В съвременния цифров свят изграждането на уеб сайтове изисква не само техническа експертиза, но и стратегическо мислене относно софтуерната архитектура. От ключово значение е да се разберат и приложат добрите практики при проектирането на уеб сайтове, както и да се избегнат лошите решения, които могат да доведат до нежелани последици. Този реферат ще проучи някои от най-важните аспекти на изграждането на уеб сайтове и софтуерната им архитектура, представяйки как добрите практики могат да допринесат за успешното функциониране и използване на уеб сайтове, както и как лошите практики могат да създадат пречки и проблеми за потребителите и разработчиците. Чрез анализ на конкретни случаи и примери, ще се разгледат как различни подходи към софтуерната архитектура могат да повлияят на функционалността, ефективността и сигурността на уеб сайтовете.

Грешки в дизайна на приложения

- **Липсата на ясна обратна връзка** Липсата на ясна обратна връзка е един от основните принципи за подобряване на употребата на приложението. Потребителите трябва да бъдат информирани за текущото състояние на системата, как са били интерпретирани техните команди и действия. Добрата обратна връзка дава на потребителите много информация, като например дали бутонът, който са натиснали, е бил правилно интерпретиран от системата и дали сега системата ще извърши действие. Обратната връзка е особено важна, когато приложението е в режим на редактиране на съществуваща информация. Правилната обратна връзка може да предаде на потребителите обхвата на редактирането и може да бъде реализирана по различни начини, като използването на различен фон за идентификация на текущата редактируема област. Вариант на липсата на обратна връзка е когато системата не уведомява потребителите, че отнема дълго време да завърши действие. Ако не може да се спазят препоръчаните времеви лимити за отговор, потребителите трябва да бъдат информирани за напредъка с индикатор за прогрес. Ако командата отнема между 2 и 10 секунди, се показва анимация за изчакване, като например "въртящ се кръг". Ако командата отнема повече от 10 секунди, се показва лента за прогрес, предпочитано като индикатор за проценти (освен ако не може да се предвиди колко работа остава да се свърши).
- **Непоследователност** Непоследователността в дизайна на приложения може да предизвика объркване у потребителите. Разлики в думите или командите за едно и също действие, разполагането на бутони за една и съща функция на различни места и непоследователното поведение на UI елементи могат да доведат до затруднения. Този вид несъгласуваност може да бъде особено изтощителен за потребителите, особено когато става въпрос за сложни

приложения като AutoCAD. В случаи като тези, когато функционалността е инконсистентна и скритите настройки пречат на потребителите да разберат защо нещо работи или не работи, дори опитните потребители могат да се сблъскат с големи затруднения.

- **Неясни съобщения за грешки** Грешките в приложенията често остават недоизяснени, което затруднява потребителите. Тези съобщения за грешки често просто казват, че нещо не е наред, без да обясняват защо и как потребителят може да поправи проблема.В миналото, дори приложенията за настолни компютри са предоставяли повече информация за грешките, въпреки че често на технически език. Информативните съобщения за грешки не само помагат на потребителите да решат текущите си проблеми, но могат също така да обясняват проблема ясно и съдържат информация за това как да се преодолее.
- **Липсващи стойности по подразбиране** Стойностите по подразбиране улесняват потребителите по няколко начина. Те ускоряват взаимодействието, като освобождават потребителите от необходимостта да посочват стойност, ако предварителната ги удовлетворява. Те също така могат да спестят значителни усилия на потребителите при повтарящи се задачи като попълване на една и съща форма многократно. Предварителните стойности имат значение и за полета с падащо меню, където предоставянето на конкретна стойност по подразбиране (предпочитано най-често срещаната) може да позволи на част от потребителите да не взаимодействат с менюто. С числови полета форми, ако потребителите се отклоняват малко от общо приетата предварителна стойност, можете да използвате стъпало, което позволява на потребителите да коригират числото без да го въвеждат (но все пак да им позволяват да въведат различна стойност, ако желаят).
- **Нестандартни иконки** Рядко иконите могат да се разберат сами по себе си. Дори иконите, които може да изглеждат универсални (като менюто "хамбургер"), не са толкова познати на потребителите, както повечето практикуващи в областта на UX биха очаквали. Вероятността, че потребителите ще разберат какво означават нестандартни икони във вашето приложение, е много малка. Повечето икони, освен ако нямат текстово описание до тях, ще бъдат трудни или невъзможни за разбиране от потребителите.
- **Прекомерна употреба на модални прозорци** Много приложения използват модални прозорци за изпълнение на взаимодействия с данни - редактиране на съществуващ елемент, добавяне на нов елемент, изтриване или дори четене на допълнителни подробности за елемент. Модалните прозорци се появяват отгоре на текущата страница и обикновено фона е затъмнен (предполагайки, че затъмняването ще намали отвличанията и ще помогне на потребителите да се съсредоточат върху текущата задача). За съжаление, този

избор на дизайн намалява контекста за потребителите, като покрива информация, към която те може да желаят да се обръщат, докато попълват формата.

