****

**Изпитна тема № 1:**

Програмиране План-тезис:

Основни понятия: програмиране, език за програмиране, алгоритъм, среда за разработка (IDE), компилация и интерпретация. Пресмятания, оператори, изрази. Условни конструкции. Логически изрази и оператори за сравнение. Вложени условни оператори. Цикли. Вложени цикли. Подпрограми (функции/методи), параметри, връщана стойност.

Програмиране

Програмирането е наука, която се занимава с методите и средствата за създаване на компютърни програми. Компютърната програма представлява алгоритъм, който може да бъде „разбран“ и изпълнен от компютъра. Процесът по създаването на програма включва писането, тестването и поддържането на нейния сорс код (изходен код). Кодът, който се изписва чрез език за програмиране, може да бъде модификация на съществуващ вече код или да бъде нов, с цел решаването на зададен проблем.

Език за програмиране

Език за програмиране е изкуствен език, предназначен за изразяване на изчисления, които могат да се извършат от машина, по-специално от компютър. Езиците за програмиране могат да се използват за създаване на програми, които контролират поведението на машина, да реализират алгоритми точно или във вид на човешка комуникация.

Алгоритъм

Алгоритъм (от името на учения ал–Хорезми) е термин от математиката, информатиката, лингвистиката и други области, с който се описва сложно действие чрез редица от елементарни (достатъчно прости) действия, които изпълняващият може да извърши в последователни стъпки без допълнителни обяснения. Обикновено изпълнението на алгорътъма включва изчисление или обработка на данни.

Интегрираната среда за разработка

Интегрираната среда за разработка (на английски: integrated development environment, IDE) e софтуерно приложение, което предоставя цялостна среда за разработване на софтуер на програмистите. IDE обикновено се състои от редактор на код, инструменти за автоматизиране построяването на изходното приложение и дебъгер. Повечето съвременни интегрирани среди предлагат автоматично предсказване и дописване на кода.

Компилатор

Компилаторът (англ. compiler, от compile – съчетавам, съставям) е компютърна програма, която превежда (компилира) даден компютърен изходен код в семантично отговарящ код на език от (обикновено) по-ниско ниво. Целевият език може да бъде машинен език или асемблерен език за конкретен процесор или процесорна фамилия, както и междинен език за конкретна виртуална машина (например байткод за виртуална машина на Java) или друг език от високо ниво. Когато се компилира до машинен език, крайният продукт е изпълнима програма или обектен код.

Интерпретатор

Интерпретаторът представлява програма, която изпълнява друга програма. Интерпретаторъ извършва последователен анализ на командите от изходния код(написата вече програма, която искаме компютърът да изпълни), непосредствено ги превежда на машинен език и изпълнява т.е. не превежда целият изходен код само веднъж както прави компилатора, а го прави всеки път когато се извиква програмата. Основният недостатък на интерпретаторите е, че когато дадена програма се интерпретира обикновено работи по-бавно от колкото би работила ако е била компилирана.

Пресмятания

Пресмятания в програмирането

За компютрите знаем, че са машини, които обработват данни. Всички данни се записват в компютърната памет (RAM памет) в променливи. Променливите са именувани области от паметта, които пазят данни от определен тип, например число или текст. Всяка една променлива в C# има име, тип и стойност. Ето как бихме дефинирали една променлива, като едновременно с декларацията ѝ, ѝ присвояваме и стойност:

Оператори

Чрез операторите в програмните езици се осъществява обработката на данни. Всеки оператор има строго определена функционалност, в зависимост от конкретния програмен език, към който принадлежи. Операторите се обозначават със символи (напр. !, %, +, || и др.) и се характеризират с: позиция спрямо аргументите си (инфиксен или постфиксен), приоритет и асоциативност.

Операторите позволят обработка на прими­тивни типове данни и обекти. Те приемат като вход един или няколко операнда и връщат като резултат някаква стойност. Операторите в C# представляват специални символи (като например "+", ".", "^" и други) и извършат специфични преобра­зувания над един, два или три операнда. Пример за оператори в C# са знаците за събиране, изваждане, умножение и делене в математиката (+, - , \*, /) и операциите, които те извършват върху целите и реалните числа.

Изрази

Голяма част от работата на една програма е пресмятането на изрази. Изразите представляват поредици от оператори, литерали и променливи, които се изчисляват до определена стойност от някакъв тип (число, сим­волен низ, обект или друг тип).

Условни конструкции

В компютърните науки, условните конструкции са функции на езика за програмиране, чрез които можем да изпълняваме различни действия в зависимост от някакво условие.

В императивните езици за програмиране, обикновено терминът „условни конструкции“ се използва, докато във функционалното програмиране, термините „условен израз“ или „условна конструкция“ са предпочитани, защото тези термини имат различни значения.

Логически изрази

Логическите изрази са еквивалент на вече разгледаните съждителни формули. Елементарните логически изрази /съждения/ в езика за програмиране се образуват с помощта на операциите за сравняване: по-голямо (>), по-малко (<), равно на (=), по-голямо от или равно на (> = ), по-малко от или равно на (< = ), и не равно на /различно от/ - (<>). Сложни логически изрази се съставят по правилата за съждителни формули, като за операциите КОНЮНКЦИЯ, ДИЗЮНКЦИЯ и ОТРИЦАНИЕ се използват служебните думи AND, OR и NOT.

Оператори за сравнение

Операторите за сравнение в Java се използват за сравняване на две или повече операнди. Java поддържа шест оператора за сравнение: -по-голямо (>) -по-малко (<) -различие (!=)

Всички оператори за сравнение са двуаргументни (приемат два операнда), а върнатият от тях резултат е булев (true или false). Операторите за сравнение имат по-малък приоритет от аритметичните, но са с по-голям от операторите за присвояване на стойност.

Вложени условни оператори

Условният оператор реализира конструкция "ако - то" и разклонените алгоритми. Това означава, че решава поставени задачи чрез вземане на решение. Съществуват две форми на условния оператор - кратка и пълна

1. Кратка форма

а) синтаксис

if(<условие>) <оператор>;

където:

<условие> е логически израз (израз за сравнение, чийто резултат е от логически тип – true или false). Задължително се загражда в скоби.

<оператор> е допустим за езика C++ оператор. Ако операторите са повече от един, се използва конструкцията блок

б) как се изпълнява

Изчислява се стойността на логическия израз (условието).

Aко условието има стойност true (истина), се изпълнява оператора, след което изпълнението на програмата продължава със следващия по ред oператор.

Ако условието има стойност false (Неистина, лъжа), програмата продължава с оператора, зададен след if.

Цикли

Цикличният изчислителен процес е процес, за по-накратко наричан цикъл, който представлява многократното изпълнение на дадена последователност от операции с различни данни. Най-често се променя само една величина, която се нарича параметър на цикъла. При различните видове цикли, програмният код се повтаря докато е в сила определено предварително зададено условие или фиксиран брой пъти, които са упоменати в началото. Всеки цикличен процес се характеризира със следните елементи: *Инициализация, Тяло на цикъла, Актуализация и Прекъсващо условие.*

Вложени цикли

Вложените цикли представляват конструкция, при която в тялото на един цикъл (външен) се изпълнява друг цикъл (вътрешен). За всяко завъртане на външния цикъл, вътрешни

ят се извърта целият. Това се случва по следния начин: При стартиране на изпълнение на вложени цикли първо стартира външният цикъл: извършва се инициализация на неговата управляваща променлива и след проверка за край на цикъла, се изпълнява кодът в тялото му. След това се изпълнява вътрешният цикъл. Извършва се инициализация на началната стойност на управляващата му променлива, прави се проверка за край на цикъла и се изпълнява кодът в тялото му. При достигане на зададената стойност за край на вътрешния цикъл, програмата се връща една стъпка нагоре и се продължава започналото изпълнение предходния (външния) цикъл. Променя се с една стъпка управляващата променлива за външния цикъл, проверява се дали условието за край е удовлетворено и започва ново изпълнение на вложения (вътрешния) цикъл. Това се повтаря докато променливата на външния цикъл достигне условието за край на цикъла.

Подпрограма

Подпрограма е явление от програмирането, чрез което се описват поредица от програмни инструкции, които решават даден проблем – може да приема параметри и да връща стойност. Подпрограмата може да се приеме като строителен блок на програмата, т.к. в нея се извършва цялостната обработка на данни, която трябва да се извърши, за да се реши поставената задача. Подпрограмите могат да се дефинират в самата програма или в библиотеки, които могат да бъдат използвани от множество програми.

Методи/Функции

Метод (на английски: method) в програмирането е съставна част от една програма, която решава конкретен проблем. Може както да приема параметри така и да връща стойност. Целта на писането на дадена програма е решаването на дадена задача. За да бъде ефективно решена дадена задача в програмирането, тя се разделя на подзадачи, разработват се алгоритми за решение на тези подзадачи и накрая тези подзадачи се сглобяват в цялостна програма. Обособените парчета код, решаващи дадената подзадача се наричат подпрограми (subroutines). В различните езици подпрограмите се срещат под други имена като функции (functions) или процедури (procedures). В C#, те се наричат методи (methods).

Параметри и Връщана стойност

Параметрите, аргументите и връщаната стойност са компоненти на функциите/методите в компютърното програмиране.

**Изпитна тема № 2:**

Програмиране План-тезис:

Команди за работа със сорс-контрол системи. Видове типове данни, бройни системи и понятие за обект. Работа с масиви и списъци. Дебъгване и работа с дебъгер. Символни низове и работа с текст. Многомерни масиви. Речници и хеш-таблици.

Команди за работа със сорс-контрол системи

Сорс контрол системите са създадени за контрол над изходния код на програмите. С течение на времето еволюират и сега се наричат с по-общото наименование системи за контрол чрез версии (version control systems).

1. От клиент (програмист) към сървър

* добвяне на нов файл / директория
* изтриване на съществуващ файл директория
* преименуване или преместване на файл директория
* добавяне на нова версия на съществуващ файл - може да се наложи синхронизиране на съдържанието на файла ако някой друг е добавил нова версия на съшия файл.

2. От сървър към клиент (програмист)

* сваляне на последна версия - може да се наложи синхронизиране на съдържанието на файла ако е бил променян от клиента.
* синхронизиране на директория - изтриват се файлове, които са били изтрити на сървъра, преименуват се и се преместват файлове директории ако така е станало в сървъра

3. Промяна на състояние на сървъра

* маркиране на файл за редактиране
* заключване / отключване на файл - само този който е заключил даден файл може да добавя нови версии
* слагане на етикет - маркиране на дадено състояние в историята на файлове / директории. Например, когато се счете, че дадена компилация на програмата е стабилна може да се слоят етикет "Stable build (1209)". Ползите от етикетите съм обяснил по-долу.
* създаване на разклонение в главното развитие на историята на файловете.
* възстановяване на стара версия на файл да е последната, т.е. отхвърляне на историята след дадена версия.

Команди на Git Bash

Git Bash е снабден с допълнителни команди, които могат да бъдат намерени в директорията / usr / bin на емулацията на Git Bash. Git Bash всъщност може да осигури доста стабилно изживяване на черупката на Windows. Git Bash се предлага в комплект със следните команди на черупки, които са извън обхвата на този документ: Ssh, scp, cat, find.

В допълнение към обсъждания по-рано набор от команди на Bash, Git Bash включва пълния набор от основни команди на Git, обсъдени в този сайт. Научете повече на съответните страници с документация за git clone, git commit, git checkout, git push и др.

Видове типове данни

Типът на данните (на английски: data type) е концепция в езиците за програмиране, с която се описва видът на променливата, какви стойности може да приема, както и множеството от операции, които могат да се прилагат върху нея. Типът е вид класификация на данните, която указва на компилатора или интерпретатора как програмистът има намерение да използва данните. Например процесът на събиране на две променливи, както и резултатът от събирането варират в зависимост от типа данни: цели числа, числа с плаваща запетая, низове.

Типовете данни представляват множества (диапазони) от стойности, които имат еднакви характеристики:

име на типа (byte, int, string, int[], List.

размер (колко памет заемат) – например 32 байта.

стойност по подразбиране (default value).

Бройни системи

Бройните системи (numeral systems) са начин за представяне (запис­ване) на числата, чрез краен набор от графични знаци наречени цифри. Към тях трябва да се добавят и правила за представяне на числата. Символите, които се използват при представянето на числата в дадена бройна система, могат да се възприемат като нейна азбука.

По време на различните етапи от развитието на човечеството, различни бройни системи са придобивали известност. Трябва да се отбележи, че днес най-широко разпространение е получила арабската бройна система. Тя използва цифрите 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9, като своя азбука. (Интересен е фактът, че изписването на арабските цифри в днешно време се различава от представените по-горе десет цифри, но въпреки това, те пак се отнасят за същата бройна система, т.е. десетичната).

Понятие за обект

В контекста на обектно-ориентираното програмиране, обектът се разбира като съвкупност от елементи и тяхното поведение. Софтуерните обекти моделират обекти от реалния свят или абстрактни концепции. Примери за реални обекти са животни, растения, коли, сгради, езера и т.н. Абстрактните обекти са понятия в някоя предметна област, които се налага да моделираме и използваме в компютърна програма. Примери за абстрактни обекти са структурите от данни стек, опашка, списък и дърво.

Масиви

Масивът в компютърните науки е наредена последователност от елементи от един и същ базов тип. Отделен елемент на масива се указва посредством името на целия масив, последвано от поредния номер (индекс) на елемента. В различните езици за програмиране масивите са изградени по различни начини.

Списъци

Списъкът (List<T>) е линейна структура от данни, която съдържаща в себе си поредица от елементи. Различава се от масива по това, че може да се оразмерява динамично. Списъците имат свойството дължина (брой елементи) и елементите му са наредени последователно. С помощта на имплементираните към него методи, е възможно добавянето на нови елементи на която и да е позиция в списъка, махането на такива, обхождането или обръщането на тези елементи и т.н.

Работа с дебъгер/Дебъгване

Дебъгване (от английското debugging) е процесът на проследяване на изпълнението на дадена компютърна програма с цел намиране и отстраняване на грешки („бъгове“) в нея. Извършва се с помощта на специализирани програмни инструменти наречени дебъгери.

Многобройни книги са написани за дебъгването, тъй като то включва много аспекти, включително интерактивни отстраняване на грешки, контрол на потока, интеграционно тестване, лог файлове, наблюдение (приложения, система), профилиране, статистически контрол на процеса, и специални дизайн тактики за подобряване на откриването на бъговете.

Символни низове

Символният низ е последователност от символи, записана на даден адрес в паметта. Помните ли типа char? В променливите от тип char можем да запишем само един символ. Когато е необходимо да обработваме повече от един символ на помощ идват низовете.

Работа с текст

Затова повечето програмисти ползват IDE в ежедневната си работа. Програмата се пише в текстов формат, а самият текст на програмата се нарича сорс код (source code). Той се компилира до изпълним файл.

Многомерни масиви

Всеки допустим в C# тип може да бъде използван за тип на елементите на масив. Масивите също може да се разглеждат като допустим тип. Така можем да имаме масив от масиви, който ще разгледаме по-нататък.

Едномерен масив от цели числа декларираме с int[], а двумерен масив с int[,]. Следния пример показва това:

|  |
| --- |
| int[,] twoDimentionalArray; |

Такива масиви ще наричаме двумерни, защото имат две измерения или още матрици (терминът идва от математиката). Масиви с повече от едно измерение ще наричаме многомерни.

Аналогично можем да декларираме и тримерни масиви като добавим още едно измерение:

|  |
| --- |
| int[,,] threeDimentionalArray; |

На теория няма ограничения за броя на размерностите на тип на масив, но в практиката масиви с повече от две размерности са рядко използвани и затова ще се спрем по-подробно на двумерните масиви.

Речници

В предните няколко теми се запознахме с някои класически и много важни структури от данни – масиви, списъци, дървета и графи. В тази - ще се запознаем с така наречените "речници" (dictionaries), които са изключително полезни и широко използвани в програмирането.

Речниците са известни още като асоциативни масиви (associative arrays) или карти (maps). Тук ще използваме терминът "речник". Всяко едно от различните имена подчертава една и съща характеристика на тази структура от данни, а именно, че в тях всеки елемент представлява съответствие между ключ и стойност – наредена двойка. Аналогията идва от факта, че в един речник, например тълковния речник, за всяка дума (ключ) имаме обяснение (стойност). Подобни са тълкованията и на другите имена.

Хеш-таблица

Структурата от данни хеш-таблица обикновено се реализира с масив. Тя съдържа наредени двойки (ключ, стойност), които са разположени в масива на пръв поглед случайно и непоследователно. В позициите, в които нямаме наредена двойка, имаме празен елемент (null):

[](https://introprogramming.info/wp-content/uploads/2011/07/clip_image004.jpg)

Размерът на таблицата (масива), наричаме капацитет (capacity) на хеш-таблицата. Степен на запълненост (load factor), наричаме реално число между 0 и 1, което съответства на отношението между броя на запълнените елементи и текущия капацитет. На фигурата имаме хеш-таблица с 3 елемента и капацитет m. Следователно степента на запълване на тази хеш-таблица е 3/m.

Изпитна тема № 3

Обектно-ориентирано програмиране План-тезис: Дефиниране на класове: клас, конструктор, полета, свойства, създаване на обекти от клас. Дефиниране на функции/методи в класовете, ключова дума this. Енкапсулация на данни в класовете, методи за достъп и промяна на полета (getters/setters). Статични полета и методи в класовете.

Дефиниране на класове

Декларирането на клас следва строго определени правила (синтаксис):

|  |
| --- |
| **[<access\_modifier>] class <class\_name>** |

Когато декларираме клас, задължително трябва да използваме ключовата дума **class**. След нея трябва да стои името на класа **<class\_name>**.

Освен ключовата дума **class** и името на класа, в декларацията на класа могат да бъдат използвани някои модификатори, например разгледаните вече модификатори за достъп.

Клас

Клас – категория обекти с общи свойства и операции, които могат да се извършват върху тях (например всички сметки в дадена банка)

Класът е една от характеристиките на обектно-ориентираното програмиране. Това е съвкупност от променливи и функции, които са обвързани в логическа структура и работят заедно. Класът служи като модел за представяне на реални обекти и софтуерни обекти, описвайки атрибути (свойства) и методи (поведение) на обектите.

Конструктор

В обектно-ориентираното програмиране конструкторът е блок от инструкции, който се изпълнява за инициализация на един обект при неговото създаване.

Конструкторът е специален метод на класа, който се извиква автоматично при създаването на обект от този клас и извършва инициализация на данните му (това е неговото основно предназначение). Конструкторът няма тип на връщана стойност и неговото име не е произволно, а задължително съвпада с името на класа. Конструкторът може да бъде със или без параметри. Конструктор без параметри наричаме още конструктор по подразбиране (default constructor).

Конструкторите може да имат параметри, както всеки друг метод. Всеки клас може да има произволен брой конструктори с единственото ограни­чение, че броят и типът на параметрите им трябва да бъде различен. При създаването на обект от този клас се извиква точно един от дефинираните конструктори.

При наличието на няколко конструктора в един клас естествено възниква въпросът кой от тях се извиква при създаването на обект. Този проблем се решава по много интуитивен начин. Подходящият конструктор се избира автоматично в зависимост от подадените параметри при създава­нето на обекта. Използва се принципът на най-добро съвпадение.

Полета

(полетата) се дефинират в тялото на класа. Те са места, които са хранилища за данни. Там се пазят данните (стойностите), които отразяват състоянието на всяка конкретна инстанция на класа (обект) и са нужни за работата на други видове членове на класа – методи, конструктори, индексатори, свойства – моделиращи поведението на коя да е инстанция (т.е. на всички инстанции) на този клас. Съществуват и такива полета, наречени статични, които са общи за всички инстанции (обекти).

Свойства

**Свойства (Properties)**

В света на обектно-ориентираното програмиране съществува елемент на класовете, наречен **свойство (property)**, който е нещо средно между поле и метод и служи за по-добра защита на състоянието в класа. В някои езици за обектно-ориентирано програмиране, като С#, Delphi / Free Pascal, Visual Basic, JavaScript, D, Python и др., свойствата са част от езика, т.е. за тях съществува специален механизъм, чрез който се декларират и използват. Други езици, като например Java, не подържат концепцията за свойства и за целта програмистите, трябва да декларират двойка методи (за четене и модификация на свойството), за да се предостави тази функционалност.

**Свойствата в С# – представяне чрез пример**

Използването на свойства е доказано добра практика и важна част от концепциите на обектно-ориентираното програмиране. Създаването на свойство в програмирането става чрез деклариране на два метода – един за достъп (четене) и един за модификация (записване) на стойността на съответното свойство.

Нека разгледаме един пример. Да си представим, че имаме отново клас **Dog**, който описва куче. Характерно свойство за едно куче е, например, цвета му (color). Достъпът до свойството "цвят" на едно куче и съответната му модифика­ция може да осъществим по следния начин:

|  |
| --- |
| // Getting (reading) a property  string colorName = dogInstance.Color;    // Setting (modifying) a property  dogInstance.Color = "black"; |

В обектите могат да се отделят следните три групи характеристики:

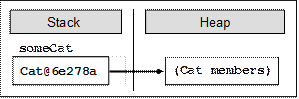
* **Състояния** (states) – това са характеристики на обекта, които го определят и описват по принцип или в конкретен момент;
* **Поведения** (behaviors) – това са специфични характерни действия, които обектът може да извършва;
* **Идентичност** (Identity) – уникалност на един обект, която го отличава от други обекти.

Създаване на обекти от клас

Създаването на обекти от предварително дефинирани класове по време на изпълнението на програмата става чрез оператора **new**. Новосъз­даденият обект обикновено се присвоява на променлива от тип, съвпадащ с класа на обекта. Ще отбележим, че при това присвояване същинският обект не се копира. В променливата се записва само **референция** към новосъздадения обект (неговият адрес в паметта). Следва прост пример как става това:

|  |
| --- |
| Cat someCat = **new** Cat(); |

На променливата **someCat**от тип **Cat** присвояваме новосъздадена инстан­ция на класа **Cat**. Променливата someCat стои в стека, а нейната стойност (инстанцията на класа **Cat**) стои в динамичната памет:

[](https://introprogramming.info/wp-content/uploads/2011/07/clip_image00218.gif)

**Създаване на обекти със задаване на параметри**

Сега ще разгледаме леко променен вариант на горния пример, при който задаваме параметри при създаването на обекта:

|  |
| --- |
| Cat myBrownCat = **new** Cat("Johnny", "brown"); |

В този случай искаме обектът **myBrownCat** да представлява котка, която се казва Johnny и има кафяв цвят. Указваме това чрез думите **"Johnny"** и **"brown"**, написани в скоби след името на класа.

При създаването на обект с оператора **new**се случват две неща**:**заделя се памет за този обект и се извършва начална инициализация на член-данните му. Инициализацията се осъществява от специален метод на класа, наречен **конструктор**. В горния пример инициализиращите пара­метри са всъщност параметри на конструктора на класа. Ще се спрем по-подробно на конструкторите след малко. Понеже член-променливите **name** и **color** на класа Cat са от референтен тип (от класа **String**), те се записват също в динамичната памет (heap) и в самия обект стоят техните референции (адреси).

