat“)
  - A naopak
  - Případné iterace

Faktory ovlivňující architekturu

1. Účel návrhu – Ne vždy vytváříme architekturu primárně pro funkční systém
   - Modifikace současného
   - Pouze pro účely nabídky (státní zakázky, velké projekty)
   - Pro účely prototypu, *Proof-of-Concept*
2. Primární funkční požadavky – kritická funkcionalita systému
   - Funkční požadavky s nejvyšší prioritou
   - Ty méně podstatné by na architekturu měly mít malý vliv
3. Požadavky na emergentní vlastnosti systému
   - Vhodná architektura je „pohlídá“
4. Podmínky (zákazníka)
   - Rozpočet
   - Doba a způsob dodání (a provozu)
   - Využití specifické technologie
   - Použití již implementovaných systémů
   - Licence zdrojových kódů

Jak navrhnout architekturu

- Je zbytečné znovu vymýšlet, jak navrhovat architekturu
  - „Proč znova vynalézat kolo?“
- Nabízí se využít *návrhových vzorů*
  - Jak architektur samotných
  - Tak modely procesu návrhu
- Existuje několik osvědčených konceptů, které při návrhu můžeme použít
- Existuje spousta návrhových vzorů samotných architektur

Návrhové koncepty – referenční architektura

- = Šablona pro architekturu, která „je nejlepší“ v dané doméně
  - Doporučené struktury, elementy a vztahy
  - Ověřené řešení
- Znovupoužití referenční architektury urychlí vývoj, minimalizuje chyby (zkušenosti z minula)
- Často si firmy vytvoří svou referenční architekturu, kterou přizpůsobují dalším zakázkám
  - Přizpůsobení rovněž v iteracích, ale rychlejší
- Např.: Webový klient - API - mobilní aplikace pro eshop
  - Specifické části se budou lišit v závislosti na eshopu
  - Většina principů architektury bude totožných

Návrhové koncepty – nákup komponent / OSS

- Vlastní (naprogramované) řešení nemusí být vždy nejlepší/nejlevnější
- Některé subsystémy existují, je možné je koupit
- Na některé subsystémy „nemáme lidi“ $\Rightarrow$ *outsourcing*
- Vlastní vývoj:
  - Řešení bude „ušité na míru“ projektu, plná kontrola
  - Můžeme později „zdarma“ znovu použít
  - Můžeme mít nejlepší řešení na trhu – konkurenční výhoda
  - Zabere nám to čas (a lidi), nemusí být odladěné řešení
- Nákup:
  - Ušetříme čas, řešení může být vyladěné
  - Nakoupené řešení se může průběžně vylepšovat $\times$ zastarání
  - Může být drahé, nevyhneme se kompromisům
  - Licence, *vendor lock-in*
- Obojí bychom si měli důkladně promyslet a obhájit argumenty

Využití open-source (OSS) řešení

- Vypadá velmi lákavě – může být zdarma, zdrojové kódy k dispozici
- Problém velkého množství alternativ $\times$ jediné rozumné řešení
- Musíme dát pozor na licenci zdrojového kódu
- *Intermezzo:* Kdo někdy řešil licenci zdrojových kódů?
- Komerčně problémová licence: GNU General Public Licence
  - „Nakažlivá“ – jakmile použijeme, musíme dát zdrojové kódy k dispozici
  - Kdo ale zaručí? Z historie: vynucení je velmi komplikované
- Vhodná licence *MIT* (moje oblíbená), a spousta dalších
- Výhody – audit zdrojových kódu, komunita, cena, bughunting, můžeme sami opravit
- Nevýhody – projekt *jednoho muže*(životnost), ideologie ovlivňuje kvalitu, žádná záruka