- **Безсмислена информация** Дълги низове от букви и цифри, като автоматично генерирани идентификатори в база данни, често се използват за идентифициране на елемент в приложение. Тези низове са напълно безсмислени за потребителите, но често се показват на видно място като първа колона на таблица, като принуждават хората да преминават през тази първа колона, за да намерят информацията, която ги интересува.
- **Неправилна подредба на функционалности** Поставянето на действия като "Запази" до действия, които унищожават работа като "Отхвърли", е обичайно решение за дизайн, което често причинява много неудобства на потребителите.

Грешки в информационната архитектура (IA)

- **Липсваща структура** Най-забележимият структурен проблем е, когато дизайнерите третират сайта като едно голямо "блато" без организационен принцип за отделните елементи. Тази грешка е често срещана на новинарски сайтове и сайтове за електронна търговия на базата на каталози, където всеки елемент (статии и продукти) се третира като самостоятелна единица без връзки със свързани елементи. Не е чудно, че потребителите напускат тези сайтове толкова бързо.
- **Търсенето и структурата не са интегрирани** Потребителите често проявяват поведение, доминирано от търсене. Това обаче не означава, че търсенето е всичко, от което се нуждаят.Те често се нуждаят да бъдат навигират около техния обект на търсене. Разбира се, локалната навигация работи само ако сайтът има подходяща структура. (вижте грешка #1). Дизайнът трябва също да излага локални опции на потребителите.Употребата на SERP (страница с резултати от търсене) се подобрява, когато всеки резултат от търсене показва своето местоположение в структурата на сайта. Външните търсачки като Google не винаги могат да направят това, защото не знаят структурата на сайта или кои навигационни измерения са най-релевантни за общите задачи на сайта. Но вие познавате структурата на вашия сайт и следователно трябва да включите информацията на вашите собствени SERP-и. Този проблем се влошава от друга обща грешка: дизайните на навигацията, които не показват текущото местоположение на потребителя.
- **Изключителна полийерархия** В сравнение с реалния свят, в онлайн средата един от плюсовете е, че артикулите могат да бъдат разположени на множество места. Понеже уебсайтовете могат да организират продуктите и другото съдържание по множество критерии, те помагат на потребителите да намерят свързано съдържание и предоставят възможност за филтриране на голямо

разнообразие от продукти в лесни за управление списъци, които могат да удовлетворят основните изисквания на потребителите. Това всичко е добро, но множествената полийерархия може лесно да се превърне във фактор, който влошава положението. Вместо да отделят необходимото време за разработване на няколко интуитивни и логични категории на високо ниво, екипите често бързат и създават множество слаби категории, в които продуктите се изброяват многократно. Какво е въздействието върху използваемостта? Потребителите губят твърде много време, занимавайки се с категории от високо ниво, а след това се объркват, когато забележат елементи, които се появяват на различни места ("това едно и също ли е?"). С твърде много възможности за класификация и твърде много структурирани измерения, потребителите се налага да мислят повече, за да продължат напред. Многото опции също карат хората да се съмняват в информацията, която предоставят. Този липсващ потребителски комфорт може да се отрази отрицателно на крайния резултат.

- **Подсайтовете или микросайтовете са недостатъчно интегрирани с основния уебсайт** Микросайтовете се разпръскват в интернет като останки от предишни маркетингови кампании. В началото един такъв микросайт може да изглежда като добра идея, особено когато става въпрос за нов продукт, но с течение на времето той може да подкопае цялостната ви онлайн стратегия и да разреди присъствието ви в мрежата. Уеб дизайнът е нещо, което трябва да се мисли в дългосрочен план. Обикновено е най-добре да се откажем от независимите микросайтове и да включим новата информация в подсайтове, които са част от основния уебсайт. Освен това, много уебсайтове не представят подсайтовете си адекватно в търсачките, което често води до напълно игнориране на микросайтовете.
- **Липса на ясни опции за навигация** Най-голямата грешка може да бъде отсъствието на навигация, но това е рядкост. Все пак, всяка функция, която потребителите не могат да забележат, по същество може да се счита за несъществуваща; Следователно, невидимата навигация е практически толкова лоша, колкото и липсата на такава. Необходимо е разкриването на навигацията да не е предизвикателство: Направете я постоянно видима на страницата. Също така, трябва да избягвате ефекта "банерна слепота", когато или самата навигация изглежда като банер, или е позиционирана до елементи, които могат да бъдат възприети като реклама и по този начин потребителите я игнорират. Дори и да е налична на екрана, навигация може да бъде невидима, ако потребителите не я забележат.
- **Неконтролируеми елементи за навигация** Обикновено, всичко, което се движи и се отклонява, намалява използваемостта на уебсайтовете; когато навигационните елементи се движат, докато потребителите се опитват да намерят пътя си, това е изключително неудобно. Потребителите

трябва да се фокусират върху високо ниво проблема - къде да отидат, а не как да манипулират GUI (графичен потребителски интерфейс).