Дефиниране на функции/методи в класовете

За разлика от декларацията на клас, при декларирането на полета и методи на класа, могат да бъдат използвани и четирите нива на достъп – public, protected, internal и private. Нивото на видимост protected ня­ма да бъде разглеждано в тази глава, тъй като е обвързано с наследяването на класове и е обяснено подробно в главата "Принципи на обектно-ориентираното програмира­не".

Преди да продължим, нека припомним, че ако един клас A, не е видим (ня­ма достъп) от друг клас B, тогава нито един елемент (поле или метод) на класа A, не може да бъде достъ­пен от класа B.

Ключова дума this

Ключовата дума **this** в C# дава достъп до референцията към текущия обект, когато се използва от метод в даден клас. Това е обектът, чийто метод или конструктор бива извикван. Можем да я разглеждаме като ука­зател (референция), дадена ни априори от създателите на езика, с която да достъпваме елементите (полета, методи, конструктори) на собствения ни клас:

|  |
| --- |
| this.myField; // access a field in the class  this.DoMyMethod(); // access a method in the class  this(3, 4); // access a constructor with two int parameters |

За момента няма да обясняваме изложения код. Разяснения ще дадем по-късно, в местата от секциите на тази глава, посветени на елементите на класа (полета, методи, конструктори) и засягащи ключовата дума **this**.

Капсулация

**Свойства – капсулация на достъпа до полетата**

Основната цел на свойствата е да осигуряват капсулация на състоянието на класа, в който са декларирани, т.е. да го защитят от попадане в невалидни състояния.

**Капсулация (encapsulation)** наричаме скриването на физическото представяне на данните в един клас, така че, ако в последствие променим това представяне, това да не рефлектира върху останалите класове, които използват този клас.

Чрез синтаксиса на C#, това се реализира като декларираме полета (физи­чес­кото представяне на данните) с възможно най-ограничено ниво на видимост (най-често с модификатор **private)** и декларираме достъпът до тези полета (четене и модифициране) да може да се осъществява единствено чрез специални методи за достъп (accessor methods).

Деклариране на свойства

За да декларираме едно свойство в C#, трябва да декларираме методи за достъп (за четене и промяна) на съответното свойство и да решим по какъв начин ще съхраняваме ин­фор­ма­цията за това свойство в класа.

Преди да декларираме методите обаче, трябва да декларираме самото свойството в класа. Формално декларацията на свойствата изглежда по следния начин:

|  |
| --- |
| **[<modifiers>] <property\_type> <property\_name>** |

С **<modifiers>** сме означили, както модифика­то­ри­те за достъп, така и други модификатори (например**static**, който ще разгледаме в [следва­щата секция на главата](https://introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D0%B8_%D0%BA%D0%BB%D0%B0%D1%81%D0%BE%D0%B2%D0%B5_(static)). Те не са задължи­тел­­на част от декларацията на едно поле.

Типа на свойството **<property\_type>**задава типа на стойностите на свойството. Може да бъде както примитивен тип (например **int**), така и референтен (например масив).

Съответно, **<property\_name>**е името на свойството. То трябва да започва с главна буква и да удовлетворява правилото **PascalCase**, т.е. всяка нова дума, която се долепя в задната част на името на свойството, започва с главна буква. Ето няколко примера за правилно именувани свойства:

|  |
| --- |
| // MyValue property  public int MyValue { get; set; }    // Color property  public string Color { get; set; }    // X-coordinate property  public double X { get; set; } |

**Тяло на свойство**

Подобно на класа и методите, свойствата в С# имат тяло, където се декларират методите за достъп до свойството (accessors).

|  |
| --- |
| **[<modifiers>] <property\_type> <property\_name>**  **{**        // ... Property's accessors methods go here  **}** |

Тялото на свойството започва с отваряща фигурна скоба "**{**" и завършва със затваряща – "**}**". Свойствата винаги трябва да имат тяло.

Методи за достъп

(getters/setters)

**Метод за четене на стойността на свойство (getter)**

Както обяснихме, декларацията на **метод за четене на стойността на едно свойство** (в литературата наричан още **getter**) се прави в тялото на свойството, като за целта трябва да се спазва следния синтаксис:

|  |
| --- |
| **get { <accessor\_body> }** |

Съдържанието на блока ограден от фигурните скоби (**<accessor\_body>**) е подобно на съдържанието на произволен метод. В него се декларират действията, които трябва да се извършат за връщане на резултата от метода.

Методът за четене на стойността на едно свойство трябва да завършва с **return** или **throw** операция. Типът на стойността, която се връща като резултат от този метод, трябва да е същият както типa **<property\_type>** описан в декларацията на свойството.

Въпреки, че по-рано в тази секция срещнахме доста примери на декла­рирани свойства с метод за четене на стойността им, нека разгледаме още един пример за свойството "възраст" (**Age**), което е от тип **int** и е декларирано чрез поле в същия клас:

|  |
| --- |
| private int age;                          // Field declaration    public string Age                         // Property declaration  {        get { return this.age; }      // Getter declaration  } |

**Извикване на метод за четене на стойността на свойство**

Ако допуснем, че свойството **Age** от последния пример е декларирано в клас от тип **Dog**, извикването на метода за четене на стойността на свойството, става чрез точкова нотация, приложена към променлива от типа, в чийто клас е декларирано свойството:

|  |
| --- |
| Dog dogInstance = new Dog();  // ...  int dogAge = dogInstance.Age;                         // Getter invocation  Console.WriteLine(dogInstance.Age);       // Getter invocation |

Последните два реда от примера показват, че достъпвайки чрез точкова нотация името на свойството, автоматично се извиква неговият getter метод (методът за четене на стойността му).

Промяна на полета

(getters/setters)

**Метод за промяна на стойността на свойство (setter)**

По подобие на метода за четене на стойността на едно свойство, може да се декларира и **метод за промяна (модификация) на стойността на едно свойство** (в литературата наричан още **setter**). Той се декларира в тялото на свойството с тип на връщана стойност **void** и в него подадената при присвояването стойност е достъпна през неявен параметър **value**.

Декларацията се прави в тялото на свойството, като за целта трябва да се спазва следнияt синтаксис:

|  |
| --- |
| **set { <accessor\_body> }** |

Съдържанието на блока ограден от фигурните скоби (**<accessor\_body>**) е подобно на съдържанието, на произволен метод. В него се декларират действията, които трябва да се извършат за промяна на стойността на свойството. Този метод използва неявен параметър, наречен **value**, който е предоставен от С# по подразбиране и който съдържа новата стойност на свойството. Той е от същия тип, от който е свойството.

Нека допълним примера за свойството "възраст" (**Age**) в класа **Dog**, за да онагледим казаното дотук:

|  |
| --- |
| private int age;                          // Field declaration    public string Age                         // Property declaration  {        get{ return this.age; }        set{ this.age = value; }      // Setter declaration  } |

Статични Полета в класовете

**Статични полета**

Когато създаваме обекти от даден клас, всеки един от тях има различни стойности в полетата си. Например, нека разгледаме отново класа **Dog**:

|  |
| --- |
| public class Dog  {        // Instance variables        private string name;        private int age;  } |

Той има две полета съответно за име – **name** и възраст – **age**. Във всеки обект, всяко едно от тези полета има собствена стойност, която се съхранява на различно място в паметта за всеки обект.

Понякога обаче, искаме да имаме полета, които са общи за всички обекти от даден клас. За да постигнем това, трябва в декларацията на тези полета да използваме модификатора **static**. Както вече обяснихме, такива полета се наричат **статични полета (static fields)**. В литерату­рата се срещат, също и като **променливи на класа**.

Казваме, че статичните полета са **асоциирани с класа**, вместо с който и да е обект от този клас. Това означава, че всички обекти, съз­дадени по описанието на един клас **споделят** статичните полета на класа.

Статични методи в класовете

**Статични методи**

По подобие на статичните полета, когато искаме един метод да е асоцииран само с класа, но не и с конкретен обект от класа, тогава го декларираме като статичен.

**Декларация на статични методи**

Синтактично да декларираме **статичен метод** означава, в деклара­цията на метода, да добавим ключовата дума **static**:

|  |
| --- |
| **[<access\_modifier>] static <return\_type> <method\_name>()** |

Нека например декларираме метода за събиране на две числа, за който говорихме в началото на настоящата секция:

|  |
| --- |
| public static int Add(int number1, int number2)  {        return (number1 + number2);  } |

**Достъп до статични методи**

Както и при статичните полета, статичните методи могат да бъдат достъп­вани чрез точкова нотация (операторът точка) приложена към името на класа, като името на класа може да се пропусне ако извикването се извършва от същия клас, в който е деклариран статичният метод. Ето един пример за извикване на статичния метод **Add(…)**:

**Изпитна тема № 4**

Обектно-ориентирано програмиране План-тезис:

Компонентно тестване. Шаблонни класове и методи. Наследяване, абстракция и интерфейси. Полиморфизъм. Итератори. Компаратори. Отражение на типовете. Ламбда изрази и функции. Библиотека за обработка на колекции. Делегати. Комуникация между обекти. Изключения. Работа с потоци и файлове. Базови шаблони за дизайн.

Компонентно тестване

(Unit testing)

Чрез този вид тестване се проверяват отделни единици (компоненти) от код – най-малките градивни частици, които могат да бъдат тествани самостоятелно. Например, ако желаете да тествате компонент, който визуализира потребителско име и аватар, ще искате да проверите дали елементите се показват правилно, без значение в каква среда е поставен компонента. Важното е да работи правилно в изолирана среда. Ако снимката излиза и името се изписва – всичко е наред.

Тестване на интеграционно ниво (integration)

Този тип проверка е подобно на компонентното, но не се проверява само един компонент, а цяла група. Взимат се модулите, които са преминали през компонентното тестване, групират се и им се дават задачите, които са предвидени да изпълняват. Целта е да се разбере дали всяка единица взаимодейства с останалите както трябва.

Системно тестване

(end-to-end testing)

Системното тестване, както можете да предположите, прави доста по-обстойна проверка на написания код. Този вид тестване цели да премине през цялостната апликация и да провери функционалността ѝ. Той пресъздава взаимодействието между потребителя и апликацията. Пример за системно тестване би било влизане на заглавната страница, кликване върху бутона Login, въвеждане на име и парола и влизане в личните данни на профила. Така се проверя последователността на действия и плавността между преминаването от едно към друго.

Шаблонни типове параметри

Шаблонни типове са концепция за типизиране на параметри в програмирането. Те позволяват да се изработват класове и методи, които да отложат спецификацията на един или повече типове данни, докато един клас или метод не се декларира и не получи своята инстанция. Например, чрез използване на шаблонен тип параметър T може да се напише един клас, който друг код също може да използва, без да понесе забавяне на времето на работа.

Шаблонни класове

**Какво представляват шаблонните класове?**

Както знаем, когато за работата на един метод е нужна допълнителна информация, тази информация се подава на метода чрез параметри. По време на изпълнение на програмата, при извикване на метода, подаваме аргументи на метода, те се присвояват на параметри­те му и след това се използват в тялото на метода.

По подобие на методите, когато знаем, че функционалността (действията) капсулирана в един клас, може да бъде приложена не само към обекти от един, а от много (разнородни) типове, и тези типове не са известни по време на деклариране на класа, можем да използваме една функционал­ност на езика С# наречена **шаблонни типове (generics)**. Тя ни позволява да декларираме параметри на самия клас, чрез които обознача­ваме неиз­вестния тип, с който класът ще работи в последствие. След това, когато инстанцираме нашия типизиран клас, ние заместваме неизвестния тип с конкретен. Съответно новосъздаденият обект ще работи само с обекти от конкретния тип, който сме задали при инициализацията му. Конкретният тип може да бъде всеки един клас, който компилаторът разпознава, включително структура, изброен тип или друг шаблонен клас.

За да добием по-ясна представа за същността на шаблонните типове, нека се върнем към нашата задача от предходната секция. Както се досещаме, кла­сът, който описва приют на животни (**AnimalShelter**), може да опе­рира с различни типове животни. Следователно, ако искаме да създа­дем генерално решение на задачата, по време на декларация на класа **AnimalShelter**, ние не можем да знаем какъв тип животни ще бъдат приютявани в приюта. Това е достатъчна индикация, че можем да типизираме нашия клас, добавяйки към декларацията на класа, като параметър, неизвестния ни тип на животни.

В последствие, когато искаме да създадем приют за кучета например, на този параметър на класа ще подадем името на  нашия тип – класа **Dog**. Съответно, ако създаваме приют за котки, ще подадем типа **Cat** и т.н.

|  |  |
| --- | --- |
| [clip_image007[26]](https://introprogramming.info/wp-content/uploads/2011/07/clip_image00726.gif) | **Типизирането на клас (създаването на шаблонен клас) представлява добавяне, към декларацията на един клас, на параме­тър (за­ме­стител) на неизвестен тип, с който класът ще работи по време на изпълне­ние на програмата. В послед­ствие, когато класът бива инстанциран, този параметър се замества с името на някой кон­кретен тип.** |

В следващите секции ще се запознаем със синтаксиса на типизирането на класове и ще представим нашия пример преработен, така че да използва типизиране.

Шаблонни методи

Подобно на класовете, когато при декларацията на един метод, не можем да кажем от какъв тип ще са параметрите му, можем да типизираме метода. Съответно, указването на конкретния тип ще стане по време на извикване на метода, заменяйки непознатият тип с конкретен, както направихме при класовете.

Типизирането на метод се прави, като веднага след името и преди отва­рящата кръгла скоба на метода, се добави **<K>**, където **K** е заместителят  на типа, който ще се използва в последствие:

|  |
| --- |
| **<return\_type> <methods\_name><K>(<params>)** |

Съответно, можем да използваме неизвестния тип **K** за параметрите в списъка с параметри на метода **<params>**, чийто тип не ни е известен, а също и като връщана стойност или за деклариране на променливи от типа заместител **K** в тялото на метода.

(ДОПЪЛНИТЕЛНА ИНФОРМАЦИЯ)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Капсулация на данни

Капсулация на данните Означава скриване на ненужните детайли за обектите и откриване към външния свят само на важните техни характеристики и свойства.

Наследяване

Избягва повторното дефиниране на атрибути и операции на обекти и техни наследници и позволява създаването на йерархии от класове, моделиращи по естествен начин зависимости и съподчинености от реалността.

Полиморфизъм

Буквално означава приемането на различни форми от един обект. Полиморфизмът позволява извършване на едно действие по различен начин от различни обекти.

Нива на Видимост

Видимостта на членовете е ключов момент в разработването на йерархии от класове, тъй като основно правило в ООП е клиентът на класа да вижда само което му е необходимо и нищо повече.

**public** - глобална видимост, членове с такова ниво на достъп могат да се достъпват от всеки тип.

**protected internal** – такива членове са видими от всички типове, дефинирани в асемблито, в което е дефиниран дадения тип данни, a също и от наследниците му.

**internal** – членове с такова ниво на достъп се достъпват от всички типове, дефинирани в асемблито, в което е дефиниран дадения.

**protected** – такива членове са видими само от наследниците на дадения тип

**private** - капсулирани членове, видими единствено в рамките на типа

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Наследяване (Inheritance)

Наследяването е основен принцип от обектно-ориентираното програми­ране. То позволява на един клас да "наследява" (поведение и характе­ристики) от друг, по-общ клас. Например лъвът е от семейство котки. Всички котки имат четири лапи, хищници са, преследват жертвите си. Тази функцио­налност може да се напише веднъж в клас Котка и всички хищници да я преизползват – тигър, пума, рис и т.н.

Абстракция (Abstraction)

Следващият основен принцип от обектно-ориентираното програмиране, който ще разгледаме, е "абстракция". Абстракцията означава да работим с нещо, което знаем как да използваме, но не знаем как работи вътрешно. Например имаме телевизор. Не е нужно да знаем как работи телевизорът отвътре, за да го ползваме. Нужно ни е само дистанционното и с малък брой бутони (интерфейс на дистанционното) можем да гледаме телевизия.

Същото се получава и с обектите в ООП. Ако имаме обект Лаптоп и той се нуждае от процесор, просто използваме обекта Процесор. Не знаем (или по-точно не се интересуваме) как той смята вътрешно. За да го използваме, е достатъчно да извикваме метода сметни() с подходящи параметри.

Абстракцията е нещо, което правим всеки ден. Това е действие, при което игнорираме всички детайли, които не ни интересуват от даден обект, и разглеждаме само детайлите, които имат значение за проблема, който решаваме. Например в хардуера съществува абстракция "устройство за съхранение на данни", което може да бъде твърд диск, USB memory stick, флопи диск или CD-ROM устройство. Всяко от тях работи вътрешно по различен начин, но от гледна точка на операционната система и на програмите в нея те се използват по еднакъв начин – на тях се записват файлове и директории. В Windows имаме Windows Explorer и той умее да работи по еднакъв начин с всички устройства, независимо дали са твърд диск или USB stick. Той работи с абстракцията "устройство за съхранение на данни" (storage device) и не се интересува как точно данните се четат и пишат. За това се грижат драйверите за съответните устройства. Те се явяват конкретни имплементации на интерфейса "устройство за съхране­ние на данни".

Абстракцията е една от най-важните концепции в програмирането и в ООП. Тя ни позволява да пишем код, който работи с абстрактни струк­тури от данни (например списък, речник, множество и други). Имайки абстрактния тип данни, ние можем да работим с него през неговия интер­фейс, без да се интересуваме от имплементацията му. Например можем да запазим във файл всички елементи на списък, без да се интересуваме дали той е реализиран с масив, чрез свързан списък или по друг начин. Този код остава непроменен, когато работим с различни конкретни типове данни. Дори можем да пишем нови типове данни (които се появяват на по-късен етап) и те да работят с нашата програма, без да я променяме.

Абстракцията ни позволява и нещо много важно – да дефинираме интерфейс на нашите програми, т.е. да дефинираме всички задачи, които тази програма може да извърши, както и съответните входни и изходни данни. Така можем да направим няколко по-малки програми, всяка от които да извършва някаква по-малка задача. Като прибавим това към факта, че можем да работим с абстрактни данни, ни дава голяма гъвка­вост при свързването на тези по-малки програми в една по-голяма и ни дава повече възможности за преизползване на код. Тези малки подпро­грами се наричат компоненти. Този начин на писане на програми намира широко приложение в практиката, защото ни позволява не само да преизползваме обекти, а дори цели подпрограми.

Интерфейси

В езика C# интерфейсът е дефиниция на роля (на група абстрактни действия). Той дефинира какво поведение трябва да има един обект, без да указва как точно се реализира това поведение.

Един обект може да има много роли (да имплементира много интерфейси) и ползвателите му могат да го използват от различни гледни точки.

Например един обект Човек може да има ролите Военен (с поведение "стреляй по противника"), Съпруг (с поведение "обичай жена си"), Данъкоплатец (с поведение "плати си данъка"). Всеки човек обаче импле­ментира това поведение по различен начин: Иван си плаща данъците навреме, Георги – не навреме, Петър – въобще не ги плаща.

Някой може да попита защо най-базовият за всички обекти клас Object не е всъщност интерфейс. Причината е, че тогава всеки клас щеше да трябва да имплементира една малка, но много важна, група методи, а това би отнемало излишно време. Оказва се, че и не всеки клас има нужда от специфична реализация на Object.GetHashCode(), Object.Equals(…), Object.ToString(), тоест имплемен­тацията по подразбиране върши работа в повечето случаи. От класа Object не е нужно да се пренапише (повторно имплементира) никой метод, но ако се наложи, това може да се направи. Пренаписването на методи е обяснено в детайли в секцията за виртуални методи.

Интерфейси – ключови понятия

В интерфейса може да има само декларации на методи и константи.

Полиморфизъм (Polymorphism)

Следващият основен принцип от обектно-ориентираното програмиране е "полиморфизъм". Полиморфизмът позволява третирането на обекти от наследен клас като обекти от негов базов клас. Например големите котки (базов клас) хващат жертвите си (метод) по различен начин. Лъвът (клас наследник) ги дебне, докато Гепардът (друг клас-наследник) просто ги надбягва.

Полиморфизмът дава възможността да третираме произволна голяма котка просто като голяма котка и да кажем "хвани жертвата си", без значение каква точно е голямата котка.

Полиморфизмът може много да напомня на абстракцията, но в програмирането се свързва най-вече с пренаписването (override) на методи в наследените класове с цел промяна на оригиналното им поведение, насле­дено от базовия клас. Абстракцията се свързва със създаването на интерфейс на компонент или функционалност (дефиниране на роля). Пренаписва­нето на методи ще разгледаме в детайли след малко.

Итератори

Итератор (на английски: Iterator) е поведенчески шаблон за дизайн, който се използва в обектно-ориентираното програмиране. Итераторът предоставя начин за последователен достъп до елементите на обект, без да е нужна вътрешна информация за обекта. В компютърно програмиране, итератор е обект, който дава възможност на програмиста да преминава през структура от данни. Различни видове итератори често са предоставяни чрез интерфейс контейнер. Въпреки че интерфейсът и семантиката на даден итератор са фиксирани, итераторите често се прилагат по отношение на свързаните с тях структури за изпълнение и често са плътно свързани към контейнера, за да отговарят на семантиката на Итератора. Трябва да се има предвид, че итераторът чете и също така дава достъп до елементите в една структура от данни, но не изпълнява итерация (при тривиално използване на терминологията). Итераторът поведенчески е подобен на курсора на база данни.

Компаратор

За сравнение на обектите на дефинирани от потребителя класове се използва интерфейс за сравнение. Обектът за сравнение може да сравнява два обекта от два различни класа. Следващата функция сравнява obj1 с obj2. Синтаксис: public int compare (Object obj1, Object obj2)

Отражение на типовете

Отражението на типовете (reflection) е един механизъм, предоставен от платформата .NET Framework, даващ възможност за получаване на информация за типовете по време на изпълнение на програмата. Това позволява проектирането на динамично разширяващи се приложения, към които лесно могат да се добавят типове на други програмисти и компании.

В някаква форма отражение (reflection) има във всички управлявани платформи като Java, Perl и др.

Ламбда изрази

(lambda expressions)

Ламбда израз е кратък блок код, който приема параметри и връща стойност. Ламбда изразите са подобни на методите, но не се нуждаят от име и могат да бъдат внедрени точно в тялото на метод.

Ламбда изразите представляват анонимни функции, които съдържат изрази или последователност от оператори. Всички ламбда изрази използват ламбда оператора =>, който може да се чете като "отива в". Идеята за ламбда изразите в C# е взаимствана от функционалните езици (например Haskell, Lisp, Scheme, F# и др.). Лявата страна на ламбда оператора определя входните параметри на анонимната функция, а дясната страна представлява израз или последователност от оператори, която работи с входните параметри и евентуално връща някакъв резултат.

Обикновено ламбда изразите се използват като предикати или вместо делегати (променливи от тип функция), които се прилагат върху колекции, обработвайки елементите от колекцията по някакъв начин и/или връщайки определен резултат.

**Ламбда изрази – примери**

Например нека да разгледаме разширяващия метод **FindAll(…)**, който може да се използва за отсяване на необходимите елементи. Той работи върху определена колекция, прилагайки ѝ даден предикат, който проверява всеки от елементите на колекцията дали отговаря на определено условие. За да го използваме, обаче, трябва да включим референция към библиотеката **System.Core.dll** и namespace-а **System.Linq**, тъй като разширяващите методи върху колекциите се намират в този namespace.