Dokumentování architektury

- Jakkoliv úžasná architektura je bez dokumentace k ničemu
- Dokumentaci je velmi dobré doplnit diagramy $\Rightarrow$ snadnější orientace
- Při dokumentaci můžete odhalit chyby
  - Nejste-li to schopni rozumně nakreslit, nejspíš je chyba v návrhu
  - Rozdělit, zjednodušit a zkusit nakreslit znovu
- Dokumentaci je vhodné doplnit o zdůvodnění, proč byly jednotlivé elementy takto navrženy
  - Také např. proč nebylo zvoleno jiné řešení
  - Usnadní znovupoužití dané architektury (stane se referenční?)
  - Usnadní komunikaci, minimalizuje „šum“ a domýšlení
- Odrazový můstek pro implementaci

Systematický návrh architektury

- Architekturu bychom neměli navrhovat *ad-hoc*
- Existuje několik systematických modelů, jak při tvorbě architektury postupovat
- Podobně jako u modelů vývoje, jedná se spíše o doporučení
  - Držet se osvědčených metod, poučit se z chyb

Obecný model

- Vznikl srovnáním pěti zaběhnutých modelů $\Rightarrow$ identifikace společných rysů
- Složen ze 3 částí
- *Analýza architektury* – identifikace problémů k řešení
  - Identifikace a prioritizace faktorů
  - Výběr těch, které zohlední architektura
  - Výstup: množina faktorů ovlivňující architekturu
- *Syntéza architektury* – samotný návrh architektury
  - Volba referenční architektury, návrhového vzoru
  - Rozhodnutí zda některé části nakoupit
  - Výstup: navržená architektura, řešící identifikované problémy
- *Zhodnocení architektury* – ověření, jak se nám to povedlo
  - Udělali jsme všechna rozhodnutí dobře?
  - Opět mnoho zaběhnutých modelů (např. konkrétní scénáře)
- Návrh architektury je iterativní proces – na první pokus to většinou nebude bez chyb

Attribute-driven design (ADD) (1/2)

- Systematický iterativní postup návrhu
- Krok za krokem popsáno, co by se v daných iteracích mělo dělat
- Klade důraz na **atributy** kvality softwaru (funkční, emergentní vlastnosti, proces, . . .)
- Zaměřuje se pouze na *syntézu architektury*

1. *Revize vstupů* – vyjasníme si vstupy a řešené problémy
   - Účel návrhu, primární funkční požadavky, emergentní vlastnosti, podmínky vývoje
   - Vyvíjíme nový systém či upravujeme současný?
2. *Stanovení cíle* aktuální iterace návrhu
   - Vybereme konkrétní vstupy, které budeme řešit
3. *Volba elementů*, které se budeme snažit konkretizovat
   - Na základě cíle zvolíme elementy
   - V první iteraci to je celý systém, později konkrétnější subsystémy

Atribute-driven design (ADD) (2/2)

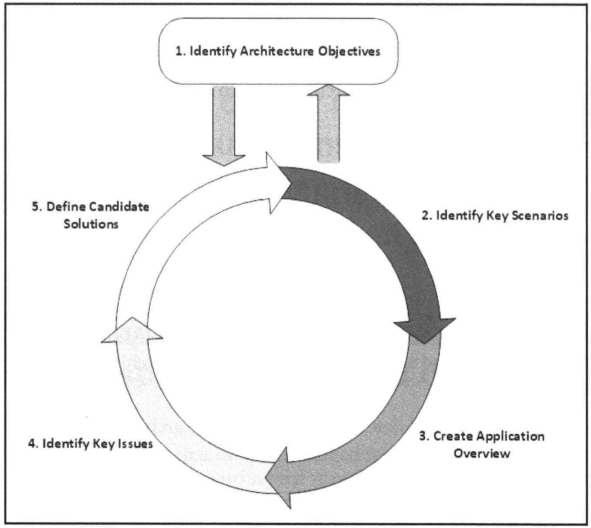
4. *Volba konceptu* pro splnění cíle
   - Budeme vyvíjet sami? Nakoupíme řešení?
   - Využijeme nějakou referenční architekturu?
5. *Stanovení elementů* – určení odpovědnosti
   - Definice rozhraní nově vzniklých elementů
   - Rozdělení odpovědností do potomků rozdělovaného elementu
6. *Rozkreslení, dokumentace* – nákresy můžeme komunikovat s *účastníky*
   - Dokumentujeme, nezapomínáme i na zdůvodnění
7. *Validace architektury* – ověříme správnost současného stavu návrhu
   - Určíme, zda je potřeba další iterace
8. *Iterace (`GOTO 2`) nebo finální návrh*