- **Непоследователна навигация** Целта на навигацията е да помага на потребителите, а не да бъде сама по себе си пъзел. Потребителите трябва да могат незабавно да разберат как работи и да я използват във всяка част от уебсайта. За съжаление, много уебсайтове променят своите навигационни елементи, докато потребителите се движат из сайта. Опциите идват и си отиват, като правят потребителите да се чувстват загубени. "Как да върна тази опция в менюто?" си мислят, след като я видят преди няколко страници. Глобалната навигация може да не е най-популярният елемент на уебсайта, но нейната устойчивост има ключово значение.
- **Прекалено много навигационни техники** Всяка техника за навигация има своето приложение в определени типове уебсайтове. Но ако се опитват да се използват всички, няма да се получи сумата от предимствата на всяка от тези техники. Вместо това, резултатът ще бъде бъркотия.

Какво е добра софтуерна архитектура?

Много трудно е да се определи каква е добрата софтуерна архитектура. Няма истински верен отговор на този въпрос. Това, което може да се каже, е че тя прави продуктите по-евтини за разработка и поддръжка. В общи линии, аспектите на добрата архитектура са всички взаимосвързани и зависят един от друг. Например:

- Софтуерната архитектура трябва да е здрава и лесна за поддръжка, когато открием грешки.
- Трябва да има домейни, които почти всички членове ще разберат.
- Трябва да бъде гъвкава, разширяема и ползваема в дългосрочен план.
- Трябва да е възможно адаптирането към изискванията.
- Нужна е висока мащабируемост на капацитета.
- Не трябва да се открива повторение в кода.
- Рефакторирането трябва да бъде лесно.
- Трябва да реагира положително на промените- при добавяне на функционалности производителността не трябва да намалява.

Много хора вярват, че само като погледнете крайния продукт можете да знаете, че имате добра софтуерна архитектура. Това определено не е вярно. Въпреки това, като погледнете продукта, можете да откриете "признаци" за добра архитектура. Отново, това не означава, че ако следните точки се виждат в софтуера, автоматично имате страхотна софтуерна архитектура.

Модели на софтуерна архитектура. Многослойна архитектура

Описание на модела

Компонентите в рамките на слоестия архитектурен модел са организирани в хоризонтални слоеве, като всеки слой изпълнява специфична роля в рамките на

приложението (напр. логика на представяне или бизнес логика). Въпреки че моделът на многослойната архитектура не уточнява броя и типовете слоеве, които трябва да съществуват в шаблона, повечето многослойни архитектури се състоят от четири стандартни слоя: презентация, бизнес, постоянство и база данни.

Всеки слой от модела на многослойната архитектура има специфична роля и отговорност в рамките на приложението. Например, презентационният слой би бил отговорен за обработката на целия потребителски интерфейс и комуникационна логика на браузъра, докато бизнес слоя би бил отговорен за изпълнението на конкретни бизнес правила, свързани със заявката. Всеки слой в архитектурата формира абстракция около работата, която трябва да се извърши, за да се удовлетвори конкретна бизнес заявка. Например, презентационният слой не трябва да знае или да се притеснява как да получи клиентски данни; трябва само да покаже тази информация на екран в определен формат. По същия начин, бизнес слоят не трябва да се притеснява как да форматира клиентските данни за показване на екран или дори откъде идват клиентските данни; трябва само да получи данните от слоя за устойчивост, да изпълни бизнес логика спрямо данните (напр. да изчисли стойности или обобщени данни) и да предаде тази информация до презентационния слой.

Ключови концепции

Във Фигура 8. всеки от слоевете в архитектурата е маркиран като "затворен". Затворен слой означава, че когато заявката преминава от един слой към друг, тя трябва да премине през слоя точно под него, за да достигне следващия слой под този.