Ако искаме например да вземем само четните числа от колекция с цели числа, можем да използваме метода **FindAll(…)** върху колекцията, като му подадем ламбда метод, който да провери дали дадено число е четно:

Видове ламбда изрази

(lambda expressions)

**Използване на ламбда изрази с анонимни типове**

С ламбда изрази можем да създаваме и колекции с анонимни типове, от колекция с някакви елементи. Например нека от колекцията **dogs**, съдържаща елементи от тип **Dog** да създадем нова колекция с елементи от анонимен тип, с 2 свойства – възраст и първата буква от името на кучето:

**Сортиране чрез ламбда изрази**

Ако искаме да сортираме елементите в дадена колекция, можем да изпол­зваме разширяващите методи **OrderBy** и **OrderByDescending** като им пода­дем чрез ламбда функция начина, по който да сортират елементите. Пример отново върху колекцията **dogs**:

**Оператори в ламбда изразите**

Ламбда функциите могат да имат и тяло. До сега използвахме ламбда функциите само с един оператор. Сега ще разгледаме ламбда функции, които имат тяло. Да се върнем на примера с четните числа. За всяко число, към което се прилага нашата ламбда функция, искаме да отпечатаме на конзолата стойността му и да върнем като резултат дали е четно или не:

**Ламбда изразите като делегати**

Ламбда функциите могат да бъдат записани в променливи от тип делегат. Делегатите представляват специален тип променливи, които съдържат функции. Стандартните типове делегати в .NET са **Action**, **Action<in T>**, **Action<in T1, in T2>,**и т.н. и **Func<out TResult>, Func<in T, out TResult>, Func<in T1, in T2, in TResult>**и т.н. Типовете **Func** и **Action** са generic и съдържат типовете на връщаната стойност и типовете на параметрите на функциите. Променливите от тези типове са референции към функции. Ето пример за използването и присвояването на стойности на тези типове.

Библиотека за обработка на колекции

В областта на програмирането, библиотеката е колекция от помощни ресурси под формата на код и данни, съхранявани в енергонезависимата памет, предназначени за използване от независими програми, например от операционната система или от програми за разработка на софтуер. Това позволява споделяне и промяна на данни и код на модулен принцип.

Повечето компилиращи програмни езици имат стандартна библиотека, макар че софтуерните разработчици могат да създават собствени.

Делегат

Делегат: човек, изпратен или оторизиран да представлява други хора. Делегатът в езика C# представя метод, който може да бъде извикан отдалечено или да бъде предаден като параметър. ‒ референтен тип, абстракция на метод; ‒ разглежда се като елегантен метод – представя метод с дадена сигнатура сигнатура; ‒ еквивалент на обект на функция; ‒ производен клас на типа System.MulticastDelegate, наследник на System.Delegate; ‒ осигурява асинхронна обработка на събития; ‒ използва се за т.н. callback функционалност – методи с параметър указател към функция, която се извиква чрез този указател:

Комуникация между обекти

В ООП обектите не комуникират непременно помежду си, като предават съобщения. Те комуникират помежду си по някакъв начин, който им позволява да определят какво искат да се направи, но оставя изпълнението на това поведение на приемащия обект. ... Друг начин е да извикате (виртуален) метод в приемащия обект

Изключения

Определение: Изключение е събитие, което се случва по време на изпълнението на програма, която нарушава нормалния поток от инструкции. ... Манипулаторът на изключения се счита за подходящ, ако видът на хвърленото изключение е същият като типа на изключението, обработвано от манипулатора.

При програмирането текат данни. Така че, просто казано, поток в програмирането означава поток от данни. Потокът е основно поредица от данни. Каквито и данни да използваме в нашето програмиране, текат през поток.

Работа с потоци

В компютърните науки потокът е поредица от елементи от данни, предоставени с течение на времето. Потокът може да се разглежда като елементи на конвейер, които се обработват един по един, а не като големи партиди.

Потоците се обработват по различен начин от пакетните данни - нормалните функции не могат да работят с потоци като цяло, тъй като имат потенциално неограничени данни и формално потоците са кодати (потенциално неограничени), а не данни (което е ограничено). Функциите, които работят върху поток, произвеждайки друг поток, са известни като филтри и могат да бъдат свързани в тръбопроводи, аналогично на функционалния състав. Филтрите могат да работят по един елемент от поток наведнъж или да основават изходен елемент на множество входящи елементи, като плъзгаща се средна стойност.

Работа с файлове

Компютърни файлове. Компютърен файл се използва за съхраняване на данни в цифров формат като обикновен текст, данни за изображения или друго съдържание. Въвеждане / извеждане на файл. Режими на работа с файлове. Отваряне на файлове. Затваряне на файл. Писане на файл. Четене на файл. Файл I / O в Java.

Шаблони за дизайн

Шаблоните за дизайн (англ: Software design pattern) представляват концепция предназначена за разрешаване на често срещани проблеми в обектно-ориентираното програмиране. Тази концепция предлага стандартни решения за архитектурни и концептуални проблеми в компютърното програмиране.

Тук не става въпрос за конкренти алгоритми или част от програмен код. Шаблоните за дизайн са независими от програмния език. Те представляват архитектурни решения на вече познати и много често срещани проблеми в програмирането. Може да се каже, че шаблоните за дизайн представляват средство за прилагане на световния опит на програмисти и аналитици.

**Изпитна тема № 5:**

Алгоритми и структури от данни План-тезис:

Въведение в алгоритмите. Линейни структури от данни. Списък, стекове, опашки и имплементации. Алгоритми върху линейни структури: подредици, нарастващи редици, площадка от еднакви елементи. Примерна приложна задача: По време на теоретичния изпит се предоставя непълен/неработещ/некоректен програмен фрагмент на приложната задача. Предоставеният фрагмент да се приведе в работещ вид.

Въведение в алгоритмите

По-строго дефинирано, алгоритъмът е ефективен метод за изчисляване на функция, който може да бъде изразен в рамките на крайно време и пространство и чрез добре дефиниран формален език. Започвайки от начално състояние и входни данни (понякога празни), инструкциите описват пресмятания, чието изпълнение преминава през краен брой добре дефинирани последователни състояния и завършва с крайно състояние, като в процеса се получават крайни резултати. Не е задължително преходът между състоянията да е детерминиран (еднозначно определен): някои алгоритми, известни като вероятностни алгоритми, съдържат елемент на случайност.

Структури от данни

Структурите от данни са множество от данни, които са организирани на основата на логически и математически закони. Често изборът на правилната структура прави програмата по-ефективна, тъй като се спестява памет и време за изпълнение. Данните биват групирани по определен начин, за да се улесни достъпът до тях и управлението им. За различни задачи са подходящи различни структури. За най-общо групиране може да се използва специфичен идентификатор, както например при речника думите са групирани по начална буква. За по-сложни случаи могат да се създадат много по-сложни структури. При избор на подходяща структура е възможно по-бързо и икономично (с ползване на минимум ресурси) обработване на информацията.

Дефинирането на структури от данни става посредством задаване на общи правила за връзките между данните и възможните операции с тях.

От основните видове са изведени специфични структури за определени задачи (например двоични дървета за реализиране на бази данни).

Линейни

Линейните структури от данни са списъци, стекове и опашки.

Линейният списък е редица от елементи от един и същи тип. Основни операции, които могат да бъдат извършвани с елементите, са добавяне и премахване.

Видове линейни списъци

линеен едносвързан списък – всеки елемент, с изключение на последния, е свързан със следващия с една връзка. Списъкът се обхожда от началото към края.

линеен двусвързан списък – всеки елемент, с изключение на последния, е свързан със следващия посредством две връзки. Това улеснява операциите. Например до елемент на списък лесно се стига в зависимост от това, дали е по-близо до началото или до края на списъка.

Цикличен списък

– двусвързан или едносвързан списък, при който последният елемент е и предшественик на първия. Тази реализация премахва условната поредност на елементи в списък и улеснява операциите с тях.

Паралелен списък

– Съвкупност от няколко списъка. Възможен е паралелен достъп до елементи от тях.

Стек

– в един стек елементи се добавят и премахват само от единия край, като се спазва правилото LIFO (last in first out – от англ. – „последният влязъл пръв излиза“), т.е. елементът, добавен най-скоро, е пръв при достъп до стека. Така операциите върху елементите биват ограничени.

Опашка

– достъпът до елементи на опашка е също ограничен като при стек. Тук действа обаче правилото FIFO (first in, fisrt out – от англ. – „първият влязъл пръв излиза“), според което елементът, който най-дълго време е в опашката (е най-рано добавен), се обработва пръв. Добавянето на елементи става само от края на опашката, а премахването – от началото.

Подредици

Последователностите са основната логическа структура на алгоритми или програми. Когато създавате алгоритми или програми, инструкциите се представят в точно правилен ред. Последователността може да съдържа произволен брой инструкции, но всяка инструкция трябва да се изпълнява в реда, в който са представени. Никоя инструкция не може да бъде пропусната.

Нарастващи подредици

Нарастващи подредици Иванчо има пред себе си пермутация P на числата от 1 до N. Числата са подредени от ляво надясно. Дълго време той се чудеше какво да прави с тази пермутация и накрая реши да взема различни нарастващи подредици от нея. Всяка подредица се определя от последователност от индекси i1 < i2 < … < ik, за които трябва да е изпълнено p[i1] < p[i2] < … < p[ik]. Иванчо иска да намери дължината на най-дългата възможна нарастваща подредица, която може да вземе от пермутацията, както и дължината на най-дългата нарастваща подредица, за която е изпълнено, че последното число е точно фиксирано А. Вход: На първия ред от входния файл maximal.in се въвежда естественото число N - дължината на пермутацията P, както и числото А, на което искаме да завършва втората търсена от нас най-дълга нарастваща подредица. На втория ред от входния файл са въведени N числа от 1 до N, всяко се среща по веднъж - началната пермутация P. Изход: На единствения ред от изходния файл maximal.out изведете две числа. Първото е дължината на най-дългата нарастваща подредица в пермутацията. Второто - дължината на най-дългата нарастваща подредица, която завършва с числото А.

Алгоритми върху линейни структури

В компютърните науки линейното търсене или последователното търсене е метод за намиране на елемент в списък. Той проверява последователно всеки елемент от списъка, докато се намери съвпадение или не се търси целият списък.

Линейното търсене се изпълнява в най-лошото линейно време и прави най-много n сравнения, където n е дължината на списъка. Ако всеки елемент е еднакво вероятно да бъде търсен, тогава линейното търсене има среден случай от n + 1/2

сравнения, но средният случай може да бъде засегнат, ако вероятностите за търсене за всеки елемент варират. Линейното търсене рядко е практично, тъй като други алгоритми и схеми за търсене, като двоичен алгоритъм за търсене и хеш таблици, позволяват значително по-бързо търсене на всички, освен на кратки списъци.

**Изпитна тема № 6:**

Алгоритми и структури от данни План-тезис:

Сортиране и търсене. Сортиране, устойчивост, бързи и бавни алгоритми. Метод на пряката селекция, метод на мехурчето, сортиране чрез вмъкване, сортиране чрез броене, бързо сортиране, сортиране чрез сливане и имплементации. Линейно търсене, двоично търсене, интерполационно търсене.

Сортиране и търсене

Алгоритъм за сортиране е алгоритъм, който подрежда списък от елементи в определена последователност. Най-използваните подредби са числовите и лексикографските подредби. Ефективните алгоритми за сортиране са важни за оптимизацията на други алгоритми (например алгоритми за търсене, алгоритми за сливане и др.), който изискват входните данни да са сортирани в определена последователност. Често също така е полезно за конкатенизиране (сливане) на данни и за генериране на разбираеми за човек крайни резултати. Формално казано, изходният резултат от алгоритъма за сортиране трябва да задоволява две условия:

Изходният резултат е в ненамаляваща последователност (всеки елемент не трябва да е по-малък от предходните на базата на очакваната обща подредба);

Изходният резултат е пермутация (пренаредба) на входните елементи.

От раждането на компютърните науки, алгоритмите за сортиране са били в процес на разработка и развитие. Ефективното решаване на проблеми, които са на пръв поглед прости и познати, се оказва доста по-сложна и трудоемка задача. Например методът на мехурчето за пръв път е бил анализиран през 1956 година. Въпреки че мнозина са смятали проблема за сортиране за решен, нови по-ефективни алгоритми са продължавали да се откриват (например методът на библиотечното сортиране е бил публикуван за първи път през 2006 година). Алгоритмите за сортиране често се представят в началните класове по компютърни науки, където изобилието от алгоритми за решаването на един проблем елегантно показва разнообразието от ключови алгоритмични концепции, като например голямото О, разделяй и владей, структурни данни, случайни алгоритми, най-добър, най-лош и среден случай, компромис време-памет, горна и долна граница.

Устойчивост

Модел за стабилност (SM) е метод за проектиране и моделиране на софтуер. Това е разширение на методологията за обектно-ориентиран софтуерен дизайн (OOSD), подобно на UML, но добавя собствен набор от правила, насоки, процедури и евристика, за да постигне по-усъвършенстван обектно-ориентиран софтуер.

Мотивацията е да се постигне по-високо ниво на функции на OO като -

Стабилност - означава, че голяма част от обектите ще бъдат стабилни с течение на времето и няма да се нуждаят от промени.

Многократна употреба - това означава, че повечето обекти могат да бъдат използвани повторно за различни видове приложения.

Поддържаемост - това е функция на ОО, която се отнася повече за SM, тъй като обектите ще се нуждаят от най-малко поддръжка.

Сложност на алгоритъм

Не може да се говори за ефективност на алгоритми и структури от данни, без да се използва понятието "сложност на алгоритъм", с което вече се сблъскахме няколко пъти под една или друга форма. Няма да даваме математическа дефиниция, за да не натоварваме читателите, а ще дадем неформално обяснение.

**Сложност на алгоритъм** е мярка, която отразява порядъка на броя операции, необходими за изпълнение на дадена операция или алгоритъм като функция на обема на входните данни. Формулирано още по-просто, сложност е груба, приблизителна оценка на броя стъпки за изпълнение на даден алгоритъм. При оценяването на сложност говорим за порядъка на броя операции, а не за техния точен брой. Например ако имаме от порядъка на N2 операции за обработката на N елемента, то N2/2 и 3\* N2 са брой операции от един и същ квадратичен порядък. Сложността на алго­ритмите се означава най-често с нотацията О(f), където f е функция на размера (обема) на входните данни.

Сложността може да бъде константна, логаритмична, линейна, n\*log(n), квадратична, кубична, експоненциална и друга. Това означава, че се изпълняват съответно от порядъка на константен, логаритмичен, линеен и т.н. брой стъпки за решаването на даден проблем.

|  |  |
| --- | --- |
| [clip_image001[31]](https://introprogramming.info/wp-content/uploads/2011/07/clip_image001311.gif) | **Сложност на алгоритъм е груба оценка на броя стъпки, които алгоритъмът ще направи в зависимост от размера на входните данни. Това е груба оценка, която се интересува от порядъка на броя стъпки, а не от точния им брой.** |

**Типични сложности на алгоритмите**

Ще обясним какво означават видовете сложност чрез следната таблица:

|  |  |  |
| --- | --- | --- |
| **Сложност** | **Означение** | **Описание** |
| константна | O(1) | За извършване на дадена операция са необходими константен брой стъпки (например 1, 5, 10 или друго число) и този брой не зависи от обема на входните данни. |
| логаритмична | O(log(N)) | За извършване на дадена операция върху N елемента са необходими брой стъпки от порядъка на log(N), където основата на логаритъма е най-често 2. Например алгоритъм със сложност O(log(N)) за N = 1 000 000 ще направи около 20 стъпки (с точност до константа). Тъй като основата на логаритъма няма съществено значение за порядъка на броя операции, тя обикновено се изпуска. |
| линейна | O(N) | За извършване на дадена операция върху N елемента са необходими приблизително толкова стъпки, колкото са елементите. Например за 1 000 елемента са нужни около 1 000 стъпки. Линейната сложност означава, че броят елементи и броят операции са линейно зависими, например броят стъпки за N елемента е около N/2 или 3\*N. |
|  | O(n\*log(n)) | За извършване на дадена операция върху N елемента са необходими приблизително N\*log(N) стъпки. Например при 1 000 елемента са нужни около 10 000 стъпки. |
| квадратична | O(n2) | За извършване на дадена операция са необходими от порядъка на N2 на брой стъпки, където N характеризира обема на входните данни. Например за дадена операция върху 100 елемента са необходими 10 000 стъпки. Реално квадратична сложност имаме, когато броят стъпки е в квадратна зависимост спрямо обема на входните данни, например за N елемента стъпките могат да са от порядъка на 3\*N2/2. |
| кубична | O(n3) | За извършване на дадена операция са необходими от порядъка на N3 стъпки, където N характеризира обема на входните данни. Например при 100 елемента се изпълняват около 1 000 000 стъпки. |
| експоненциална | O(2n), O(N!), O(nk), … | За извършване на дадена операция или изчисление са необходими брой стъпки, който е в експоненциална зависимост спрямо размера на входните данни. Например при N=10 експоненциалната функция 2N има стойност 1024, при N=20 има стойност 1 048 576, а при N=100 функцията има стойност, която е число с около 30 цифри. Експоненциалната функция N! расте още по-бързо: за N=5 има стойност 120, за N=10 има стойност 3 628 800, а за N=20 – 2 432 902 008 176 640 000. |

Бързи и бавни алгоритми

Сложност и време за изпълнение

Скоростта на изпълнение на програмата е в пряка зависимост от слож­ността на алгоритъма, който се изпълнява. Ако тази сложност е малка, програмата ще работи бързо, дори за голям брой елементи. Ако слож­ността е голяма, програмата ще работи бавно или въобще няма да работи (т.е. ще заспи) при голям брой елементи.

Ако вземем един средностатистически компютър от 2008 година, можем да приемем, че той изпълнява около 50 000 000 елементарни операции в секунда. Разбира се, това число трябва да ви служи единствено за груб ориентир. Различните процесори работят с различна скорост и различните елементарни операции се изпълняват с различна скорост, а и компютър­ната техника постоянно напредва. Все пак, ако приемем, че използваме средностатистически домашен компютър от 2008 г., можем да направим следните изводи за скоростта на изпълнение на дадена програма в зави­симост от сложността на алгоритъма и обема на входните данни

Метод на пряката селекция

В компютърните науки методът на пряката селекция (на английски: Selection sort) е алгоритъм за сортиране. Той е един от фундаменталните методи за сортиране и е прост и лесен на имплементиране.

Алгоритъмът има сложност от Θ(n2), т.е. времето за изпълнението му е пропорционално на квадрата на броя на елементите в масива. Това го прави неефикасен при големи списъци и като цяло работи по-зле от подобния му алгоритъм за сортиране чрез вмъкване (insertion sort). Сортирането чрез пряка селекция впечатлява с простотата си, а също така в дадени ситуации има предимства пред някои сложни алгоритми.

Алгоритъмът работи по следния начин:

1. Намира най-малкия елемент в списъка като сравнява първият елемент с всички останали
2. Разменя го с елемента на първа позиция
3. Повтаря горните две стъпки за всеки следващ елемент

Алгоритъмът за сортиране на сортиране сортира масив, като многократно намира минималния елемент (като се разглежда възходящ ред) от несортираната част и го поставя в началото. Алгоритъмът поддържа два подмасива в даден масив.

1) Подмасивът, който вече е сортиран.

2) Оставащият подсорт, който е несортиран.

Във всяка итерация на сортиране на селекцията, минималният елемент (като се разглежда възходящ ред) от несортирания подмасив се избира и премества в сортирания подмасив.

Mетод на мехурчето

Алгоритъмът работи по следния начин: взимаме първият елемент на масива и го сравняваме със следващия (втория в нашия случай) и разменяме стойностите им, ако първият е по – голям от втория. След това сравняваме вторият елемент с третия и пак разменяме, ако има нужда. Ако нашият масив е от 10 елемента, след 9 такива сравнения най – отгоре ще изплува най – голямата стойност. След това започваме отново да сравняваме като пак взимаме първият елемент и сравняваме с втория и така нататък.

В най-добрия случай методът на мехурчето има сложност Ω(n). Ако масивът е вече сортиран, методът на мехурчето ще премине през масива веднъж и ще установи, че не трябва да разменя никакви елементи.

Сортиране чрез вмъкване

Сортиране чрез вмъкване (на английски: Insertion sort) е прост сортиращ алгоритъм. Чрез сравняващо сортиране сортираният списък се допълва с по един елемент всеки път. Алгоритъмът е доста неефикасен в сравнение с quicksort, heapsort или mergesort, ако се прилага върху големи списъци, но от друга страна има и доста предимства.

В най-лошия случай алгоритъмът има сложност O(n2)

## **Принцип на действие**

1. Списъкът с елементи, които ще бъдат сортирани се разделя на две части: частта със сортираните елементи и частта с несортираните
2. При всяка стъпка се взема първият елемент от несортирания списък и се вмъква на правилната позиция в сортираната част от списъка
3. Сортирането продължава докато елементите от несортираната част на списъка се изчерпят

Сортиране чрез броене

В информатиката сортирането чрез броене е алгоритъм за сортиране на съвкупност от обекти спрямо техните ключове, които са малки цели числа. Алгоритъмът брои обектите с различни стойности на ключа, след което разполага в изходния масив съответен брой обекти от всяка стойност на ключа последователно – от най-малкия до най-големия ключ. Времето за изпълнение е линейно спрямо броя на елементите и спрямо разликата между максималната и минималната ключова стойност, затова е подходящ само в случаите, когато разликата в ключовете не е много по-голяма от броя на елементите. Въпреки това често се използва като подпрограма в други алгоритми за сортиране, например в метода на бройните системи, който по-добре се справя с големи ключове.

Тъй като сортирането чрез броене използва ключови стойности като индекси в масив, то не сортира чрез сравняване и долната граница Ω(n log n) не се отнася за него. Bucket сортирането може да бъде използвано за много подобни случаи, в които се използва и сортирането чрез броене. Въпреки това, сравнено със сортирането чрез броене, bucket сортирането изисква свързани списъци, динамични масиви или голямо количество заделена памет, където да съхранява групите от елементи, разпределени в bucket-и, докато сортирането чрез броене пази едно-единствено число (броя елементи) за всеки bucket.

Бързо сортиране

Бързо сортиране (на английски: quick sort) е добре известен сортиращ алгоритъм, разработен от Ч. А. Р. Хор през 1960 г. и публикуван през 1961 г.,. Основната част на алгоритъма се състои в сравняващо сортиране.

В най-лошия случай алгоритъмът има сложност O(n2). Средната времева сложност е O(n log n), а амортизираната сложност е 1,386 n log n сравнения. В практиката бързото сортиране е с по-добро време от други сортиращи алгоритми с време O(n log n). Освен това, последователността на бързото сортиране и локализираните препратки към паметта работят добре с кеша. Бързото сортиране се основава на сравнения. То не е устойчиво, тоест може да размества елементи с еднакви ключове. Класическият алгоритъм използва допълнителен масив, но съществува вариант, който сортира данните на място – без заделяне на втори масив, така че се използва само O(log n) допълнителна памет.