Technika pro návrh architektury od Microsoftu

- Podobné jako ADD, několik pevně daných fází

1. *Stanovení cílů architektury* – ujištění, že víme, co máme dělat (podobné jako u ADD)
   - Odehrává se pouze jednou
2. *Volba klíčových scénářů* – zvolíme scénáře použití
   - Ne pouze jeden konkrétní $\Rightarrow$ množina důležitých
   - Důležité případy užití, které ovlivní architekturu
3. *Vytvoření náhledu na aplikaci*
   - Určení typu aplikace – okenní aplikace, web, mobil, služba, ...
   - Určení podmínek nasazení – kde bude aplikace běžet
   - Zvolení návrhového vzoru, stylu architektury
   - Určení vhodných technologií k dosažení cíle (framework, databáze, webserver, ...)
4. *Identifikace klíčových problémů* k řešení
   - Konkrétní funkční požadavek, emergentní vlastnost
5. *Návrh kandidáta řešení* – konkretizace, koncová komponenta
   - Integrace kandidáta do architektury
   - Validace, rozhodní k další iteraci

MS technika – obrázek



Architecture-centric design method (ACDM)

- Iterativní metoda, která se snaží najít rovnováhu mezi obchodními a technickými aspekty projektu

1. *Odhalení faktorů*
2. *Vymezení rozsahu projektu* – ujasnění faktorů od *účastníků*
3. *Vytvoření „nástřelu“ architektury* – prvotní elementy a jejich dokumentace
4. *Revize (současné) architektury* – interně i s *účastníky*
   - Ověření důvodů volby architektury
   - Není lepší alternativa?
5. *Rozhodnutí* zda se jedná o finální architekturu
   - Možné i částečné – něco už můžeme implementovat, něco ještě navrhneme lépe
6. *Plán pokusů* – navrhneme, jak můžeme architekturu zlepšit
7. *Provedení pokusů* – vyzkoušíme dané návrhy, zhodnotíme
   - Slibné výsledky (např. vhodné rozdělení na komponenty) zahrneme do architektury
   - `GOTO 4`

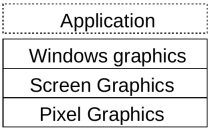
Vzory architektur – úvod

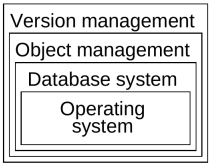
- Někdy též architektonické styly/paradigmata
- Většinu problémů řešil někdo (dávno) před námi
  - Velmi zřídka se dostaneme k novému, unikátnímu problému
  - (Opravdové) problémy většinou vzniknou, až v běžící aplikaci
- Postupně se ustálilo/uchytilo několik osvědčených *návrhových vzorů*
- Pro podobný kontext, problém bude existovat podobné řešení
  - Kostra(=vzor) ale bude téměř stejná
  - Popis abstraktních struktur vedoucí k řešení
- Usnadní a urychlí návrh, jsou již ustálené pojmy „všichni je znají“
- Mohou se aplikovat na celý systém nebo jen na části
- Vždy posoudit vhodnost. Není dobré nadužívat za každou cenu