Концепцията за слоеве на изолация означава, че промените, направени в един слой на архитектурата, обикновено не засягат компонентите в други слоеве: промяната е изолирана в компонентите в този слой и, вероятно, в друг свързан слой (като например слой за постоянство съдържащ SQL). Ако се позволи на презентационния слой директен достъп до този за постоянство, промените в SQL биха засегнали както бизнес слоя, така и презентационния, като следствие се създава много тясно свързано приложение с много зависимости между компонентите. Този тип архитектура става много труден и скъп за промяна. Въпреки че затворените слоеве улесняват изолацията и по този начин помагат да се изолират промените в архитектурата. Има моменти, когато има смисъл за определени слоеве да бъдат отворени. Например, ако трябва да се добави слой за споделени услуги към архитектурата, съдържащ общи компоненти за услуги, достъпени от компонентите в бизнес слоя (например, класове за обработка на данни и низове или класове за одит и регистрация). Създаването на слой за услуги обикновено е добра идея в този случай, защото архитектурно ограничава достъпа

до споделените услуги до бизнес слоя (и не до презентационния слой). Без отделен слой, няма нищо архитектурно, което ограничава презентационния да достъпва тези общи услуги, което прави трудно управлението на това ограничение на достъпа.

Пример на шаблон

За да се разбере как работи слоестата архитектура, трябваа да се разгледа заявката за извличане на информация за конкретно лице, както е показано на Фигура 10. Черните стрелки показват заявката, която се пренасочва към базата данни, за да се извлече информацията за клиента, а червените стрелки показват отговора, който се връща към екрана, за да се покаже информацията. В този пример информацията за клиента се състои както от данни за клиента, така и от данни за поръчките (поръчките, направени от клиента).

Фиг. 10 Заявка за извличане на информация

Екранът за клиента е отговорен за приемането на заявката и показването на информацията. Той не знае къде се намират данните, как се извличат или колко много таблиците в базата данни трябва да бъдат заявени, за да се получат данните. След като екранът за клиента получи заявка за получаване на информация, тя препраща тази заявка към модула за делегиране на клиенти. Този модул е отговорен за знанието за това, кои модули в бизнес слоя могат да обработват тази заявка, както и за начина на достъп до този модул и какви данни са му необходими (договорът). Обектът на клиента в бизнес слоя е отговорен за агрегирането на цялата информация, необходима за бизнес заявката (в този случай за получаване на информация за клиента). Този модул се обръща към модула за достъп до данни на клиенти (DAO - data access object) в слоя за постоянство, за да получи данни за клиента, както и към модула за достъп до поръчки, за да получи информация за поръчките. Тези модули, от своя страна, изпълняват SQL заявки, за да извлекат съответните данни и ги предават нагоре на обекта на клиента в бизнес слоя. След като обектът на клиента получи данните, той ги агрегира и предава тази информация нагоре на делегата на клиента, който после предава тези данни на екрана за клиентите, за да бъдат показани на потребителя.

Модели на компоненти на уеб приложения

1. **Един Уеб Сървър, Една База Данни** Това е най-простият, но и най-малко надежден модел на компонентите на уеб приложението. Този модел използва един сървър и една база данни. Уеб приложение, изградено върху такъв модел, ще се срине веднага след като сървърът се срине. Следователно, той не е много надежден. Моделът за уеб приложение с един уеб сървър и една база данни обикновено не се използва за реални уеб приложения. Той се използва главно за

изпълнение на тестови проекти, както и с цел научаване и разбиране на основите на уеб приложението.

2. **Няколко Уеб Сървъра, Една База Данни** Идеята с този тип модел на компонентите на уеб приложението е, че уеб сървърът не съхранява данни. Когато уеб сървърът получи информация от клиент, той я обработва и я записва в базата данни, която се управлява извън сървъра. Това понякога се нарича и архитектура без състояние. За този модел на компонентите на уеб приложението са необходими поне 2 сървъра. Това е за избягване на отказ. Дори когато един от сървърите се срине, другият ще вземе контрол. Всички направени заявки автоматично ще бъдат пренасочени към новия сървър и уеб приложението ще продължи изпълнението. Следователно надеждността е по-добра в сравнение с единния сървър с вградена база данни. Все пак, ако базата данни се срине, уеб приложението ще последва същия път.

3. **Няколко Уеб Сървъра, Няколко Бази Данни** Това е най-ефективният модел на компонентите на уеб приложението, защото нито уеб сървърите, нито базите данни имат единна точка на отказ. Има две опции за този тип модел. Или да се съхраняват идентични данни във всички използвани бази данни, или да се разпределят равномерно между тях. Обикновено за първия случай са необходими не повече от 2 бази данни, докато за втория случай някои данни може да станат недостъпни в случай на срив на базата данни. Нормализацията на DBMS се използва, въпреки че в двете сценария. Когато мащабът е голям, т.е. повече от 5 уеб сървъра или бази данни или и двете, се препоръчва инсталиране на балансьори на товара.