## **Принцип на действие**

1. Избира се „главен“ елемент от списъка с елементи, които ще бъдат сортирани.
2. Списъкът се пренарежда така, че всички елементи, които са по-малки от „главния“ се поставят вляво от него, а всички, които са по-големи – вдясно от него.
3. Рекурсивно се повтарят горните стъпки върху списъка с по-малките и списъка с по-големите елементи.
4. Получените списъци се сливат (конкатенация) и се получава сортираният списък.

Сортиране чрез сливане

В информатиката сортирането чрез сливане е алгоритъм за сортиране, базиран на сравняване, който винаги има сложност {\displaystyle \Theta (nlog(n))}{\displaystyle \Theta (nlog(n))}. Алгоритъмът се гради на принципа „разделяй и владей“. Създаден е от Джон фон Нойман през 1945. Стабилен е и паметта, която му трябва, в най-лошия случай е n. По-подробно описание и анализ на сортирането чрез сливане, се е появило в доклад на Goldstine и Neumann още през 1948 година.

## **Принцип на действие**

1. Несортираният списък по произволен начин се разделя на два подсписъка с приблизително еднаква дължина (за линейно време)
2. Рекурсивно се разделят подсписъците, докато не се достигне до списъци с единична дължина
3. Сливат се два подсписъка в нов сортиран списък (за линейно време)

**Имплементации**

Метод на пряката селекция

Пример на C# за сортиране на числа по метода на пряката селекция

using System;

class SelectionSort

{

static void Main()

{

int[] array = new int[] { 64, 25, 12, 22, 11 };

for (int i = 0; i < array.Length – 1; i++)

{

for (int j = i + 1; j < array.Length; j++)

{

if (array[i] > array[j]) // swap items

{

int tmp = array[i];

array[i] = array[j];

array[j] = tmp;

}

}

}

for (int i = 0; i < array.Length; i++) // print sorted array

{

Console.Write(array[i] + " ");

}

}

}

Метод на мехурчето

## **Пример на C# за сортиране на числата, чрез алгоритъма на мехурчето**

using System;

class BubbleSort

{

static void Main()

{

int[] array = new int[] { 6, 9, 4, 3, 5, 1, 42, -2 };

for (int i = 0; i < array.Length - 1; i++)

{

for (int j = 0; j < array.Length - 1; j++)

{

if (array[j] > array[j + 1]) // swap the elements

{

int tmp = array[j];

array[j] = array[j + 1];

array[j + 1] = tmp;

}

}

}

for (int i = 0; i < array.Length; i++) // print the elements

{

Console.Write(array[i] + " ");

}

}

}

Сортиране чрез вмъкване

## **Примерен Код на C#**

static int[] InsSort(int[] arr)

{

for (int i = 0; i < arr.Length; i++)

{

int value = arr[i];

int index = i;

while (index > 0 && arr[index - 1] >= value)

{

arr[index] = arr[index - 1];

index--;

}

arr[index] = value;

}

return arr;

}

Сортиране чрез броене

*# allocate an array Count[0..k]; THEN*

*# initialize each array cell to zero; THEN*

**for** each input item x:

increment Count[key(x)]

*''' изчислява се началния индекс за всеки ключ: '''*

total = 0

**for** i = 0, 1, ... k:

oldCount = Count[i]

Count[i] = total

total = total + oldCount

*''' входните данни се копират подредени в изходния масив: '''*

*# allocate an output array Output[0..n-1]; THEN*

**for** each input item x:

Output[Count[key(x)]] = x

increment Count[key(x)]

**return** Output

След първия цикъл Count[i] съхранява броя елементи с ключ, равен на i. След второто обхождане той вече съхранява броя елементи с ключ, по-малък от i, който е същият като първия индекс, на който елемент с ключ i трябва да бъде запазен в изходния масив. По време на третия цикъл Count[i] винаги запазва следващата позиция в изходния масив, където трябва да бъде съхранен елемент с ключ i, така че всеки елемент е преместен на вярната позиция в изходния масив. Относителният ред на елементи с еднакви ключове се съхранява тук, т.е. това е едно стабилно сортиране.

Бързо сортиране

С помощта на опростен [псевдокод](https://bg.wikipedia.org/wiki/%D0%9F%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D0%BA%D0%BE%D0%B4), можем да видим реализацията на алгоритъма:

**function** quicksort('array')

**if** length('array') ≤ 1

**return** 'array' *// масив от нула или един елементи е вече сортиран*

select and remove a pivot element 'pivot' from 'array' *// вижте избор на "главен" елемент по-долу*

create empty lists 'less' and 'greater'

**for** each 'x' **in** 'array'

**if** 'x' ≤ 'pivot' then append 'x' to 'less'

**else** append 'x' to 'greater'

**return** concatenate(quicksort('less'), list('pivot'), quicksort('greater')) *// две рекурсивни извиквания*

Сортиране чрез сливане

## **Примерен код на C#**

Има 2 метода. Първият разделя масива на 2, а втория сравнява елементите.

**static** int[] Splits(int[] arr)

{

**if** (arr.Length == 1) *// Ако дължината на масива стане равна на 1 елемент,*

{ *// тогава се връща този елемент и отиваме към другия метод*

**return** arr;

}

*// Инициализират се два масива с равен брой елемента спрямо подаденият масив(arr).*

int middle = arr.Length / 2;

int[] leftArr = **new** int[middle];

int[] rightArr = **new** int[arr.Length - middle];

*// Слагаме в първия масив(left), половината от елементите на подадения масив (arr).*

**for** (int i = 0; i < middle; i++)

{

leftArr[i] = arr[i];

}

*// Слагаме във втория масив(right), другата половината от елементи на подадения масив (arr).*

**for** (int i = middle, j = 0; i < arr.Length; i++, j++)

{

rightArr[j] = arr[i];

}

leftArr = Splits(leftArr); *// Вика се рекурсия на лявата половина, докато нейната дължина не стане равна на 1.*

rightArr = Splits(rightArr); *// След като свършим изцяло с лявата половина на първоначално подадения масив,*

*// прави се същото и с дясната половина, докато не се изчерпят всички нейни стойности*

**return** Merge(leftArr, rightArr); *// Когато в двата масива остане само по 1 елемент, викаме метода*

*// Merge, който ще ги слее, разпределени по големина*

}

Какво е линейно търсене

Линейното търсене е най-простият метод за търсене, който проверява последователно всеки елемент в списък, докато намери определен елемент. Входът за метода на линейно търсене е последователност (като масив, колекция или низ) и елементът, който трябва да се търси. Изходът е true, ако посоченият елемент е в рамките на предоставената последователност или false, ако не е в последователността. Тъй като този метод проверява всеки елемент от списъка, докато не бъде намерен посоченият елемент, в най-лошия случай той ще премине през всички елементи в списъка, преди да намери необходимия елемент. Сложността на линейното търсене е o (n). Следователно се счита за твърде бавен, за да се използва при търсене на елементи в големи списъци. Но това е много просто и по-лесно за изпълнение.

Какво е двоично търсене

Бинарното търсене също е метод, използван за намиране на определен елемент в сортиран списък. Този метод започва чрез сравняване на търсения елемент с елементите в средата на списъка. Ако сравнението определи, че двата елемента са равни, методът спира и връща позицията на елемента. Ако търсеният елемент е по-голям от средния елемент, той отново стартира метода, използвайки само долната половина на сортирания списък. Ако търсеният елемент е по-малък от средния елемент, той отново стартира метода, използвайки само горната половина на сортирания списък. Ако търсеният елемент не е в списъка, методът ще върне уникална стойност, указваща това. Следователно бинарният метод за търсене намалява наполовина броя на сравняваните елементи (във всяка итерация), в зависимост от резултата от сравнението. Следователно,бинарното търсене се изпълнява в логаритмично време, което води до средна производителност на o (log n).

Двоично търсене и линейно търсене

Въпреки че както линейното търсене, така и бинарното търсене са методи за търсене, те имат няколко разлики. Докато двоичното търсене работи върху сортирани списъци, търсенето на линии може да работи и върху несортирани списъци. Сортирането на списък обикновено има средна сложност на n log n. линейното търсене е просто и лесно за изпълнение, отколкото бинарното търсене. Но линейното търсене е твърде бавно, за да се използва с големи списъци поради средното представяне на случая o (n). От друга страна, бинарното търсене се счита за по-ефективен метод, който може да се използва с големи списъци. Но прилагането на двоичното търсене може да бъде доста сложно и проучване показа, че точният код за двоично търсене може да бъде намерен само в пет от двадесет книги.

Интерполационно търсене

Търсене с интерполация е алгоритъм за търсене на ключ в масив, който е подреден по числови стойности, присвоени на ключовете (стойности на ключове). За първи път е описан от W. W. Peterson през 1957 г. Интерполационното търсене прилича на метода, чрез който хората търсят име в телефонния указател (ключовата стойност, по която се подреждат записите в книгата): във всяка стъпка алгоритъмът изчислява къде в останалото пространство за търсене може да бъде търсеният елемент, въз основа на ключа стойности в границите на пространството за търсене и стойността на търсения ключ, обикновено чрез линейна интерполация. След това ключовата стойност, действително намерена при тази прогнозна позиция, се сравнява с търсената ключова стойност. Ако не е равно, тогава в зависимост от сравнението оставащото пространство за търсене се намалява до частта преди или след очакваната позиция. Този метод ще работи само ако изчисленията за размера на разликите между ключовите стойности са разумни.

За сравнение, двоичното търсене винаги избира средата на останалото пространство за търсене, като отхвърля едната или другата половина, в зависимост от сравнението между ключа, намерен в очакваната позиция, и търсения ключ - не изисква числови стойности за ключовете, обща поръчка върху тях. Останалото пространство за търсене се намалява до частта преди или след очакваната позиция. Линейното търсене използва равенство само тъй като сравнява елементи един по един от самото начало, пренебрегвайки всяко сортиране.

Средно търсенето с интерполация прави сравнения в log (log (n)) (ако елементите са равномерно разпределени), където n е броят на елементите за търсене. В най-лошия случай (например, когато числовите стойности на ключовете се увеличават експоненциално) той може да направи сравнения с O (n).

При интерполационно-последователно търсене интерполацията се използва за намиране на елемент в близост до търсения, след което се използва линейно търсене за намиране на точния елемент.

**Изпитна тема № 7:**

Алгоритми и структури от данни План-тезис:

Алчни алгоритми. Рекурсия и рекурсивни алгоритми. Комбинаторни алгоритми: вариации, комбинации, пермутации. Динамично оптимиране. ПРОЕКТ!

Алчни алгоритми

Алчните алгоритми са полезни за решаване на широк спектър от проблеми, особено когато изготвянето на глобално решение е трудно. Понякога си струва да се откажете от сложните планове и просто да започнете да търсите ниско висящи плодове, които приличат на решението, от което се нуждаете. В алгоритмите можете да опишете недалновиден подход като този като алчен.

Търсенето на лесни за разбиране решения представлява основната отличителна характеристика на алчните алгоритми. Алчният алгоритъм достига до решение на проблем, като използва последователни стъпки, при които на всяка стъпка взема решение въз основа на най-доброто решение по това време, без да разглежда бъдещи последици.

Съществуват няколко названия един особен подклас алгоритми на евристичните: жадни, алчни от англ. greedy.

Характерни особености на алчен алгоритъм:

а) за всеки конкретен етап от решението на задачата се взема най-доброто за момента решение като се "надява", че по този начин ще се стигне до глобалния максимум, т.е. реално намира локалния екстремум;

б) не дава възможност за преразглеждане на вече взети решения - главното му отличие от динамичното оптимиране, където решението се взима на базата на всички възможни решения взети до момента;

в) един алчен алгоритъм извежда отговор много близък до оптималния и в повечето случаи е доста ефективен;

г) в общия случай всеки алчен алгоритъм е по-лесен за реализация (за разлика от динамично оптимиране).

Нека разгледаме следната задача илюстрираща работата на алчен алгоритъм:

В България стотинките са с номинална стойност 1, 2, 5, 10, 20 и 50

Да се състави програма, която по въведена сума [10..999] извежда, чрез функция, възможно най-малък брой монети/стотинки, с които може да се плати.

Входни данни: естествено число от интервала [10 .. 999].

Пример: 88 Изход:

50 стотинки - 1 бр.

20 стотинки - 1 бр.

10 стотинки - 1 бр.

5 стотинки - 1 бр.

2 стотинки - 1 бр.

1 стотинкa - 1 бр.

В този случай описваният алчен алгоритъм ще се стреми да ползва най-малък брой монети. Крайният резултат е стремеж за плащане с монети имащи възможно най-голяма номинална стойност.

Рекурсия

При практическото приложение на рекурсия обектът не се дефинира чрез себе си, а чрез по-прости свои версии. Подобно на матрьошките, при отваряне на голямата матрьошка, резултатът е по-опростено копие на същата матрьошка, до достигане на най-малкото нейно копие, опростено до такава степен, че не подлежи на последваща манипулация. Т.е. рекурсивното дефиниране на обект е извикване на неговата опростена версия.

Съществуват различни подходи за онагледяването на рекурсивно извикване, някои от които са стек и дърво.

Рекурсията е явление в програмирането, при което едно понятие се дефинира чрез самото себе си. Тя дава възможност за крайна дефиниция на безкрайно множество. Посредством рекурсия се описват много математически структури и формули, като тяхното моделиране и обработка става удобно чрез рекурсивни методи.

Рекурсията дава възможността да бъдат описани циклични алгоритми без да се използват оператори за цикъл. При повечето декларативни езици, като Лисп и Пролог, основна структура за повторение е рекурсията, а оператори на цикъл липсват.

Рекурсията условно се разделя в две категории: директна (пряка) и индиректна (косвена). Рекурсията е пряка, когато в тялото на подпрограмата има референция към нея. Косвена е тази рекурсия, при която една подпрограма вика друга, а тя вика предходната. Съществуват и случаи на косвена рекурсия, при които подпрограмата извиква себе си, след поредица от обръщения към други подпрограми.

Правилната употреба на рекурсията води до елегантни решения на определени проблеми. Понякога нейното използване опростява значително кода и подобрява четимостта му.

Рекурсивни алгоритми

Непряка рекурсия – подпрограма P1 извиква подпрограма P2, която от своя страна извиква P1. (в общия случай: P1извиква P2, P2 извиква P3, ..., Pn-1 извиква Pn, а Pn извиква P1).

При съставянето на рекурсивни подпрограми, трябва да се спазват няколко важни условия:

Задачата, която разглеждаме, трябва да се разбива на подзадачи, за които рекурсивно може да се приложи същият алгоритъм. Комбинирането на решенията на всички подзадачи трябва да води до решение на изходната задача.

Реализирайки рекурсивен алгоритъм, трябва да бъдем сигурни, че след краен брой стъпки ще достигнем до конкретен резултат, т.е. трябва да се налице краен брой прости случаи (поне един), чието решение може да бъде намерено директно. Прието е тези прости случаи да се наричат дъно на рекурсията.

Всички подзадачи на задачата трябва да се стремят към някой от тези прости случаи, т.е. след краен брой рекурсивни извиквания да се достига дъното на рекурсията.

Примери за рекурсивни подпрограми:

Факториел (P1)

Механизъм на изпълнение на рекурсивната подпрограма: Еднократното извикване на функцията fact(n) в глaвната програма (функция) води до последователност от рекурсивни извиквания на функцията до достигане на простия случай (main()-> fact(n) -> fact(n-1) -> ... -> fact(0)). Така се реализира правия ход на рекурсията. Следва обратният ход на рекурсията, при който последователно завършва изпълнението на всяка една от функциите и връщане на управлението на извикващата функция (fact(0) -> fact(1) -> ... ->fact(n) -> main().

Функцията n! не е дефинирана за отрицателни стойности на n. При извикването на функцията трябва да се гарантира, че n>=0. В противен случай при извикване на функцията с отрицателна стойност на параметъра се получава безкрайна рекурсия (теоретично). На практика програмата се прекратява след запълване на програмният стек (STACK OVERFLOW!).

Комбинаторни алгоритми

Комбинаторните алгоритми (Калгоритми) са нова важна М дисциплина, която намира голямо приложение в КИ. Съществуват три основни фактора за създаването на тази теория. Първо, нарасна практическото значение на Калгоритмите в М, биологията, физиката, химията, социалните и други науки. Второ, налице е бърз прогрес в създаването на нови Калгоритми, в тяхното разбиране и анализ от М гледна точка. Трето, извършва се преход от изучаване на отделни алгоритми към изследване на свойства, които са присъщи на класове алгоритми.

Може да се направи следната аналогия: Калгоритмите се отнасят към комбинаториката (К, комбинаторната М) така, както числените методи на анализа се отнасят към математическия анализ.

Основната задача в К е определянето на Кобекти и намирането на броя им. В Калгоритмите основната задача е образуването (построяването) на всички Кобекти, които имат определени свойства.

При използването на компютърните системи (КС) този термин ще използваме винаги вместо компютър, естествено възниква въпросът за ефективност на Калгоритмите кои от тях се изпълняват по-бързо (с по-малък брой операции) и с по-малко оперативна памет (ОП), т.е изобщо с по-малко компютърни ресурси.

Ще разглеждаме Калгоритми за пораждане на елементарните Кобекти пермутации, комбинации и вариации.

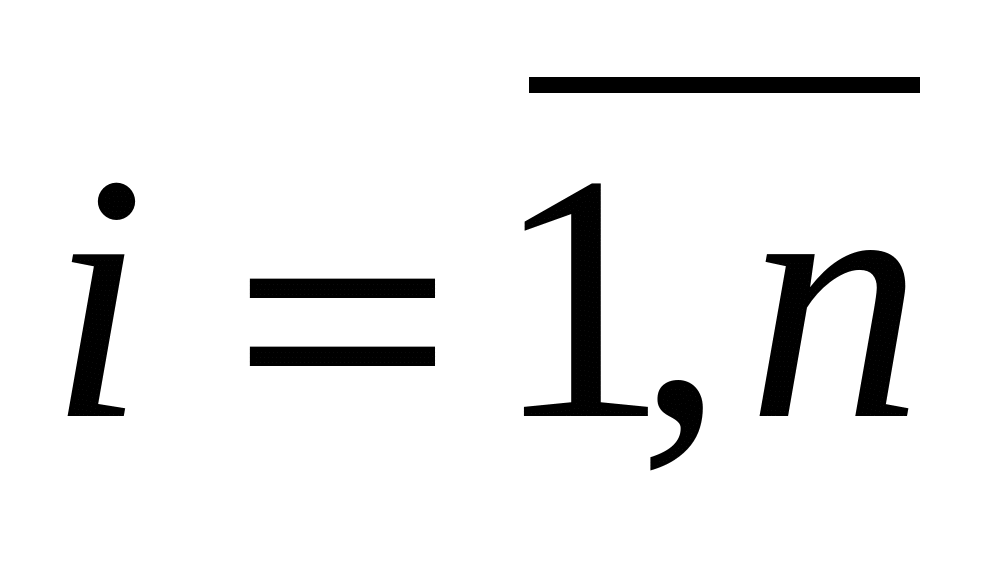
Основни комбинаторни обекти

Кобектите са *пермутации*, *комбинации* и *вариации*.

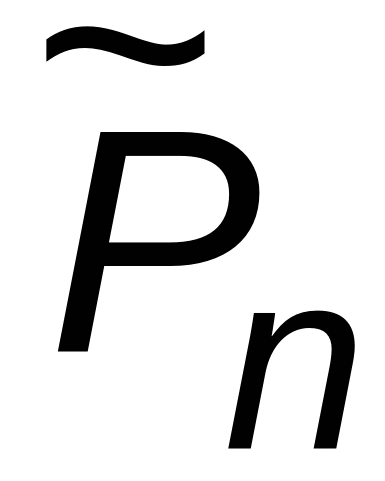
Нека е дадено множеството *А* = {*A*1, *A*2, ..., *An*}, *n* 1, с *различни* елементи. Множество, от елементите на *А,* образувано по комбинаторни правила (вж. по-долу), ще наричаме *комбинаторно съединение* (*К-съединение*). Преди години в българската М се казваше *съчетание*.

К-съединенията могат да се различават по *брой на елементите си* (*дължина на съединението)*, при равен брой по *състав на съединението* (ако в две К-съединения има поне един различен елемент, те са различни), а при равен брой и състав по *мястото* на елементите си (линейна наредба на елементите). Ако в две К-съединения на съответни места има различни елементи, тези съединения са различни.

Пермутации

Това са К-съединения, в които участват *всички* елементи от множеството *А*. Пермутациите се различават една от друга по местата (наредбата) на елементите *Аi*, .

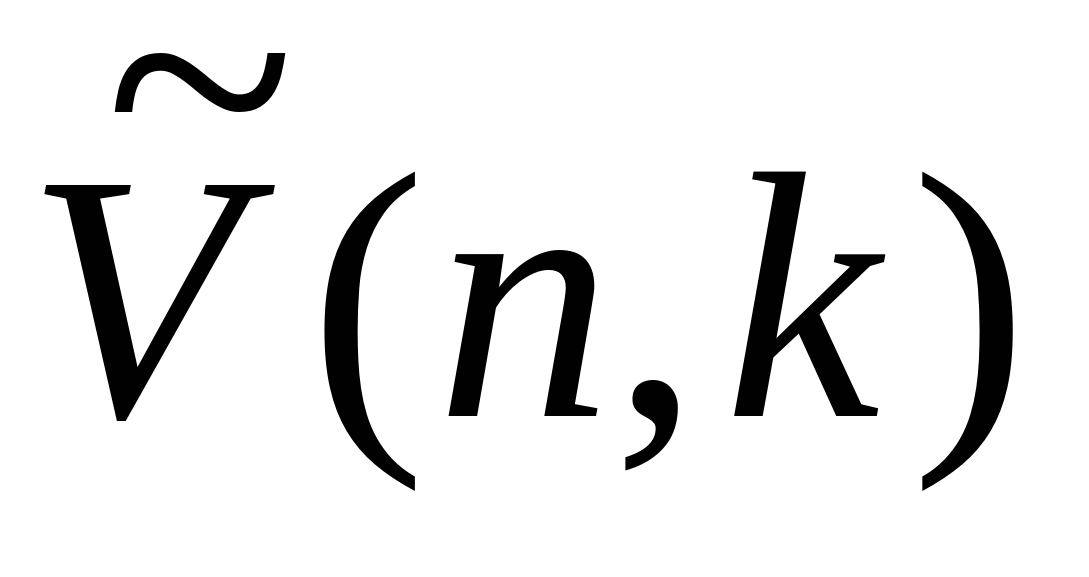
*Пермутации без повторения* са К-съединения, образувани от всички елементи на множеството *А* (те са различни помежду си). Множеството на пермутациите без повторения на *n* елемента, *n* 1, се бележи с *Pn*.

*Пермутации с повторения* са К-съединения от елементите на множеството *А*, като някои от елементите на пермутацията не са различни помежду си (повтарят се). Множеството на пермутациите с повторения на *n* елемента, *n* 1, се бележи с .