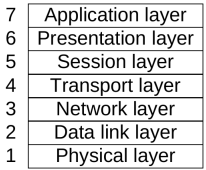
Vrstvená architektura

- Rozdělení systému do horizontálních, na sobě ležících, vrstev (*layers*)
  - Každá je závislá(přímo $\times$ nepřímo) na vrstvách pod ní
  - Každá je nezávislá na vrstvách nad ní
- Dva způsoby vrstvení podle propustnosti:
  - *Uzavřené vrstvy* – implementace **pouze** pomocí nejbližší nižší vrstvy
    - „kratší závislost“
    - Snadnější změny rozhraní (změna přímého souseda)
    - Např.: ISO/OSI, TCP/IP
  - *Otevřené vrstvy* – implementace za pomocí kterékoliv nižší vrstvy
    - Možná menší režie $\Rightarrow$ vyšší výkon
    - Hůře se udržuje (změna může ovlivnit více vrstev)
    - Např.: Grafické systémy

Vrstvená architektura







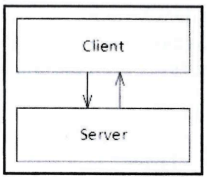
- Specifikace problému (většinou) definuje jen nejvyšší vrstvu
- Spodní vrstva je nejblíže fyzickým zdrojům (knihovny, OS, HW, síť)
- Někdy se používá výraz *patro (tier)* – většinou pokud je logická vrstva na jiném fyzickém zařízení

Vrstvená architektura

- Výhody:
  - (Striktní) rozdělení odpovědností – nezávislost vrstev, každá řeší svou menší část problému
  - Malá závislost mezi vrstvami – Snadnější změny, údržba
  - Velmi známý přístup – není potřeba dlouhého tréninku
  - Snadná testovatelnost (fake vrstvy, mock)
  - Znovupoužitelnost (můžeme vyměnit „jen“ horní vrstvu a máme jinou aplikaci
  - Bezpečnost – můžeme přidávat *firewally* jako mezivrstvy bez změny okolních
- Nevýhody:
  - Hlídání a dodržování nezávislosti vrstev
  - I drobná změna může způsobit změny na všech vrstvách (pole ve formuláři)
  - Složitost komunikace (kód „navíc“)
  - Nižší výkon ($\times$ přeskočení vrstvy)
  - *Vícepatrový systém* bude přirozeně dražší (více strojů)

Client-server

- = (vysokoúrovňová) dvouvrstvá architektura – též ale dvoupatrová
- Obousměrná komunikace mezi vrstvami, úzce spjaté



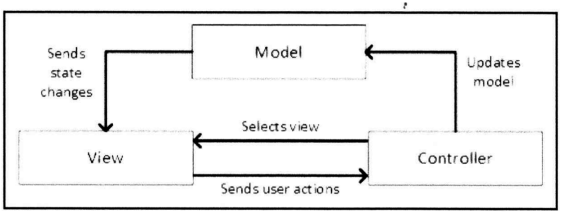
- *Klient* – Většinou pouze uživatelské rozhraní
- *Server* – Logika aplikace, persistentní vrstva
- Na klientovi by neměla být žádná logika aplikace (nepořádek kde dělat změny)
- Server by měl vždy validovat data z klienta (=nedůvěra)

Systémy řízené událostmi (Event Driven)

- *Událost* = informace o změně stavu systému, která může zajímat jiné části systému
  - Např.: Zadání objednávky, uložení souboru, klik myši, stisk klávesy, ...
- Distribuovaný, asynchronní vzor, kde subsystémy *vysílají a reagují na* události
- Události řídí *dispečer* – spravuje *frontu událostí* a zajišťuje její doručení
  - Poskytován OS, frameworkem či běhovým prostředím
  - Spustí *callback* – metoda u „adresáta“ události
- Vzor *publish-subscribe* – zdroj událostí (*publisher*) se nestará, komu je událost určena
  - Jednoduše událost *odvysílá (broadcast)* do „éteru“
  - Kdo má o událost zájem – *odběratel (subscriber)* si ji „vyslechne“ a může reagovat
- Použití: GUI, sledování změn v DB, Android, ...
- Velké množství typů zpracování událostí (prostředník, broker topologie, ...)