Вариации

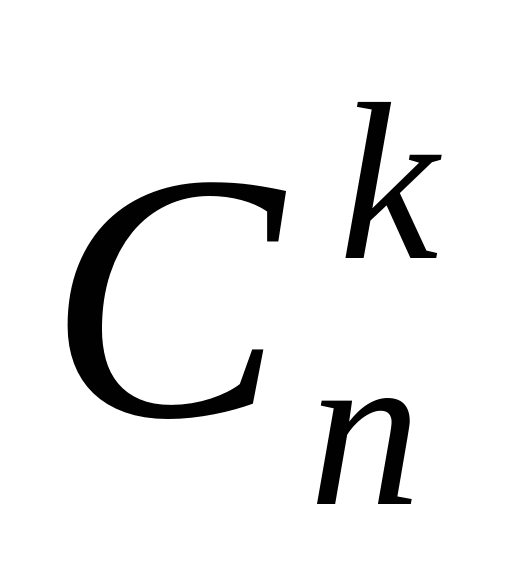
 Това са К-съединения с *k* на брой елемента на множеството *A*, 1 *k* *n*. Вариациите се различават една от друга по мястото на елементите си и/или по състава си.

*Вариации без повторения* са K-съединения от *k* различни елемента на множеството *А*,  
1 *k* *n*. Множеството на вариациите без повторения на *n* елемента от *k-*ти клас се бележи с  или *V*(*n*, *k*). При *k* = *n* вариациите без повторения стават пермутации, *V*(*n*, *n*) = *n*! = *Pn*.

*Вариации с повторения* са K-съединения от *k* елемента на множеството *А*, 1 *k* *n*, като някои елементи на *A* могат да участват и повече от един път. Съответното множество се бележи с  или .

Комбинации

 Това са К-съединения с *k* на брой елемента на множеството *А*, 1 *k* *n*, които се различават само по състава си.

*Комбинации без повторения* на *n* елемента от *k-*ти клас са K-съединения с *k* различни елемента на множеството *А*, 1 *k* *n*. Множеството на комбинациите без повторения на *n* елемента от *k-*ти клас се бележи с  или *C*(*n*, *k*).

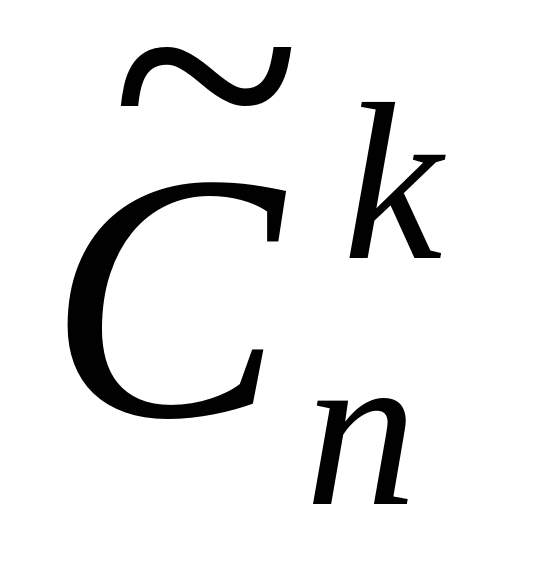
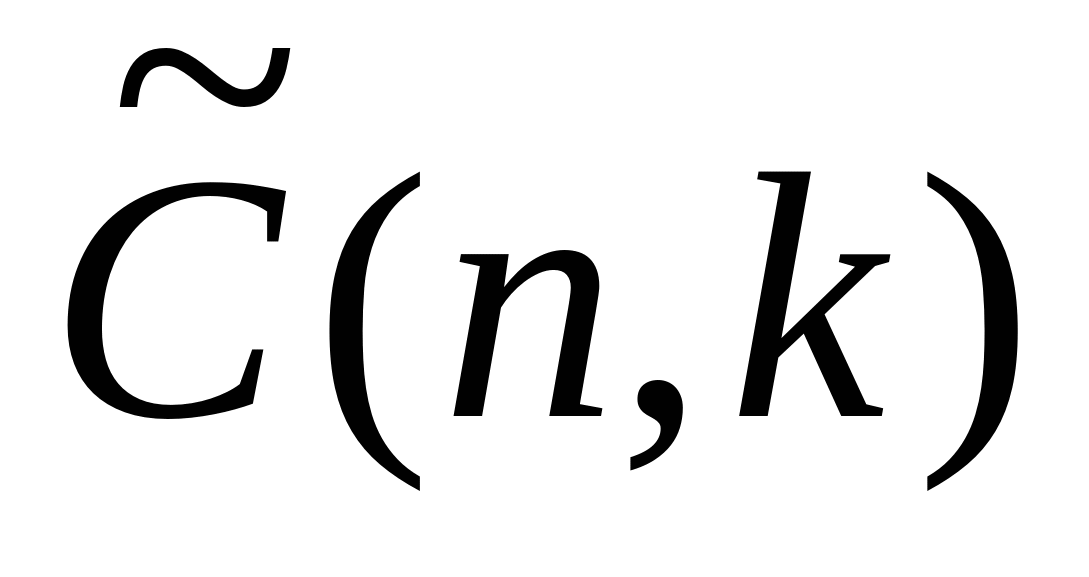
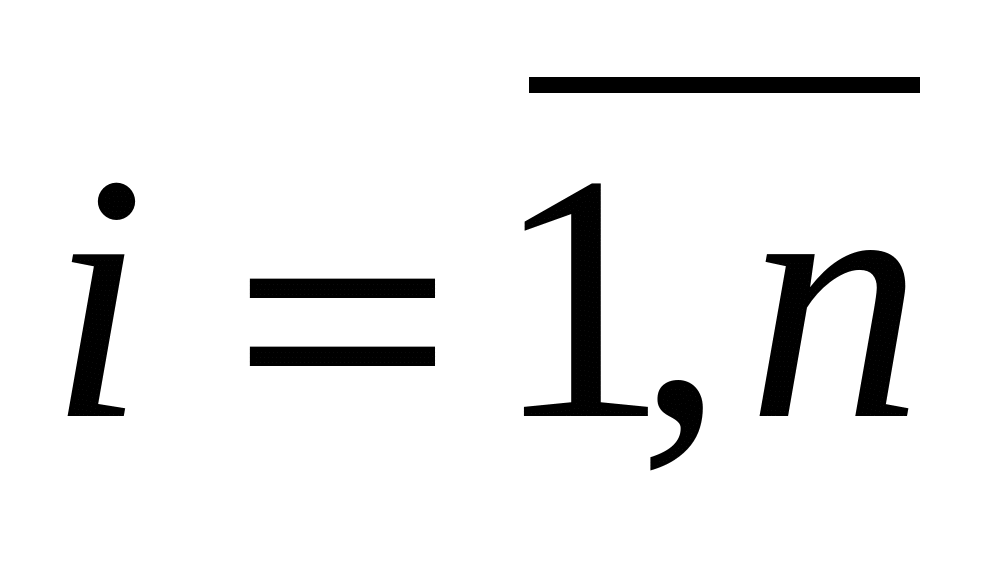
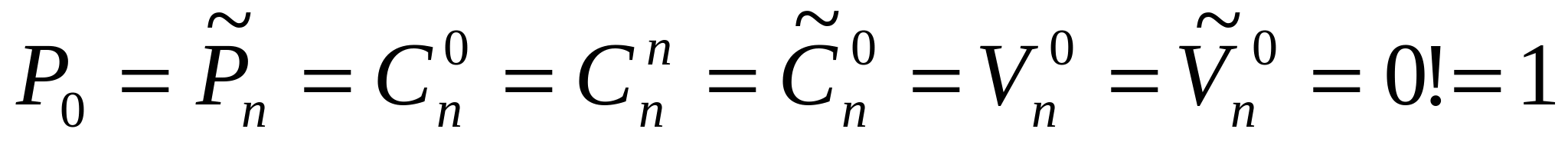
*Комбинации с повторения* са К-съединения от *k* елемента на множеството *А*, 1 *k* *n*, като някои елементи на *А* могат да участват и повече от един път. Съответното множество се бележи с  или .

Таблица 1. Брой на пермутациите, комбинациите и вариациите

|  |  |  |
| --- | --- | --- |
|  | Без повторения | С повторения |
| Пермутации | *Pn* = *n*! |  |
| Вариации |  |  |
| Комбинации |  |  |

К-съединенията пермутации, комбинации и вариации са добре известни обекти в К. Някои от основните задачи се свързват с пресмятането на техния брой. В таблицата по-долу са дадени формулите за намирането на броя на пермутациите, комбинациите и вариациите без и със повторения.

*Mi*, , е броят на повторенията на *i-*тия елемент в пермутацията. По определение .

Без ограничение на общността може да се предполага, че елементите на множеството *A* са целите числа от 1 до *n*, т.е. *А* = {1, 2, ..., *n*} вместо *A* = {*A*1, *A*2, ..., *An*}.

Пораждане на премутации, вариации и комбинации

**Пораждане на пермутации без повторения**

*Рекурсивен алгоритъм за пораждане на пермутации*. При този алгоритъм от една пермутация на *k* елемента (*P*1, *P*2, ..., *Pk* ) може да се образува пермутация от *k*+1 елемента, като поставим *Pk*+1-ия елемент на последно място (*P*1, *P*2, ..., *Pk*, *Pk*+1), на предпоследно място  
(*P*1, *P*2, ..., *Pk*+1, *Pk*) и т.н., на второ място (*P*1, *Pk*+1, *P*2, ..., *Pk*) и най-сетне на първо място (*Pk*+1, *P*1, *P*2, ..., *Pk*).

Динамично оптимиране

Динамичното оптимиране е математически метод за решаване на оптимизационни задачи (търсене на решение, което е най-добро според един или друг критерий). Негов аналог, известен като динамично програмиране, служи за решаване на комбинаторни задачи (преброяване на обекти от определен вид). Поради сходството на двата метода имената им често са взаимозаменяеми.

Динамичното оптимиране се използва в математиката, икономиката, информатиката и биоинформатиката. Същността му се състои в решаването на задача чрез разделянето ѝ на набор от по-прости подзадачи, всяка от които се решава само веднъж, а решенията им се запазват в подходяща структура от данни. Когато решението на някоя подзадача потрябва пак, то се взима наготово. Така се пести време, но за сметка на изразходването на повече памет. Техниката за запомняне на решенията на подзадачите (вместо повторното им решаване) се нарича „мемоизация“.

Оптимизационните задачи могат да се решават не само чрез динамично оптимиране, а и по други начини, например с помощта на т. нар. „алчни алгоритми“ на английски: greedy algorithm. Те построяват търсеното най-добро решение стъпка по стъпка, като на всеки етап избират онзи вариант, който е най-добър за момента. Затова се изпълняват бързо и се програмират лесно, но не винаги са коректни. Динамичното оптимиране е коректно, но по-бавно и изразходва повече памет, тъй като пази не едно възможно решение, а няколко – толкова, колкото е нужно. От друга страна, динамичното оптимиране е много по-бързо от пълното изчерпване, защото разглежда не всички възможни случаи, а само най-малкия необходим брой.

Например в задачата за намиране на минималния брой монети за изплащане на дадена парична сума динамичното оптимиране ще открие най-доброто решение, като първо потърси оптимално решение за всички по-малки суми, а след това ще използва тези решения, за да построи оптималното решение за търсената сума. За сравнение, алчният алгоритъм на всяка стъпка използва монета с възможно най-голяма стойност (която е по-малка от оставащата сума). Той е коректен за система от монети със стойности 1, 2, 5, 10, 20, 50, но не е коректен, ако стойностите на монетите са 1, 4, 5, 15, 20: например за сумата 23 алчният алгоритъм предлага решението 20 + 1 + 1 + 1, което не е оптимално (използва четири монети); най-доброто решение е 15 + 4 + 4 (с три монети).

**Изпитна тема № 8:**

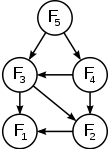
Алгоритми и структури от данни План-тезис:

Динамично оптимиране. Дървовидни структури от данни и алгоритми върху тях. Хеширане и хеш-таблици. Графи и алгоритми върху графи.

Динамично оптимиране в програмирането

За да може да се прилага динамично оптимиране, една задача трябва да има два основни атрибута: оптимална подструктура и припокриващи се подзадачи. Ако една задача може да бъде решена чрез комбиниране на оптимални решения за неприпокриващи се подзадачи, алгоритъмът се нарича „разделяй и владей“. Ето защо сортирането чрез сливане и бързото сортиране не се считат за задачи за динамично оптимиране.

* *Оптимална подструктура* означава, че решението може да се получи от комбинацията на оптималните решения на подзадачите. Такива оптимални подструктури обикновено се описват чрез рекурсия. Такава оптимална подструктура например при граф G = (V, E) e най-краткият път p от връх u до връх v: р е наистина най-краткият път и ако се вземе някой междинен връх w, който е част от p, то р може да се раздели на части p1 (от u до w) и p2 (от w до v) така, че p1 и p2 също са най-кратки пътища между съответните върхове. Следователно, може лесно да се формулира рекурсивен алгоритъм за намиране на най-кратките пътища, което се прави например с алгоритъм на Белман-Форд или алгоритъм на Флойд-Уаршал.

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Fibonacci_dynamic_programming.svg)

Графика на подзадачите за редицата на Фибоначи. Фактът, че структурата не е дървовидна, показва припокриващи се подзадачи.

* *Припокриващи се подзадачи* означава, че пространството на подзадачите трябва да бъде малко, тоест, всеки рекурсивен алгоритъм трябва да решава същите подзадачи отново и отново, а не да създава нови. Например да разгледаме рекурсивен метод за генериране на редицата на Фибоначи: Fi = Fi-1 + Fi-2, с база F1 = F2 = 1. Тогава F43 = F42 + F41 и F42 = F41 + F40. F41 се решава в рекурсивните поддървета на F43 и F42. ако приемем наивно рекурсивно решение като това, ние решаваме отново и отново една и съща подзадача, макар че общият брой всъщност не е голям (само 43).

Динамичното оптимиране отчита този недостатък и решава всяка подзадача само веднъж. Това може да се постигне по два начина:

* От горе надолу: Ако решението на задачата може да се получи чрез решение на подзадачи и ако подзадачите се припокриват, тогава техните решения може да се запазят в таблица. Всеки път, когато е нужно решение на подзадача, се поверява в таблицата дали тази подзадача е вече решена. Ако да, тогава се използва записаното в таблицата решение. В противен случай подзадачата се решава и решението се добавя в таблицата.
* От долу нагоре: След като сме се открили решение на задача рекурсивно, можем да пренаредим задачата отдолу-нагоре: решаваме първо подзадачите и използваме решенията им, за да решим все по-големи задачи. Този метод също обикновено се реализира с матрица. Например, ако сме намерили стойностите на F41 и F40 в примера за редицата на Фибоначи, можем директно да изчислим F42.

Дървовидни структури от данни

[Дървовидните структури от данни](https://bg.wikipedia.org/w/index.php?title=%D0%94%D1%8A%D1%80%D0%B2%D0%BE%D0%B2%D0%B8%D0%B4%D0%BD%D0%B8_%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B8_%D0%BE%D1%82_%D0%B4%D0%B0%D0%BD%D0%BD%D0%B8&action=edit&redlink=1) включват различни типове дървета.[[3]](https://bg.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BE%D1%82_%D0%B4%D0%B0%D0%BD%D0%BD%D0%B8#cite_note-:0-3)

**Дърветата** са мрежови структури от данни, едно от най-важните понятия в теория на графите. Следват три еквивалентни дефиниции на понятието „неориентирано дърво“:

* Свързан граф, съдържащ n върха и n-1 ребра;
* Свързан граф, несъдържащ цикъл;
* Граф, в който всяка двойка върхове е съединена с проста верига:

Ако {\displaystyle G=(X,A)} е неориентиран граф с {\displaystyle n} върха, то всяко дърво, образувано от негови дъги се нарича „покриващо дърво“, ако включва в себе си всички върхове на графа. Очевидно покриващото дърво има {\displaystyle n-1} ребра.

Ориентирано дърво: ориентиран граф без цикли, в който степента-вход на всеки връх (с изключение на един) е равна на 1, а на отбелязания връх изключение (наречен корен) е 0.

***Графи*** са мрежови структури от данни, съвкупност от множеството Х, елементите на което се наричат върхове и множеството А от наредени двойки върхове, наречени дъги (ребра). Означението е (Х, А).

Дървовидна структура

Дървото (или дървовидна структура) в програмирането е рекурсивна структура от данни, която се състои от върхове, които са свързани помежду си с ребра. За дърветата са в сила твърденията:

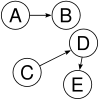
* Всеки връх може да има 0 или повече преки наследници (деца).
* Всеки връх има най-много един баща. Съществува точно един специален връх, който няма предшественици – коренът (ако дървото не е празно).
* Всички върхове са достижими от корена, т.е. съществува път от корена до всички тях.

В практиката често се налага да работа със съвкупност от обекти (данни). Данните, организирани в т. нар. структура от данни, позволяват обработване, така че това да доведе до подобрение качеството на работа с тях. Понякога се добавят елементи, понякога се премахват, друг могат да бъдат подредени по специфичен начин. В програмирането дървото е често използвана структура от данни, която изобразява по естествен начин всякакви йерархии от обекти и тяхната взаимосвързаност.

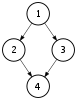
Терминология[

* *Корен* – Най-важния връх в дървото. Няма предшественици.
* *Родител* – предшественик на наследник.
* *Братя* – върхове с общ родител.
* *Непряк наследник* – връх който не е пряк наследник.
* *Дете* – пряк наследник.
* *Прародител* – връх, който е непряк родител.
* *Вътрешен връх* – връх, който има и родител, и дете.
* *Външен връх (или листо)* – връх, който няма наследници.
* *Ребро* – пряка връзка между два върха.
* *Път* – поредица от ребра между върховете.
* *Дължина на път* – броя на ребрата свързващи върховете.
* *Дълбочина на връх* – дължината на пътя от корена до върха.
* *Височина на дърво* – максималната дълбочина.
* *Гора* – съвкупност от несвързани дървета.

Описание

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Directed_graph,_disjoint.svg)

Не е дърво: две несвързани части A→B and C→D→E

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Directed_graph_with_branching_SVG.svg)

Не е дърво:Не пряк кръг, 1-2-4-3

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Directed_graph,_cyclic.svg)

Не е дърво: кръг,B→C→E→D→B

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Graph_single_node.svg)

Не е дърво: кръг,A→A

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Directed_Graph_Edge.svg)

дърво

Дървото се състои от върхове, а линиите посредством които са свързани се наричат ребра. Върхът, който има наследници (деца), но няма родители (предшестеници) се нарича корен (root). Всеки друг връх може да има родител, както и 0 или повече деца. Коренът може да достъпи всеки един връх. Абстрактното разстояние от корена до всяко едно разклонение и всеки връх се нарича път. Един връх не може да има повече от един родител. Следователно в едно дърво не може да има затворен кръг

Тип на данните срещу структура на данните

Съществува разлика между едно дърво като абстрактен тип данни и като конкретна структура от данни, аналогично на разграничението между *списък* и *свързан списък*. Като тип данни, дървото има стойност и деца, като и самите деца са дървета; стойността и децата на дървото се интерпретират като стойността на коренът и на поддървета на синовете на коренът. За да се даде възможност на крайните дървета, списъкът на децата трябва да бъде празен (в този случай дърветата може да не бъдат празни), или дърветата да бъдат празни, в този случай списъкът на децата може да бъде с определени размери, каквито пожелаем.

Като структура от данни, свързано дърво е група от възли (върхове), където всеки възел има стойност и списък с препратки към други възли (наследници). Тази структура данни определя насочена графика,[a], защото може да има цикли или няколко препратки към един и същи възел, както свързания списък може да има цикъл. Следователно не може две препратки да сочат към един и същи възел (всеки възел има точно по 1 родител с изключение на корена) и дърво което нарушава това е „корумпирано“.

Поради използването на препратки към дървета в свързано дърво-структура от данни. Дърветата са често имплицитно обсъждани, приемайки се, че те са представлявани от препратки към корена, тъй като това е често прилагана практика. Например, вместо празно дърво, може да няма препратки:дървото никога не е празно, но препратките към него могат да не съществуват.

Двоично дърво

Дърво, в което броят на наследниците на върховете е 0, 1 или 2 се нарича двоично дърво. Преките наследници на всеки връх са два в този случай, затова единият се нарича ляв наследник, а другият – десен наследник (може и празно множество). Те, от своя страна, са корени съответно на лявото поддърво и на дясното поддърво на техния родител.

Има няколко основни операции при двоичните дървета: добавяне, изтриване, търсене, обхождане. Примерите по-долу са написани на Java.

Двоичното дърво в Java изглежда по този начин:

**public** **class** **Node** {

**private** int data;

**private** Node left;

**private** Node right;

**private** Node parent;

**public** int getData() {

**return** **this**.data;

}

**public** void setData(int data) {

**this**.data = data;

}

**public** Node getLeft() {

**return** **this**.left;

}

**public** void setLeft(Node left) {

**this**.left = left;

}

**public** Node getRight() {

**return** **this**.right;

}

**public** void setRight(Node right) {

**this**.right = right;

}

**public** Node getParent() {

**return** **this**.parent;

}

**public** void setParent(Node parent) {

**this**.parent = parent;

}

}

**Добавяне**

Добавянето на елемент в двоично дърво е операция, която има **логаритмична сложност** по отношение на размера на дървото. Всеки елемент в двоичното дърво има добре определена позиция, затова първо трябва да намерим къде трябва да го поставим, а после и да го направим. Следва и примера за добавяне:

**public** void insert(int data) {

root = insert(root, data, **null**);

}

**private** Node insert(Node current, int data, Node parent) {

**if** (current == **null**) {

current = **new** Node();

current.setData(data);

current.setLeft(**null**);

current.setRight(**null**);

current.setParent(parent);

} **else** **if** (data < current.getData()) {

current.setLeft(insert(current.getLeft(), data, current));

} **else** {

current.setRight(insert(current.getRight(), data, current));

}

**return** current;

}

**Търсене**

Търсенето изглежда като добавянето и отнема **логаритмично време** (**logn**).

**public** Node find(int data) {

**return** find(root, data);

}

**private** Node find(Node current, int data) {

**if** (current == **null**) {

**return** **null**;

}

**if** (current.getData() == data) {

**return** current;

}

**return** find(data < current.getData() ? current.getLeft() : current.getRight(), data);

}

Двоичното дърво има някои интересни характеристики, позволяващи ни да намерим специални елементи много лесно. Например минималният и максималният елемент. Най-малката стойност е тази, която се намира най-вляво, а максималната – най-вдясно.

**public** Node findMin() {

Node min = root;

**if** (min == **null**) {

**return** **null**;

}

**while** (min.getLeft() != **null**) {

min = min.getLeft();

}

**return** min;

}

**Изтриване**

Изтриването на елемент от дърво е най-трудната операция. Тя също отнема **logn** време. Има няколко специални ситуации, които трябва да бъдат овладени:

1. Изтриване на елемент без деца – просто освобождаваме паметта.
2. Изтриване на елемент само с едно дете – промяна на указателите на родителя да сочат директно към детето на изтривания елемент и освобождаване на паметта.
3. Изтриване на елемент само с едно дете и това е КОРЕНЪТ – преместване на детето на мястото на корена и освобождаване на паметта.
4. Изтриване на елемент с две деца – това е най-сложната операция. Най-добрият начин за изпълнение е да бъдат разменени стойностите на изтривания елемент и максималната стойност на лявото поддърво или минималната на дясното поддърво (защото това ще запази характеристиките на дървото) и тогава да изтрием елемент без или с едно дете.

**public** void delete(int data) {

root = delete(root, data);

}

**private** Node delete(Node startNode, int data) {

Node element = find(startNode, data);

**if** (element == **null**) {

**return** startNode;

}

boolean hasParent = element.getParent() != **null**;

boolean isLeft = hasParent && element.getData() < element.getParent().getData();

**if** (element.getLeft() == **null** && element.getRight() == **null**) {

**if** (hasParent) {

**if** (isLeft) {

element.getParent().setLeft(**null**);

} **else** {

element.getParent().setRight(**null**);

}

}

} **else** **if** (element.getLeft() != **null** && element.getRight() == **null**) {

**if** (hasParent) {

**if** (isLeft) {

element.getParent().setLeft(element.getLeft());

} **else** {

element.getParent().setRight(element.getLeft());

}

} **else** {

startNode = element.getLeft();

}

} **else** **if** (element.getLeft() == **null** && element.getRight() != **null**) {

**if** (hasParent) {

**if** (isLeft) {

element.getParent().setLeft(element.getRight());

} **else** {

element.getParent().setRight(element.getRight());

}

} **else** {

startNode = element.getRight();

}

} **else** {

Node rightMin = findMin(element.getRight());

element.setData(rightMin.getData());

**return** delete(rightMin, rightMin.getData());

}

element = **null**;

**return** startNode;

}

**Обхождане на дърво**[[редактиране](https://bg.wikipedia.org/w/index.php?title=%D0%94%D1%8A%D1%80%D0%B2%D0%BE_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BE%D1%82_%D0%B4%D0%B0%D0%BD%D0%BD%D0%B8)&veaction=edit&section=8) | [редактиране на кода](https://bg.wikipedia.org/w/index.php?title=%D0%94%D1%8A%D1%80%D0%B2%D0%BE_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BE%D1%82_%D0%B4%D0%B0%D0%BD%D0%BD%D0%B8)&action=edit&section=8)]

Съществуват три типа обхождане на двоични дървета:

1. Inorder – посещаване на лявото поддърво, корена и дясното поддърво.
2. Preorder – посещаване на корена, лявото поддърво и дясното поддърво.
3. Postorder – посещаване на лявото поддърво, дясното поддърво и корена.

**public** void traverseTree(TraverseType traverseType) {

traverseTree(root, traverseType);

System.out.println();

}

**private** void traverseTree(Node current, TraverseType traverseType) {

**if** (current == **null**) {

**return**;

}

**switch** (traverseType) {

**case** INORDER:

traverseTree(current.getLeft(), traverseType);

processNode(current);

traverseTree(current.getRight(), traverseType);

**break**;

**case** PREORDER:

processNode(current);

traverseTree(current.getLeft(), traverseType);

traverseTree(current.getRight(), traverseType);

**break**;

**case** POSTORDER:

traverseTree(current.getLeft(), traverseType);

traverseTree(current.getRight(), traverseType);

processNode(current);

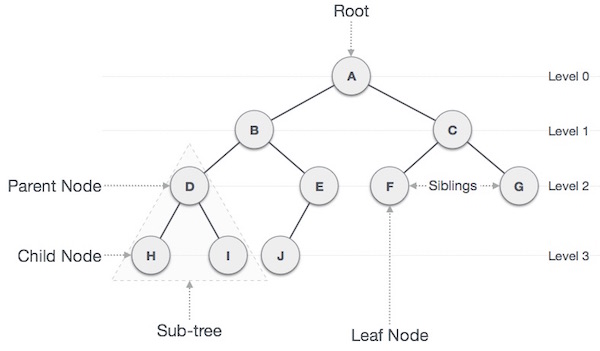
**break**;

}

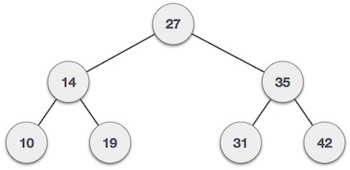
}

Дървовидна структура и алгоритми върху тях

Дървото представлява възлите, свързани с ръбове. Ще обсъдим конкретно бинарното дърво или бинарното дърво за търсене.

Бинарното дърво или „Binary Tree“ е специална структура от данни, използвана за съхранение на данни. Бинарното дърво има специално условие, че всеки възел може да има максимум две деца. Двоичното дърво има предимствата както на подреден масив, така и на свързан списък, тъй като търсенето е толкова бързо, колкото в сортиран масив, а операцията за вмъкване или изтриване е толкова бърза, колкото в свързания списък.

Представяне на бинарното дърво за търсене

Дървото на двоичното търсене показва специално поведение. Лявото дете на възел трябва да има стойност, по-малка от стойността на родителя си, а дясното дете на възела трябва да има стойност, по-голяма от родителската му стойност.

Ще приложим дърво, като използваме обект на възел и ще ги свържем чрез препратки.

Дървесен възел

(Tree Node)

Кодът за писане на дървесен възел би бил подобен на този, даден по-долу. Той има част с данни и препратки към своите леви и десни дъщерни възли.

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

В дърво всички възли споделят обща конструкция.

Основни операции на BST

Основните операции, които могат да бъдат извършени върху двоична структура с данни на дървото за търсене, са следните -

Вмъкване - Вмъква елемент в дърво / създава дърво.

Търсене - Търси елемент в дърво.

Прекратяване на предварителна поръчка - обхожда дърво по начин на предварителна поръчка.

Inorder Traversal - Преминава дърво по ред.

Пренасочване след поръчка - Преминава дърво по начин след поръчка.

Ще се научим да създаваме (вмъкваме в) дървесна структура и да търсим елемент от данни в дърво в тази глава. Ще научим за методите за обхождане на дървета в следващата глава.

Операция за вмъкване

(Insert Operation)

Първото вмъкване създава дървото. След това, винаги, когато трябва да се вмъкне елемент, първо намерете правилното му местоположение. Започнете да търсите от основния възел, след това ако данните са по-малки от ключовата стойност, потърсете празното място в лявото поддърво и вмъкнете данните. В противен случай потърсете празното място в дясното поддърво и вмъкнете данните.

Алгоритъм Операция за вмъкване

(Algorithm of insert Operation)

If root is NULL

then create root node

return

If root exists then

compare the data with node.data

while until insertion position is located

If data is greater than node.data

goto right subtree

else

goto left subtree

endwhile

insert data

end If

Изпълнение (Implementation)

Изпълнението на функцията за вмъкване трябва да изглежда така –

void insert(int data) {

struct node \*tempNode = (struct node\*) malloc(sizeof(struct node));

struct node \*current;

struct node \*parent;

tempNode->data = data;

tempNode->leftChild = NULL;

tempNode->rightChild = NULL;

//if tree is empty, create root node

if(root == NULL) {

root = tempNode;

} else {

current = root;

parent = NULL;

while(1) {

parent = current;

//go to left of the tree

if(data < parent->data) {

current = current->leftChild;

//insert to the left

if(current == NULL) {

parent->leftChild = tempNode;

return;

}

}

//go to right of the tree

else {

current = current->rightChild;

//insert to the right

if(current == NULL) {

parent->rightChild = tempNode;

return;

}

}

}

}

}

Операция за търсене

(Search Operation)

Винаги, когато трябва да се търси елемент, започнете да търсите от основния възел, а ако данните са по-малки от ключовата стойност, потърсете елемента в лявото поддърво. В противен случай потърсете елемента в дясното поддърво. Следвайте един и същ алгоритъм за всеки възел.

If root.data is equal to search.data

return root

else

while data not found

If data is greater than node.data

goto right subtree

else

goto left subtree

If data found

return node

endwhile

return data not found

end if

Изпълнението на този алгоритъм трябва да изглежда така.

struct node\* search(int data) {

struct node \*current = root;

printf("Visiting elements: ");

while(current->data != data) {

if(current != NULL)

printf("%d ",current->data);

//go to left tree

if(current->data > data) {

current = current->leftChild;

}

//else go to right tree

else {

current = current->rightChild;

}

//not found

if(current == NULL) {

return NULL;

}

return current;

}

}

Алгоритъм на операцията за търсене

(Search Operation Algorithm)

If root.data is equal to search.data

return root

else

while data not found

If data is greater than node.data

goto right subtree

else

goto left subtree

If data found

return node

endwhile

return data not found

end if

Изпълнението на този алгоритъм трябва да изглежда така.

struct node\* search(int data) {

struct node \*current = root;

printf("Visiting elements: ");

while(current->data != data) {

if(current != NULL)

printf("%d ",current->data);

//go to left tree

if(current->data > data) {

current = current->leftChild;

}

//else go to right tree

else {

current = current->rightChild;

}

//not found

if(current == NULL) {

return NULL;

}

return current;

}

}

Хеширане

Хеширане (от английски hashing) е действието, при което асоциираме даден обект с хеш стойност. Тъй като самото хеширане много зависи от обектите, върху които го прилагаме, а също така и какви хеш стойности ще използваме (може да са един int, няколко int-а, последователност от битове, или нещо друго), това е по-скоро техника, отколкото алгоритъм.

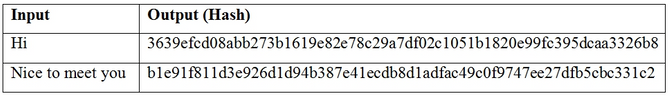
Хеширане е криптографско представяне на набор от информация. Независимо от първоначалния размер (това може да бъде файл от всякакъв формат, с размер от 1 байт или 100 терабайта и повече), а изходът ще бъде низ с фиксирана дължина.

Хеш (от английски hash), или хеш стойност, е удобен за нас "псевдоним" на някакъв сложен обект. По псевдонима обикновено не можем да възстановим целия обект, но все пак можем да извършваме някои по-прости операции - като например да видим дали два обекта (например стринга) са еднакви или не. Най-често хешът е число, но в някои случаи може да бъдат две или повече числа, или последователност от битове. По-общо, хешът е представяне на данни с произволен размер (един стринг може да има произволна дължина) чрез данни с фиксиран размер (размерът на int в байтове е фиксиран).

Говорейки за това как работи блокчейнът, на първо място, трябва да говорите за хеширане. Биткойн блокчейнът използва алгоритъма за хеширане SHA-256 с 256-битова изходна последователност. Всеки блокчейн се различава от останалите по използвания алгоритъм за криптиране, което води до интересно разнообразие от криптография за отделни решения на блокчейн.

Хеширане на биткойн е преобразуване на масив от данни с всякакъв размер в оригинален низ с фиксирана дължина. За по-лесно разбиране, нека ги наречем Input (входящият масив от данни) и Output (полученият хеш, т.е. низ с фиксирана дължина, който вече беше споменат по-горе). Ето как работи хеширането на биткойни.

Нека разгледаме един малък пример за това как работи процесът. За да преобразуваме масив от информация в хеш, ще използваме SHA-256 (Secure Hashing Algorithm) алгоритъм за хеширане, който вече беше споменат по-горе:



Хеш-таблици

Хеш таблицата е структура от данни, съдържаща ключ и данни, която се характеризира с директен достъп до елементите, независимо от типа им. Елементите ѝ, подобно тези на други структури от данни използвани за търсене, се състоят от ключ и данни. Ключът е уникален за разлика от данните – не може да съществуват два елемента с един и същи ключ. Сложността на елементарните операции по ключ (търсене, вмъкване, изтриване и актуализиране) в общия случай е константна, което я прави изключително полезна.

Приложение на Хеш-таблиците

При процеса на компилиране на дадена програма всеки идентификатор в нея се съхранява в определена структура (таблица), в която за него има полета и за друга информация – тип, адрес от паметта и други. Когато се срещне такъв идентификатор трябва да се определи дали той вече не е сред множеството идентификатори, които вече са в таблицата. Последователното обхождане и поддържането в сортиран вид на тези структури са бавни операции. Най-удачна реализация на операциите търсене, четене и вмъкване в такова множество от идентификатори (ключове) е хеширането (от англ. „hash, hashing“). Хеширането е много добър пример за компромисно решение на проблема памет-време. При хеширането ключът се преобразува чрез определена хеш функция в адрес на таблица (хеш таблица) т.е. на дадени данни се съпоставят други, посредством правила, но така че обратната операция да е невъзможна (ако хеширате даден низ, с хеш функция, то от резултата няма как да получите обратно първоначалния низ).

Операции за Хеш-таблици

Основните операции за работа с хеш таблица са следните три:

* Void put (key, data); – включване на елемент с ключ (key) и данни (data) в хеш таблицата
* Data get (key); – търсене и извличане на данните от елемент с ключ (key) от хеш таблицата
* Void remove (key); – изключване на елемент с ключ (key) от хеш таблицата

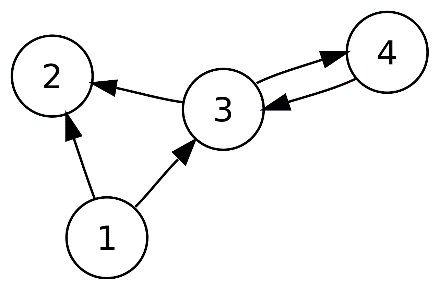
Всеки елемент на хеш таблицата се характеризира с две полета: ключ (key) и данни (data). Ключът представлява уникален идентификатор: ако два елемента имат един и същ ключ се считат идентични.

Граф

В компютърните науки, граф (мн. ч. Графи) е абстрактна структура от данни, имаща за цел да имплементира терминът граф от математиката.

Графът като структура от данни представлява връзките между отделните елементи на дадено множество. Всеки член на това множество се нарича връх, а връзката между два върха се нарича ребро. Честата употреба на графите в практиката е довела до задълбочени изследвания в теория на графите, в която са известни огромен брой задачи за графи и за повечето от тях има и добре известно решение.

Имплементацията на граф може да асоциира към всеки връх дадена стойност, като буквено-цифрово означение или дадена стойност(константа, капацитет, дължина и т.н.).



Граф (Основни понятия)

Дефиниция

Термин от математиката, с който се означава наредена двойка G=(V,E), където

V е множество от елементи, наречени върхове,

E е множество от двучленни подмножества на V, т.е. E ⊆ V×V. Когато в тези двучленни подмножества няма наредба, т.е. формират ненаредени двойки, е прието да се наричат ребра или още ръбове на графа, а той от своя страна – неориентиран (ненасочен) граф. Когато тези двойки са наредени, елементите на Е се наричат дъги, а графът G – ориентиран (насочен) граф.

Връх

Връх на граф (на английски език: vertex или node) – елементи, съвкупността от които, свързани в подмножества образуват елемент от наредената двойка репрезентирана от графа.

Ребро (Дъга)

Дъга (англ.: arc) за ориентиран граф, ребро (англ.: edge) за неориентиран – това е релация между два върха в графа. В общия случай, дъга (i, j) бележи дали има път от i-тия връх до j-ия, като i-ия се нарича опашка (англ: tail), а j-ия – глава на дъгата (англ.: head). Често дъгите са свързани и със съответна стойност за преход през тях, която се нарича тегло на дъгата (англ.: edge value).

Път

Път в ориентиран (неориентиран) граф G(V,A) се нарича последователност от върхове v1, v2, … vk, такива че за всяко i = 1, 2… k-1 е в сила (vi,vi+1)∈A. Върховете v1 и vk се наричат краища на пътя. Ако v1 = vk, то има цикъл. Ако няма повтаряне на върхове в даден път (в частност цикъл), тогава пътят е прост (за всяко i≠j (1≤i,j≤k) следва vi≠vj). Ако даден граф има път от всеки връх до всеки друг – графът е свързан. С O(n) сложност, където n е броят на дъгите, може да се определи дали даден граф е свързан или не (DFS).

Дължина на път е броят на ребрата, свързващи последователността от върхове в пътя. Този брой е равен на броят на върховете в пътя минус единица.

Цена на път в претеглен граф, се нарича сумата от теглата на ребрата участващи в пътя.

Цикъл (англ.: loop) в графа G е път, за който началният и крайният връх съвпадат.

Прост цикъл – цикъл, в който няма повтарящи се възли.

Хамилтонов цикъл – цикъл, който включва всички възли на графа точно по веднъж.

Ойлеров цикъл – цикъл, който включва всички ребра на графа точно по веднъж.

Видове графи

Ориентиран граф (англ.: directed graph) – Фир. 1 – ребрата са насочени, изобразяват се чрез стрелки. Две ребра, свързващи еднакви върхове, но различно ориентирани, за по-голяма прегледност се изобразяват с една двупосочна стрелка.

Краен неориентиран граф (англ.: undirected graph) се нарича наредената двойка (V,E), където:

V = {v1,v2,…,vn} е крайно множество от върхове

E = {e1, e2, …,em} е крайно множество от неориентирани ребра.

Претеглен(тегловен) (англ.:weighted graph) – на всяко ребро е присвоена някаква стойност – тегло.

Мултиграф (англ.: multigraph) – възможно е повече от едно ребро да свързва два върха (при ориентиран граф – възможно е тези ребра, освен това, да са ориентирани еднакво).

Регулярен граф – граф, при който всеки връх има равен брой съседни върхове, т.е. всички върхове на графа са от една и съща степен.

Граф алгоритми

**Графовите алогоритми са значителна област на интерес в компютърните науки. Типични орерации от високо ниво, свързани с графите за намиране на път между два върха:**

* Обхождане в дълбочина.

Обхождане в дълбочина (на английски: Depth-First Search (DFS)) е алгоритъм за обхождане на структури от данни, и по-специално дърво и граф. За реализация на алгоритъма се избира даден връх (възел) от структурата, който се обозначава като корен (връх без предшественици) и обхождането стартира от него. Последователно се посещават всички следващи върхове до достигането на връх без наследници (от който не излизат ребра), след което се осъществява търсене с връщане назад (на англ. backtracking) до достигане на нова крайна точка или при цялостно реализирано обхождане – към корена.

Шарл Пиер Тремо прилага версия на алгоритъма през 19 век за решаване на задачи с лабиринти.

* Обхождане в ширина.

В теорията на графите, обхождането в ширина е начин за търсене в граф, когато търсенето се ограничава до две основни операции:

посещаване и достъпване на разклонение от графа;

получаване на достъп до съседите на посещаваното в даден момент разклонение.

Обхождането в ширина започва от главното разклонение и достъпва всички съседни разклонения. За всяко едно от тези съседни разклонения, проверява техните съседни разклонения, които не са били посетени и така нататък.

**Намиране на най-кратък път между два върха:**

* Алгоритъм на Дейкстра.

Алгоритъмът на Дейкстра, наречен на автора си Едсхер Дейкстра (Edsger Dijkstra), служи за пресмятане на най-къс път от даден връх до всички останали върхове на граф с неотрицателни тегла на ребрата.

Графът може да бъде ориентиран или неориентиран. Резултатът от работата на алгоритъма е дърво на най-късите пътища с начало дадения връх. Алгоритъмът се използва още за намиране на най-къс път от един даден връх до друг даден връх; в този случай алгоритъмът спира работа, след като е намерил най-краткия път между тези два върха.

Пример: Всяка пътна карта може да се разглежда като граф. Върховете на графа са градовете, а ребрата са шосетата между градовете.

* Алгоритъм на Белман-Форд.

Алгоритъмът на Белман—Форд намира най-късите пътища от един връх до всички останали върхове в насочен тегловен граф. По-бавен е от алгоритъма на Дейкстра, но е много по-гъвкав, когато се обхождат графи с отрицателни тегла на ребрата. Алгоритъмът е именуван на двама от създателите си – Ричард Белман и Лестър Форд младши. Публикуван е през 1958 г. и 1956 г. съответно. Едуард Ф. Мур публикува същия алгоритъм през 1957 г. и поради тази причина алгоритъмът се среща понякога като алгоритъм на Белман—

* Алгоритъм на Флойд(на английски език: Floyd–Warshall algorithm).

**Изпитна тема № 9:**

Бази от данни План-тезис:

Въведение в базите от данни. Типове данни. Основни команди. Моделиране на релационни бази от данни. Заявки за извличане и промяна на данни. Сложни заявки за извличане на данни. Съединения на таблици (SQL JOIN). Агрегиращи функции. Групиране на данни. Скаларни функции, транзакции, съхранени процедури, тригери. Примерна приложна задача: По време на теоретичния изпит се предоставя непълен/неработещ/некоректен програмен фрагмент на приложната задача. Предоставеният фрагмент да се приведе в работещ вид. Условие: Като използвате дадената ER диаграма напишете заявките дадени по-долу:

Въведение в базите от данни

База данни (БД, още база от данни) представлява колекция от логически свързани данни в конкретна предметна област, които са структурирани по определен начин. В първоначалния смисъл на понятието, използван в компютърната индустрия, базата от данни се състои от записи, подредени систематично, така че компютърна програма да може да извлича информация по зададени критерии. Например БД може да се използват в моделирането на хотелските системи, за да се проверява дали има налични свободни стаи в даден хотел.

Поддръжката на база от данни се осъществява от т.нар. система за управление на бази от данни (СУБД).

Система за управление на бази данни е компютърно приложение (софтуер) създадено за комуникация между потребителя, други приложения, както и други БД, с цел да се сравнят и анализират данни. Общото специфично предназначение на СУБД е да позволи определянето, създаването, заявки, актуализацията и администрирането на бази данни. Добре известни СУБД включват MySQL, PostgreSQL, Microsoft SQL Server, Oracle, Sybase, SAP HANA, и IBM DB2. Бази данни не са съвместими с различните СУБД, за това различните СУБД работят със стандартни като SQL и ODBC или JDBC, за да позволи на всяко приложение да работи с различни СУБД, а така и с различни БД. Управлението на БД често се избира от модела им, които те подкрепят. Най-използвани системи от бази данни от 1980 г. насам са всички поддържани релационния модели на езика SQL. Често срещано е СУБД да се нарича само „база данни“.