Model-View-Controller (MVC)

- Rozšířený vzor pro GUI a webové aplikace(ASP MVC, Spring MVC)
- Striktní rozdělení odpovědností do tří částí



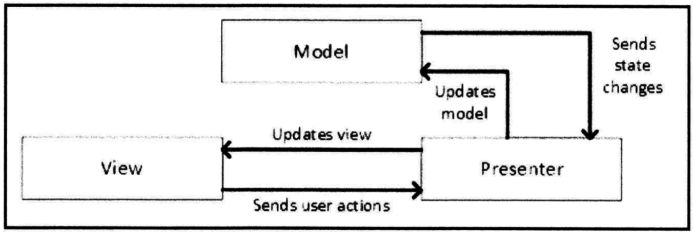
- *Model* – spravuje aplikační data, udržuje jejich stav, persistence
  - Nezávislý na ostatních
  - Poslouchá příkazy kontroleru – je pasivní
- *View* – řeší zobrazení/prezentaci aplikace
  - Vidí/interakce s uživatelem, zadává akce

Model-View-Controller (MVC)

- *Controller* – obsahuje aplikační logiku
  - Přijímá akce od view
  - Zadává příkazy pro model
  - Volí view, který se má zobrazit
- Výhody
  - Striktní rozdělení odpovědnosti
  - Jednodušší testování
  - Znovupoužití komponent
  - Specializace vývojářů (frontend $\times$ backend)
- Nevýhody
  - Malý tým, nerozdělený $\Rightarrow$ *full-stack* vývojáři
  - Komunikace model$\rightarrow$ view nemusí být žádoucí(změna modelu=změna view)

Model-View-Presenter (MVP)

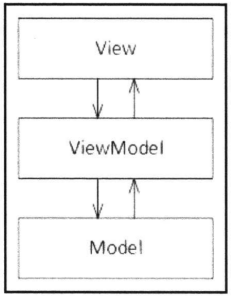
- Varianta MVC – zakázána přímá komunikace model→view



- *Model* – spravuje aplikační data, udržuje jejich stav, persistence
- *View* – řeší zobrazení UI a dat
  - Odesílá uživatelské akce do presenteru
  - Poskytuje rozhraní pro presenter, pasivní prvek
- *Presenter* – interakce s ostatními
  - Každý view má presenter, většinou vazba 1:1
  - Upravuje model, zpracovává data pro view

Model-View-ViewModel (MVVM)

- Podobné jako MVC i MVP (rozdělení odpovědností)
- Striktní oddělení uživatelského rozhraní
- Interakce mezi View a ViewModel pomocí *data bindingu*
- Desktopové, mobilní, webové aplikace



Model-View-ViewModel (MVVM)

- *Model* – spravuje aplikační data, udržuje jejich stav, persistence
  - Ale také aplikační logika
- *View* – uživatelské rozhraní
  - Aktivní – řeší si vlastní, události (jak co zobrazit)
  - Neudržuje stav – přeposílá akce ViewModelu
  - Velká část komunikace – *databinding*, málo kódu
- *ViewModel* – veškerá zobrazovací logika
  - Udržuje stav UI
  - „Nezná svůj View“
  - Uživatelské akce překlápí do Modelu

Service Oriented Architecture (SOA)

- Rozdělení systému na (mírně závislé) spolupracující služby, které dohromady plní požadavky
- *Služba* = část systému, která řeší ucelenou funkcionalitu a poskytuje ji ostatním
  - Zapouzdřují funkcionalitu, poskytují rozhraní
  - Různé velikosti (od komplexních po maličké služby)
- Služby často fyzicky odděleny
  - Samostatný proces – IPC
  - Oddělený stroj – síťová komunikace
  - Webové služby
- Výhody
  - Oddělení logiky a konkrétní technologie – např.: služby v jiných jazycích
  - Často *federalizované služby* – napříč organizacemi
    - ⋆ Známe nějaký příklad federalizované služby?
  - Možné mít vice dodavatelů (není vendor lock-in)
  - Kompatibilní s agilním vývojem