Типове данни

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Схематично описание на основни характеристики на Примитивни типове данни в С# (C Sharp)** | | | | |
| **Име** | **Размер в паметта** | **Стойност по подразбиране** | **Минимална стойност** | **Максимална стойност** |
| sbyte | 8 bits | 0 | -128 (-{\displaystyle 2^{7}}) | 127 (-{\displaystyle 2^{7}} – 1) |
| byte | 8 bits | 0 | 0 | 255 ({\displaystyle 2^{8}} – 1) |
| short | 16 bits | 0 | -32 768 (-{\displaystyle 2^{15}} – 1) | 32 767 ({\displaystyle 2^{15}}) |
| ushort | 16 bits | 0 | 0 | 65 535 ({\displaystyle 2^{16}} – 1) |
| int | 32 bits | 0 | -2 147 483 648 (-{\displaystyle 2^{31}}) | 2 147 483 647 ({\displaystyle 2^{31}} – 1) |
| uint | 32 bits | 0u или 0U | 0 | 4 294 967 295 ({\displaystyle 2^{32}} – 1) |
| long | 64 bits | 0l или 0L | -9 223 372 036 854 775 808 (-{\displaystyle 2^{63}}) | 9 223 372 036 854 775 807 ({\displaystyle 2^{63}} – 1) |
| ulong | 64 bits | 0l или 0L | 0 | 18 446 744 073 709 551 615 ({\displaystyle 2^{64}} – 1) |
| float | 32 bits | 0.0f или 0.0F | ±1.5 × {\displaystyle 10^{-45}} | ±3.4 × {\displaystyle 10^{38}} |
| double | 64 bits | 0.0d или 0.0D | ±5.0 × {\displaystyle 10^{-324}} | ±1.7 × {\displaystyle 10^{308}} |
| decimal | 128 bits | 0.0m или 0.0M | -7.9 × {\displaystyle 10^{-28}} | +7.9 × {\displaystyle 10^{28}} |
| bool | 4 bits | false | само две възможности: bool или false |  |
| char | 16 bits | ‘\u0000′ | 0 или ‘\u0000′ | 65535 или ‘\uffff’ |
| object | - | null | - | - |
| string | - | null | - | - |

Основни команди

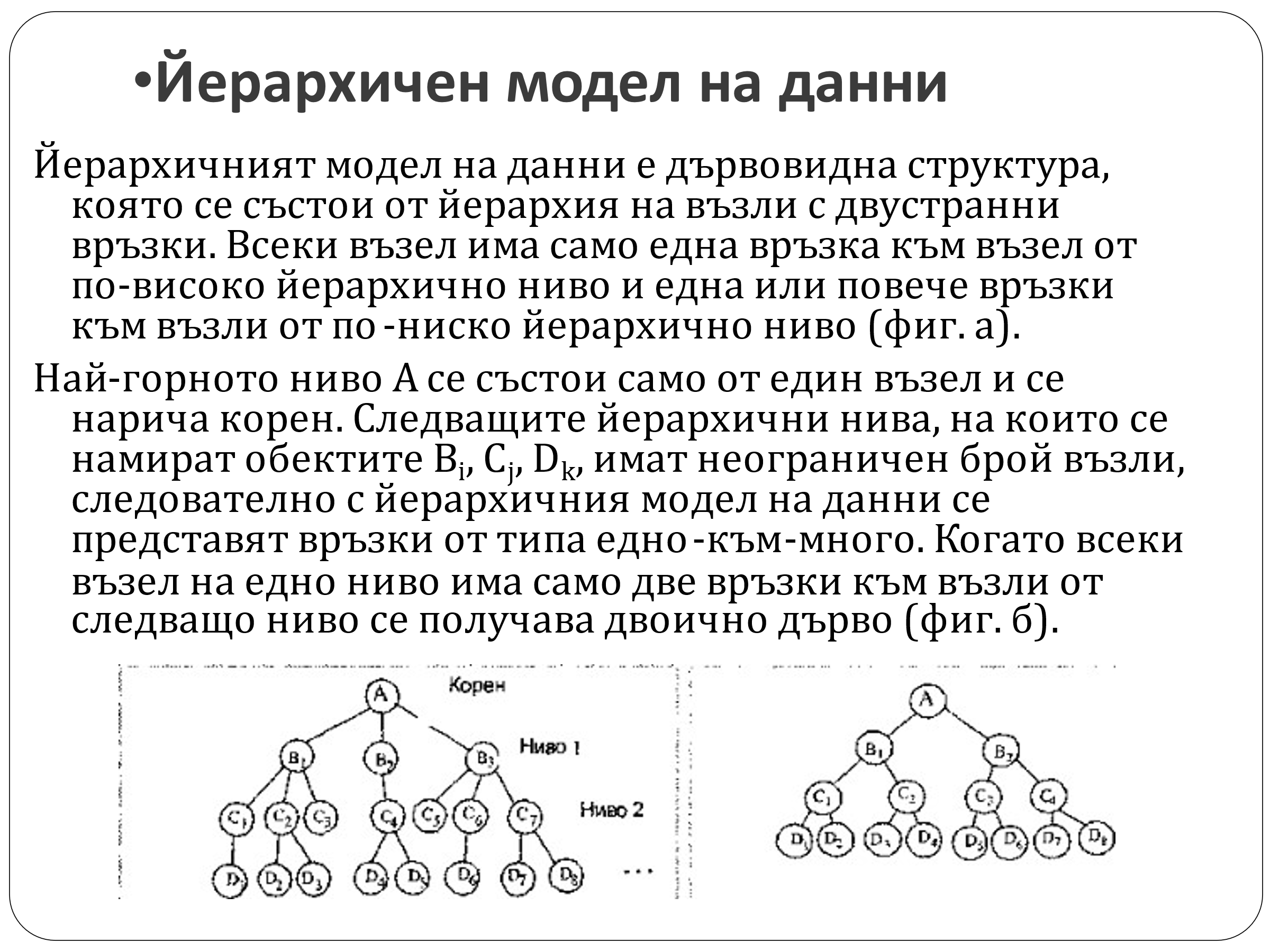
| **SQL клауза** | **Какво прави** | **Задължително ли е** **?** |
| --- | --- | --- |
| SELECT | Съдържа списък на полетата, които съдържат данни, представляващи интерес. | Да |
| FROM | Съдържа списък на таблиците, съдържащи полетата, изброени в клаузата SELECT. | Да |
| WHERE | Задава критерии за полета, които трябва да бъдат изпълнени от всеки запис, за да бъде включен в резултатите. | Не |
| ORDER BY | Задава как да се сортират резултатите. | Не |
| GROUP BY | В SQL команда, съдържаща агрегатни функции, съдържа списък на полета, които не са обобщени в клаузата SELECT. | Само ако има такива полета |
| HAVING | В SQL команда, съдържаща агрегатни функции, задава условия към полета, които са обобщени в командата SELECT. | Не |

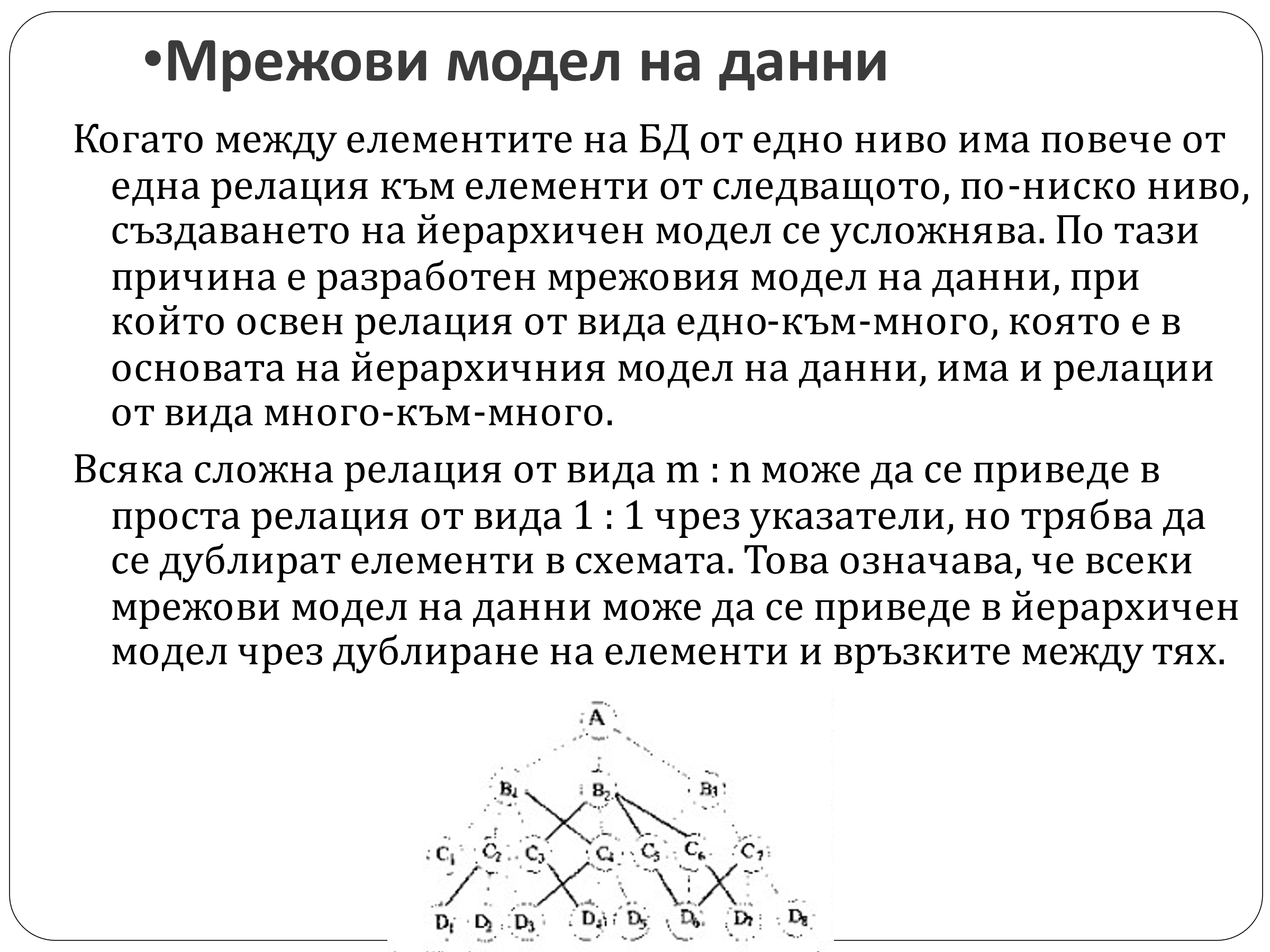
Всяка SQL клауза е съставена от условия. Ето списък с някои общи SQL условия.

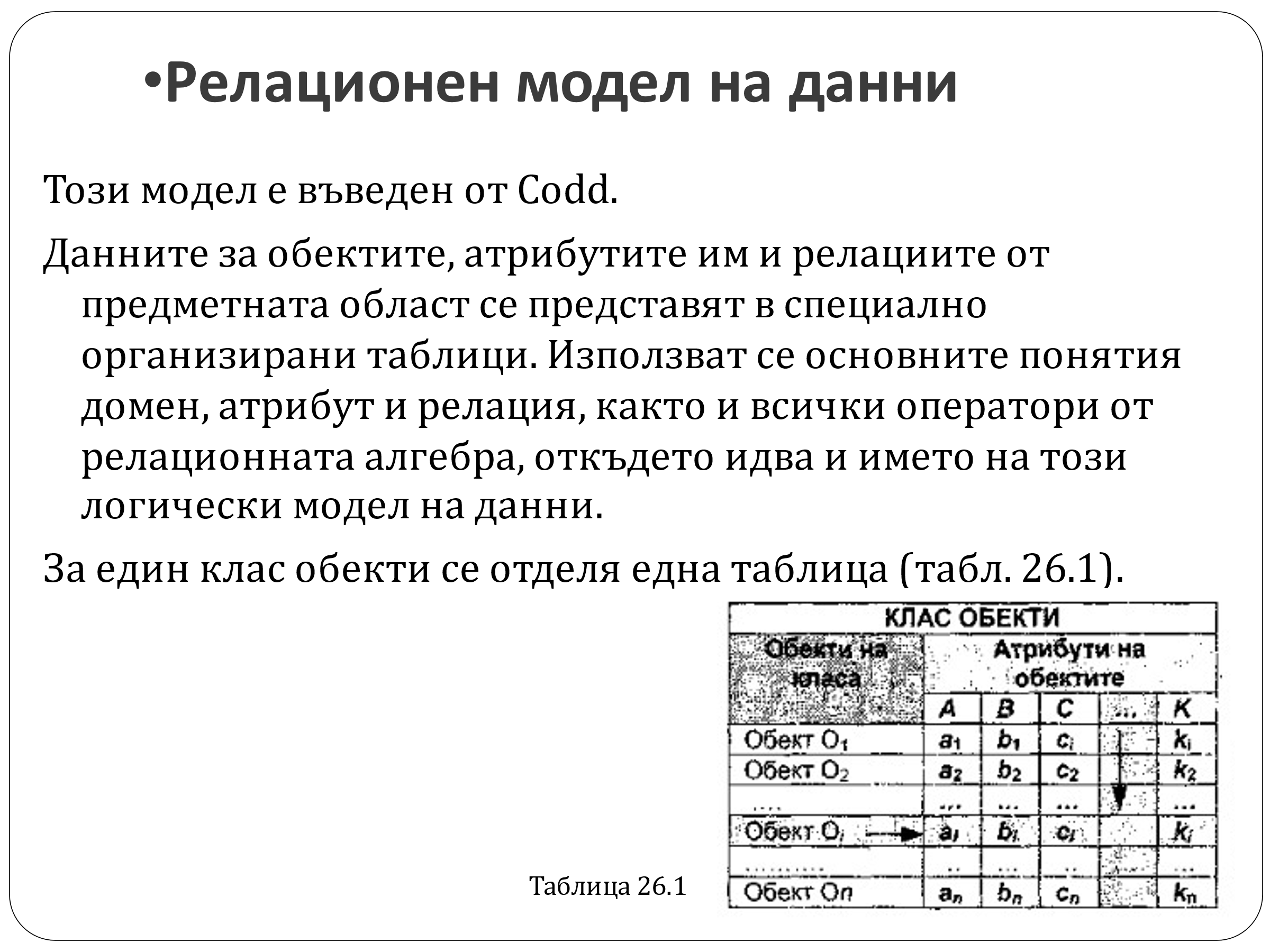
| **Елемент на SQL** | **Определение** | **Пример** |
| --- | --- | --- |
| идентификатор | Име, което използвате, за да идентифицирате обект на база данни, например име на колона. | [Имейл адрес] и фирма |
| оператор | Ключова дума, която представя действие или променя действие. | AS |
| константа | Стойност, която не се променя, например число или NULL. | 42 |
| израз | Комбинация на идентификатори, оператори, константи и функции, които дават като резултат единична стойност. | >= Продукти.[Единична цена] |

Моделиране на релационни бази от данни

Под моделиране на данни се разбира намиране на концептуално решение за организация на данните. Структурата на данни е детайлно описание на данните чрез списъци, масиви, файлове и тяхното съхраняване в компютър. Най-често се използват информационните структури дърво, мрежа и таблица, на които съответстват три известни модели на данни - йерархичен, мрежови и релационен. Базите данни, които използват тези модели се наричат съответно йерархични, мрежови и релационни.

Съществуващите връзки между обектите се описват графически като се използват елементи от теорията на графите. На върховете на графа отговаря определен тип запис, а на дъгата между два върха на графа отговаря тип релация. 







Заявки за извличане и промяна на данни

1. Поставете всеки израз в заявката в нов ред.
2. Поставете ключовите думи SQL в заявката с главни букви.
3. Използвайте CamelCase с главни букви в заявката и избягвайте подчертаването (Напишете ProductName, а не Product\_Name).

* SELECT - extracts data from a database
* UPDATE - updates data in a database
* DELETE - deletes data from a database
* INSERT INTO - inserts new data into a database
* CREATE DATABASE - creates a new database
* ALTER DATABASE - modifies a database
* CREATE TABLE - creates a new table
* ALTER TABLE - modifies a table
* DROP TABLE - deletes a table
* CREATE INDEX - creates an index (search key)
* DROP INDEX - deletes an index

Сложни заявки за извличане на данни

1. **To fetch ALTERNATE records from a table. (EVEN NUMBERED)**select \* from emp where rowid in (select decode(mod(rownum,2),0,rowid, null) from emp);
2. **To select ALTERNATE records from a table. (ODD NUMBERED)**select \* from emp where rowid in (select decode(mod(rownum,2),0,null ,rowid) from emp);
3. **Find the 3rd MAX salary in the emp table.**select distinct sal from emp e1 where 3 = (select count(distinct sal) from emp e2 where e1.sal <= e2.sal);
4. **Find the 3rd MIN salary in the emp table.**select distinct sal from emp e1 where 3 = (select count(distinct sal) from emp e2where e1.sal >= e2.sal);
5. **Select FIRST n records from a table.**select \* from emp where rownum <= &n;
6. **Select LAST n records from a table**select \* from emp minus select \* from emp where rownum <= (select count(\*) - &n from emp);
7. **List dept no., Dept name for all the departments in which there are no employees in the department.**select \* from dept where deptno not in (select deptno from emp);    
   alternate solution:  select \* from dept a where not exists (select \* from emp b where a.deptno = b.deptno);  
   altertnate solution:  select empno,ename,b.deptno,dname from emp a, dept b where a.deptno(+) = b.deptno and empno is null;
8. **How to get 3 Max salaries ?**select distinct sal from emp a where 3 >= (select count(distinct sal) from emp b where a.sal <= b.sal) order by a.sal desc;
9. **How to get 3 Min salaries ?**select distinct sal from emp a  where 3 >= (select count(distinct sal) from emp b  where a.sal >= b.sal);
10. **How to get nth max salaries ?**  
    select distinct hiredate from emp a where &n =  (select count(distinct sal) from emp b where a.sal >= b.sal);
11. **Select DISTINCT RECORDS from emp table.**select \* from emp a where  rowid = (select max(rowid) from emp b where  a.empno=b.empno);
12. **How to delete duplicate rows in a table?**delete from emp a where rowid != (select max(rowid) from emp b where  a.empno=b.empno);
13. **Count of number of employees in  department  wise.**select count(EMPNO), b.deptno, dname from emp a, dept b  where a.deptno(+)=b.deptno  group by b.deptno,dname;

Съединения на таблици

(SQL JOIN)

Общ преглед

Сложно присъединяване в SQL също се нарича външно съединение. Това не е непременно по-сложно от вътрешно съединение. Той се нарича "сложен", просто защото SQL провежда вътрешно присъединяване в допълнение към събирането на малко повече информация от една или повече таблици. Каква е тази допълнителна част от информацията? Ще открием това, като разгледаме разликата между външните и вътрешните съединения.

**Разлика между Вътрешно присъединяване (Inner JOIN) и Външно присъединяване (Outer JOIN)**

Вътрешно присъединяване(Outer JOIN)

Както си спомняте, вътрешното присъединяване ще върне само редовете от базата данни, които съответстват на заявката. Например, представете си, че имаме следните таблици:

TEACHERS TABLE STUDENTS TABLE

id student\_id teacher\_id

--------------- ------------------------

1 1 NULL

2 2 1

3 3 NULL

**Note**: If you would like to follow along in the terminal, first run sqlite3 to open the Command Line Shell for SQLite3. Once opened, run the following queries to create the two tables:

**CREATE** **TABLE** Teachers(id);

**CREATE** **TABLE** Students(student\_id, teacher\_id);

**INSERT** **INTO** Teachers **VALUES** (1);

**INSERT** **INTO** Teachers **VALUES** (2);

**INSERT** **INTO** Teachers **VALUES** (3);

**INSERT** **INTO** Students **VALUES** (1, **NULL**);

**INSERT** **INTO** Students **VALUES** (2, 1);

**INSERT** **INTO** Students **VALUES** (3, **NULL**);

First, let's look at an inner join:

**SELECT** **\***

**FROM** Teachers

**INNER** **JOIN** Students

**ON** Teachers.id **=** Students.teacher\_id;

This query returns only the teacher with the id = 1 because student 2 is in the first teacher's class.

id | student\_id | teacher\_id

--------------------------------

1 | 2 | 1

Забележка: Тъй като обединяваме таблици, стартирането на тази примерна SQL команда ще върне резултат както с id, така и с учител\_идентификатор, въпреки че те са еднакви.

Външно присъединяване

Външните съединения, от друга страна, ще върнат всички съответстващи редове И всички допълнителни редове от посочената таблица. Коя таблица / допълнителни редове се определят от вида на външното съединение. Има три вида външни съединения: ляво външно присъединяване, дясно външно присъединяване и пълно външно присъединяване.

ЗАБЕЛЕЖКА: SQLite, системата за управление на база данни, която използвахме за изследване на SQL, не изпълнява пълния SQL стандарт. По-конкретно, SQLite не прилага ПРАВО ВЪНШНО ПРИСЪЕДИНЯВАНЕ или ПЪЛНО ВЪНШНО ПРИСЪЕДИНЯВАНЕ. Концепциите, залегнали в основата на тези обединения, все още са важни за разбиране (и други бази данни, като PostgreSQL, ги прилагат), така че ще трябва да знаете за тези JOIN, дори ако няма да ги използвате веднага.

Ляво външно присъединяване

Това е най-често срещаното външно съединение и това, което ще използвате най-често. Това връща нормалния резултат от вътрешно присъединяване и също така връща всички редове от най-лявата (т.е. първата спомената) таблица.

Също така използваме ключовата дума as, която ни позволява да посочим как се показват нашите върнати данни. Ще видите как името as помага да направите изхода по-лесен за четене.

**SELECT**

Teachers.id **as** teacher\_id,

Students.student\_id

**FROM** Teachers

**LEFT** **OUTER** **JOIN** Students

**ON** Teachers.id **=** Students.teacher\_id;

teacher\_id | student\_id

--------------------------

1 | 2

2 | NULL

3 | NULL

Notice that every row from the teacher's table is returned whether there is a corresponding student or not.

**Right Outer Join**

As you might imagine, this is the same as the Left Outer Join with the minor difference being that it ***returns all of the rows from the right-most (i.e. last-mentioned) table***. Sticking with our example:

**SELECT**

Teachers.id **as** teacher\_id,

Students.id **as** student\_id

**FROM** Teachers

**RIGHT** **OUTER** **JOIN** Students

**ON** Teachers.id **=** Students.teacher\_id;

teacher\_id | student\_id

-----------------------------

NULL | 1

1 | 2

NULL | 3

**Full Outer Join**

The full ***returns all of the rows from all the tables***.

**SELECT**

Teachers.id **as** teacher\_id,

Students.id **as** student\_id

**FROM** Teachers

**FULL** **OUTER** **JOIN** Students

**ON** Teachers.id **=** Students.teacher\_id;

teacher\_id | student\_id

-----------------------------

NULL | 1

1 | 2

NULL | 3

2 | NULL

3 | NULL

Агрегатни функции

Агрегатните функции ни позволят да правим обобщения по множество от стойности, да търсим мин. или макс. елемент в такова множество и др.

А те са:

* SUM();
* MAX();
* MIN();
* AVG();
* COUNT();

В скобите обикновено присъства име на колона, за която желаем да бъде приложена агрегатната функция.

Нека малко да разширим базата данни. Ще създадем две нови таблици. Едната ще отразява плащанията на всеки ученик. Ще съдържа информация за това кой плаща, за коя група, за кой месец, за коя година, сума и дата, на която е извършено плащането. Дргата таблица ще бъде за месечни заплати на треньорите. Тя ще съдържа информация за това кой колко е получил, за кой месец на коя година и дата на получаване.

USE school\_sport\_clubs;

DROP TABLE IF EXISTS taxesPayments;

CREATE TABLE taxesPayments(

id INT AUTO\_INCREMENT PRIMARY KEY,

student\_id INT NOT NULL,

group\_id INT NOT NULL,

paymentAmount double NOT NULL,

month TINYINT,

year YEAR,

dateOfPayment datetime not null,

CONSTRAINT FOREIGN KEY (student\_id) references students(id),

CONSTRAINT FOREIGN KEY (group\_id) references sportgroups(id)

);

Групиране на данни

(GROUP BY)

Както виждате, агрегатните функции работят с цялото множество от стойности на колоните, които са им подадени и съответно изчисялват и връщат един резултат. Как да накараме, обаче, една агрегатна функция да изчислява не цялото множество, а определени групички от него. Например искаме да изведем средната стойност на платените такси по групи – тоест в резултата ще искаме група – средна стойност на такса. Това няма как да стане без да ползваме групиране. Ще трябва да укажем на агрегатната функция някакъв критерий, на който тя да раздели множеството от стойности в колоната на отделни подмножества, да се изпълни над всяко едно от тях по отделно и за всяко да върне резултат. За целта използваме клаузата GROUP BY. Обикновено групирането се използва в комбинация с агрегатни функции. Целта е агрегатната функция да се приложи върху множество по-малки резултатни таблици или групи, които накрая ще се обединят в една обща.