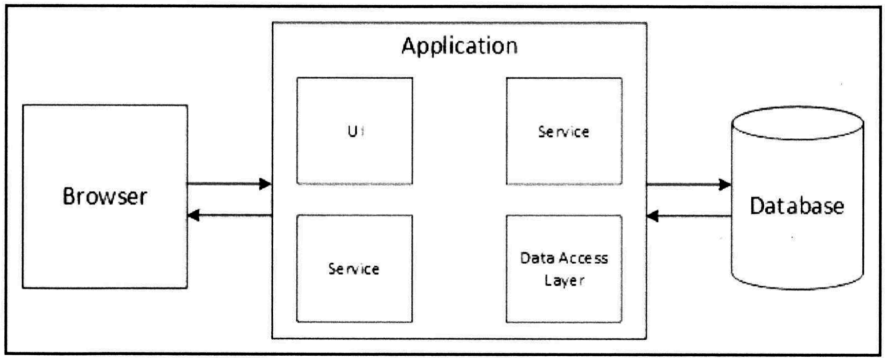
Service Oriented Architecture (SOA)

- SOA nemá pouze výhodu, přináší s sebou i komplikace/výzvy (jsou to nevýhody?)
- Standardizace poskytované služby – definice komunikace služeb
  - Jakým způsobem dát vědět, které služby kdo poskytuje?
- (Částečná) nezávislost služeb – jak moc být nezávislý?
  - Výhody i nevýhody
- Znovupoužitelnost – Jak rozumně navrhnout službu, abychom ji mohli bez modifikací použít jinde?
- „Bezstavovost“ (statelessness) – Udržování stavu je „drahé“
  - Na stejný požadavek, stejná odpověď

Monolitická architektura

- Software je ucelený, pevně svázaný
  - Různé odpovědnosti (Ui, autorizace, logování, databáze) možná odděleny, ale v jednom celku
- Nízká režie, vyšší výkon
- Vhodné u menších aplikací
- Těžko se škáluje jinak než zprovoznění více (různých) instancí
- Hůře udržitelné, jen načtení projektu a kompilace zabere spoustu času
- Menší variabilita technologií

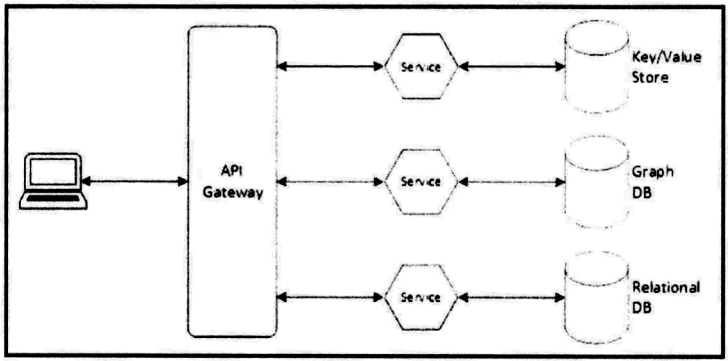
Monolitická architektura



Architektura mikroslužeb

- Systém rozdělen na velmi malé, autonomní služby *(microservices)*
- Nezávisle verzované, technologie, umístění
- Každá mikroslužba pro ostatní jako *black box*
- Velmi vhodná pro velké projekty
- Možnost distribuce zátěže
- Častá komunikace pomocí HTTP protokolu
- Dnes velmi oblíbené

Architektura mikroslužeb



Doporučená literatura

- Ian Sommerville. Software Engineering (10th Edition). Pearson, 2015. ISBN: 978-93-325-8269-9.
  - Chapter 6 – Architectural design
- Joseph Ingeno. Software Architect's Handbook. Packt, 2018. ISBN: 978-1-78862-406-0
  - Chapter 5 – Designing Sofware Architectures
  - Chapter 7 – Software Architecture Patterns