Нека да преработим последната заявка и да видим за всяка група от двете въведени какви са средните стойности на таксите.

SELECT group\_id as GroupId, AVG(paymentAmount) as AvgOfAllPaymentPerGroup

FROM taxespayments

GROUP BY group\_id;

Скаларни функции

SQL скаларна функции връщат една стойност, на базата на стойността на входа.

Полезни скаларни функции:

**Съвет:** Обобщените функции и функциите на скаларни ще бъдат обяснени подробно в следващите глави.

SQL, **S** tructured **Q** uery **L** anguage, е език за програмиране, предназначен да управлява данните, съхранявани в релационни бази данни. SQL оперира чрез прости, декларативни изявления. Това поддържа данни точното и сигурно, и това помага да се поддържа целостта на бази данни, независимо от размера.

Ето един придатък на често използваните команди.

ALTER TABLE ви позволява да добавите графи таблица в база данни.

ANDе оператор, който комбинира две условия. И двете условия трябва да са верни на реда, за да бъдат включени в комплекта резултат.

ASе ключова дума в SQL, която ви позволява да преименувате колона или маса с помощта на *псевдоним* .

AVG() е обобщаваща функция, която връща средната стойност за цифрова колона.

На BETWEENоператора се използва за филтриране на набор резултат в рамките на определен обхват. Стойностите могат да бъдат числа, текст или дати.

CASEотчети са използвани за създаване на различни изходи (обикновено в SELECTотчета). Това е начин за справяне ако-тогава логиката на SQL е.

COUNT()е функция, която е на името на колона като аргумент и брои броя на редовете, където колоната не е NULL.

CREATE TABLEсъздава нова таблица в базата данни. Тя ви позволява да укажете името на таблицата и наименованието на всяка колона в таблицата.

DELETE отчети са използвани за премахване на редове от таблица.

GROUP BYе клауза в SQL, която се използва само с обобщаващи функции. Той се използва в сътрудничество с SELECTдекларация да се организира идентични данни в групи.

HAVINGе добавен в SQL, тъй като WHEREключовата дума не може да се използва с обобщаващи функции.

Вътрешно се присъединят ще комбинира редове от различни таблици, ако *присъединят условие* е вярно.

INSERT отчети са използвани за добавяне на нов ред, за да една маса.

IS NULLи IS NOT NULLса оператори използват с WHEREклауза за изпитване на празни стойности.

LIKEе специален оператор използва с WHEREклаузата за търсене на конкретен модел в колона.

LIMIT е клауза, която ви позволява да посочите максималния брой редове, на снимачната площадка резултат ще има.

MAX() е функция, която е на името на колона като аргумент и връща най-голямата стойност в тази графа.

MIN() е функция, която е на името на колона като аргумент и връща най-малката стойност в тази графа.

OR е оператор, който филтрира резултатния набор да се включат само редове, където от двете условия не е вярно.

ORDER BY е клауза, която показва, която искате да сортирате резултатния набор от конкретен колона или по азбучен ред или по цифров път.

Външно се присъединят ще комбинира редове от различни таблици, дори ако присъединят условие не е изпълнено. Всеки ред в *лявата* таблица се връща в резултатния набор, и ако присъединят условие не е изпълнено, тогава NULLстойности се използват за попълване на колоните от *дясно* масата.

ROUND()е функция, която носи името на колона и цяло число като аргумент. Тя кръга на стойностите в колоната с броя на знака след десетичната запетая, определени от цяло число.

SELECT отчети са използвани за извличане на данни от база данни. Всяка заявка ще започне с SELECT.

SELECT DISTINCT посочва, че изявлението ще бъде въпрос, който връща уникални стойности в определения колона (а).

SUM() е функция, която е на името на колона като аргумент и връща сумата от всички стойности в тази графа.

WHERE е клауза, която показва, която искате да филтрирате резултатния набор да се включват само редове, където след *условието* е вярно.

WITH клауза ви позволява да съхранявате резултата от заявка в временна таблица, използвайки псевдоним. Можете също така да определите множество временни таблици, използващи запетая и с едно копие на WITHключовата дума.

На WITH клаузата е известен също като обща маса експресия (КТР) и подзаявка факторинг.

Tранзакции

Транзакциите имат за цел да осигурят цялост и съгласуваност на данните. Основно тяхно свойство е така наречената атомарност (Atomicity), която определя, че дадена транзакция или трябва да завърши успешно всички промени, или трябва да върне данните до първоначално състояние.

* START TRANSACTION (или BEGIN WORK, в зависимост от SQL диалекта) – маркира началото на транзакция. Допустимо е неявното стартиране на транзакция. (До SQL92 в стандарта не фигурира команда за явно стартиране на транзакция)
* COMMIT – всички промени се записват.
* ROLLBACK – отказва всички промени след последния COMMIT или ROLLBACK, така че данните са възстановени в състоянието в което са били преди началото на транзакцията.

COMMIT и ROLLBACK комуникират с контрола на транзакции и „заключване“ (transaction control and locking). И двете команди завършват транзакция и отключват данните. При липса на START TRANSACTION действието зависи от разработчика на продукта.

Пример:

START TRANSACTION;

UPDATE inventory SET quantity = quantity – 3 WHERE item = 'pants';

COMMIT;

Съхранени процедури

Съхранените процедури представляват съвкупност от SQL заявки, съхранени в специален каталог в базата данни и много наподобяват методите(функциите) в езиците за програмиране. Често се налага да се пазят заявки, които ще се ползват често, но с различни параметри (динамични), като например отчети, справки, проверки и тн. Друг типичен случай е, когато се налага да правим зареждане на данни от едно място на друго, или когато трябва да изпълним последователност от операции за приключване на отчетен период, например. Тогава удобно решение е да имаме процедури в базата, съдържащи необходимото множество от заявки, които да бъдат извиквани от някаква програма или от самия сървър в определено време. Писането на съхранени процедури може да увеличи производителността на едно приложение. Веднъж написани, процедурите в MySQL се компилират всеки път по време на извикването им и компилираната им версия се пази в кеша на сървъра за текущата отделна връзка (сесия). Съществено предимство на работата с процедури е, че те намаляват значително трафика на данни между приложението и сървъра на СУБД. Вместо да се изпращат множество заявки към сървъра, той да връща отговори на всяка една от тях, програмата да проверява или валидира тези отговори и пак да върне заявка към сървъра, то всичко това би могло да се случи само с извикването на името на процедурата и предаването на определени параметри, с които тя да си свърши работата.

Синтаксис:

DELIMITER <symbol> -> Сменя знака за край на заявка.

CREATE PROCEDURE <name of procedure>( ) -> декларира процедура с име<име>

BEGIN

<QUERIES> -> заявки, които ще се изпълняват при викане на процедурата

…

END

<Symbol> -> край на процедурата

DELIMITER ; -> Сменя символа за край на завка да бъде “;“.

Тригери

Тригерът представлява множество от SQLзаявки, съхранени под определено име в каталог в базата данни, но за разлика от процедурата, този обект е асоцииран с определена реална таблица от базата. Под реална се разбира – перманентна таблица, а не временна.  Тригерите спомагат за запазването на интегритет на базата данни. Активират се при настъпване на определено събитие върху таблицата, с която са асоциирани, като това събитие е свързано с промяна на данните в нея. Тоест, тригерът се „активира“ (fire), когато настъпи събитие от тип  INSERT, UPDATE, DELETE- това са трите операции, при които се променят данните в една таблица. Тригерите много приличат на съхранени процедури, но за разлика от тях не могат да бъдат извиквани собственоръчно или пък да се извикват един друг. От това, че те се активират при опит за промяна на данни в таблиците, тези обекти в БД намират широко приложение при валидация на данни, проверка на определени бизнес правила на ниво БД, извършване на елементарни преобразувателни операции върху данните, преди или след като бъдат вкарани, както и за извършване на лог върху всички промени по дадена таблица. Като недостатък на използването на тригери можем да посочим, че изпълнението им остава скрито за приложението, което ползва базата. При таблици, при които често се модифицират данните, би довело до overhead на базата.

За да си изясните как работят тригерите, трябва да се запознаете с някои понятия: *trigger time, trigger event, trigger action*. Първото показва кога да се извика тригерът – преди/след определена операция(insert, update, delete), наречена trigger event. Trigger action е всъщност действието, което ще извършва тригерът. Предстои да разгледаме и възможните стойности на тези три характеристики на тригерите:

* **trigger time –**Може да бъде BEFORE или AFTER.
* **trigger event –**Може да бъде INSERT, UPDATE, DELETE.
* **trigger action–**Може да бъде съвкупност от SQL заявки.

**Жизнен цикъл:**

Тригерът се „активира“ всеки път при настъпване на неговия trigger event, по време на неговия trigger timе. Проверява се дадено условие и ако то е истина, се изпълнява зададения trigger action.

**Синтаксис:**

**CREATE TRIGGER** <trigger\_name> <trigger time> <trigger event> **ON** <table>  
**FOR** **EACH** **ROW** <trigger action>;

Изпитна тема № 10:

Разработка на софтуер План-тезис:

Трислоен модел и MVC. Концепция за тестване и писане на компонентни тестове. Концепция за дебъгване, откриване и отстраняване на грешки. Концепция за рефакториране и правене на „инкрементални промени“. Примерна приложна задача

Трислойна архитектура

В софтуерното инженерство многослойната архитектура (наричана още N-слойна архитектура) е архитектура от тип клиент-сървър, в която интерфейсът, обработката на приложения и съхранението и обработката на данни са логически разделени на отделни модули. Най-разпространената форма на многослойна архитектура е трислойната архитектура.

Многослойната архитектура осигурява модел, по който разработчиците могат да създават гъвкави приложения, които могат да бъдат използвани многократно. При разделянето на едно приложение на слоеве, разработчиците имат възможността да добавят или променят отделен слой, вместо да преработват цялото приложение.

Употреба в уеб разработката

В областта на уеб разработката трислойната архитектура често се използва в уеб сайтове, най-често т.нар. „електронни магазини“, които се състоят от 3 слоя:

Front-end слой (Клиентски слой) – този слой, е съдържанието, което се възпроизвежда от уеб браузърът и достига до крайния потребител. То може да бъде генерирано статично или динамично.

Среден слой – той реализира логиката на самото уеб приложение. Това се осъществява със сървър, който обработва и генерира динамично съдържание. Този сървър може да е с платформа един от следните езици – Ruby on Rails, Java EE, ASP.NET, PHP, ColdFusion, Perl, Node.js.

Back-end слой – той представлява база-данни, която се състои, както от самата информация в таблици, така и от система за управлението и (RDBMS).

Обширно разгледана трислойната архитектура

Всички съвременни бизнес приложения имат достъп до различен вид данни като част от тяхната основна функционалност. След като релационните бази данни добиха широка популярност, IT индустрията се преориентира от еднослоен модел (mainframe) към модела клиент-сървър. При него са налице клиент, който включва презентационна логика и голямата част от бизнес логиката на приложението, както и сървър включващ в себе си хранилище за данни (data storage) и част от бизнес логиката под формата на готови и съхранени заявки към базата данни. От началото на 90-те години клиент-сървър моделът започна да излиза от употреба поради високите разходи за поддръжка и невъзможността за разпределение на отговорности при възникване на проблеми. Моделът, който постепенно го измества е трислойната архитектура показана на картинката от дясно.

Трислойната архитектура обикновено се състои от презентационен слой (потребителски интерфейс) на най-високо ниво в приложението. Той служи за прякото взаимодействие с потребителя и изпращането на заявки към бизнес слоя. Не трябва да има директна връзка между този слой и слоят за бази данни. Бизнес слоят служи за обработка на данните и работните процеси. Той комуникира както с презентационния слой, така и с базите данни. Слоят за данни комуникира само с бизнес логиката и служи за съхранение данни и тяхното използване. Често отделните модули се разработват и на различни платформи. Разработена е от Джон Дж. Донован, в неговата компания OEC (Open Environment Corporation), основана в Кеймбридж, Масачузетс.

Освен стандартните предимства на модулния софтуер, трислойната архитектура е проектирана да позволява да бъде променен или заменен независимо, който и да е от трите слоя, за да отговаря на промени в изискванията или технологиите. Например, промяна на операционната система на презентационния слой би засегнала само кода за потребителският интерфейс.

В повечето случай, потребителският интерфейс работи на десктоп система и използва стандартен графичен интерфейс, функционалната логика за обработка на данните, която може да се състои от един или повече отделни модула, работещи на десктоп система или сървърно приложение, и РСУБД (релационните системи за управление на бази данни) работещи на сървър база данни или мейнфрейм, който съдържа логиката за съхранението и обработката на базите данни. Междинният слой може самият да бъде многослоен. В такъв случай цялостната архитектура се нарича N-слойна архитектура.

Трислойната архитектура се състои от следните три слоя:

Презентационен слой

Презентационният слой е на най-високо ниво в приложението и потребителят има директен достъп до него. Освен, че служи комуникира с останалите слоеве, презентационният слой предоставя различни видове информация на потребителя.

Слой за бизнес логика (междинен слой, слой за обработка на данни)

Този слой е изтеглен от презентационния слой, и като отделен такъв, контролира функционалността на приложението като извършва различни процеси по обработката на данните.

Слой за данните

Този слой се състои от сървър база данни. Тук информацията се съхранява и чете. В слоя за бази данни информацията се съхранява независима от бизнес логиката или сървърът за приложения. Когато данните се съхраняват в отделен слой се увеличава мащабируемостта и се подобрява производителността.

Модел-изглед-контролер

(MVC)

Модел-Изглед-Контролер (Model-View-Controller или MVC) е архитектурен шаблон за дизайн в програмирането, основан на разделянето на бизнес логиката от графичния интерфейс и данните в дадено приложение. За пръв път този шаблон за дизайн е използван в програмния език Smalltalk.

Описание и компоненти на MVC

Модел

Ядрото на приложението, предопределено от областта, за която се разработва; обикновено това са данните от реалния свят, които се моделират и над които се работи – въвеждане, промяна, показване и т.н. Трябва да се прави разлика между реалния обкръжаващ свят и въображаемият абстрактен моделен свят, който е продукт на разума, който се възприема като твърдения, формули, математическа символика, схеми и други помощни средства. Например в банково приложение това са класовете, описващи клиентите, техните сметки, транзакциите, които са осъществили и т.н., както и класовете за извършване на операции над тези обекти (engines) – например клас Transfer с методи като createInterBankTransfer(), createInnerBankTransfer(), getCash() и т.н.

Изглед

(View)

Тази част от изходния код на приложението, отговорна за показването на данните от модела. Например изгледът може да се състои от PHP шаблонни класове, JSP страници, ASP страници, JFrame наследници в Swing приложение. Зависи от това какъв графичен интерфейс се прави и каква платформа се използва;

Контролер

Тази част от сорс кода (клас или библиотека), която взима данните от модела или извиква допълнителни методи върху модела, предварително обработва данните, и чак след това ги дава на изгледа. Например може да бъде създаден един малък обект, в който да бъдат сложени данните за транзакцията – като в контролера бъдат взети данните за транзакцията от модела, бъдат преведени датите от UNIX формат в четим от потребителя формат, бъде преобразувана валутата от долари в евро например, бъде закръглено до втория знак вместо да се виждат данните както са в модела (и в базата) до 10-ия. Също така когато се прави уеб графичен интерфейс това би довело до много лесна модификация на HTML кода дори от човек, който не е програмист – той ще гледа на шаблона просто като на обикновена HTML страница.

Предимства

Моделът е независим от контролера и изгледа.

Моделът може да бъде планиран и осъществен независимо от другите части на системата.

За един и същи модел могат да бъдат осъществени различни изгледи (интерфейси) – например уеб интерфейс и нативен интерфейс към Facebook.

Контролерът и изгледът могат да бъдат променени, без да се налага промяна в модела.

Недостатъци

Софтуерната система достига ново ниво на сложност, което я прави по-трудно разбираема за софтуерните разработчици, работещи по нея.

Концепция за тестване и писане на компонентни тестове

Концепция за тестване

Компонентното тестване (на английски: unit testing) е процес в програмирането, чрез който се тестват отделни единици (компоненти) от сорс код – един или повече програмни модула и техните контролни данни, процедури за използване, оперативни процедури с цел да се установи дали работят правилно. Под компонент се разбира най-малката част на едно програмно приложение, която може да бъде самостоятелно тествана. В т.нар. процедурно програмиране компонент би могъл да бъде целия програмен модул, но в по-честите случаи е отделна функция или процедура. В случая на обектно ориентирано програмиране компонент е най-често цял клас, но в някои случаи може и да е отделен метод. Компонентните тестове се създават от програмист или в някои случаи от white box тестери по време на процеса на разработка, за да се гарантира правилното поведение на софтуера и дали той покрива всички първоначално поставени изисквания.

Тестване и писане на компонентни тестове

Предимства

Целта на компонентното тестване е да се изолира всяка част от програмата и да се покаже, че отделните части работят правилно. Компонентният тест предоставя строго определени правила, които частите от кода трябва да удовлетворяват. От това произтичат серия предимства:

Ранно откриване на проблеми

Чрез компонентните тестове могат да се откриват проблеми в цикъла на разработка на софтуер на ранен етап.

В test-driven разработката, която често се използва в т.нар. екстремно програмиране и Scrum, компонентните тестове се създават още преди кодът да бъде завършен. Той става такъв едва след преминаването на тестовете. Едни и същи компонентни тестове се повтарят когато базата на кода се разширява или когато се правят промени по кода. Ако някой тест се провали, това се счита за програмна грешка при промяната на кода или в грешка в самия тест. По този начин тестовете позволяват лесно да се открие къде точно се появява грешката още в процеса на разработка.

Улеснение на промените

Компонентните тестове позволяват на програмиста да рефакторира кода в по-късен етап и да се увери, че промененият модул все още работи коректно. Процедурата се състои в това да се напишат тестови случаи за всички методи, така че ако някоя от промените предизвика грешка, тя да бъде идентифицирана бързо.

Опростяване на интеграцията

Компонентното тестване може да намали несигурността в отделните компоненти и може да се използва за bottom-up подход на разработка и тестване. Този подход се изразява в тестване първо на отделни части от програмата, а след това в тестване на всички части заедно, което значително улеснява интеграцията на софтуера.

Основната цел на компонентното тестване е да вземе най-малкото самостоятелно парче от код в софтуерното приложение, това парче да се изолира от останалата част от кода и да се определи дали тази част работи точно по начина, по който се очаква от нея да работи. Всяка единица се тества поотделно преди интеграцията на всички единици в модул, за да се тестват интерфейсите между модулите.

Документация

Компонентното тестване представлява „живата“ софтуерна документация на системата. Разработчиците, които искат да разберат какви са функционалностите на някой компонент и как могат да ги използват, преглеждат компонентните тестове, за да разберат приложно-програмния интерфейс.

Характеристиките на компонентното тестване могат да индикират за подходящо/неподходящо използване на компонент, също така и за неговото неправилно поведение. Компонентните тестове документират всички тези критични характеристики, въпреки че доста компании, занимаващи се със софтуерна разработка, не разчитат само на тази документация. В процеса на имплементиране на програмата е доста по-вероятно обикновената документация да остарее и да не е вече актуална.

Дизайн

При разработката на софтуер, когато се използва test-driven подход, комбинацията между писане на тестове за определяне на интерфейса плюс рефакторирането, извършено след преминаването на теста могат да заемат мястото на формален дизайн. Всеки компонентен тест може да се разглежда като елемент на дизайна, определящ класове, методи и поведение. Това може да се илюстрира от следния пример на Java:

Имаме тестов клас, който определя броя елементи в имплементацията. Първо трябва да има интерфейс на име Adder и имплементиращ клас с конструктор нулев аргумент на име AdderImpl. След това Adder трябва да има метод, който се казва add с два целочислени параметъра (int), който връща също целочислена стойност. Също така е определено поведението на метода за малък обхват от стойности.

Концепция за дебъгване, откриване и отстраняване на грешки

Отделяне на интерфейса от имплементацията

Тъй като някои класове могат да съдържат референции към други класове, тестването на един клас може да се разпростре върху тестване на друг клас. Често срещан пример за това са класове, които са зависими от база данни: за да тества класа, разработчикът обикновено пише код, който взаимодейства с базата данни. Това е грешка, тъй като компонентното тестване на даден клас не трябва да излиза от границите на самия този клас и в никакъв случай не трябва да се прехвърля върху процесите/мрежите, защото това може да доведе до недопустими проблеми в компонентните тестове. Излизането на теста изън границите на даден клас, води до използването на т.нар. интеграционно тестване и ако то се провали, много трудно ще открием кой компонент в причинил проблема.

Вместо това, разработчикът трябва да създаде абстрактен интерфейс, около заявките на базата данни, и след това да имплементира този интерфейс с техния собствен mock-обект. След отделянето на тази необходима част от кода, независимите компоненти могат да бъдат детайлно тествани. Получените резултати са с по-добро качество и са по-лесни за поддръжка.

Параметризирано компонентно тестване

Параметризирани компонентни тестве са тестове, които приемат параметри. Противно на традиционните тестове, които в повечето случаи са затворени методи, параметризираното компонентно тестване (ПКТ) използва списък от параметри. ПКТ се поддържа от TestNG, JUnit и от още много .NET тестови платформи. Параметри, подходящи за компонентно тестване, могат да бъдат подадени ръчно или в някои случаи могат да бъдат автоматично генерирани от тестовата платформа. Съществуват различни пособия за тестване, които генерират тестови вход за параметризираното компонентно тестване, например QuickCheck.

Ограничения на компонентното тестване

Тестването не може да установи всяка грешка в дадена програма, тъй като не може да оцени всяка пътека на изпълнение на програмата, освен в тривиални програми. Същото нещо важи и за компонентното тестване, което в допълнение на това по дефинция се занимава само с функционалността на отделните компоненти, следователно не може да улови грешки в интеграцията или системни грешки (като например функции, които се изпълняват на различни единици или перформънс като цяло). Компонентното тестване трябва да се прилага заедно с други софтуерни тестове, тъй като показва само наличието или отсъствието на конкретни грешки, но не може да докаже абсолютното неналичие на грешки. За да може да се гарантира коректно поведение за всяка пътека на изпълнение на програмата и да се осигури пълното неналичие на грешки, се използват други техники, а именно прилагане на формални методи за доказателство, че софтуерният компонент не може да има неочаквано поведение.

Софтуерното тестване също може да се разглежда като комбинаторен проблем. Например, всяко булево твърдение, което може да има стойности „Истина“ или „Неистина“ изисква поне два теста за всеки един от възможните изходи. Като резултат за всеки ред от написания код, програмистите често се нуждаят от 3 – 5 реда код за тестване.[3]. Това очевидно отнема време и прекалено много усилия. Освен това има много проблеми, които изобщо не могат да бъдат тествани по лесен начин – например тези, които са недетерминистчни или включват няколко неща. Може да се случи така, че кодът на компонентния тест да е също така неправилно написан и да съдържа грешки, както и самият тестван код.

Друго предизвикателство, свързано с писането на компонентни тестове, е трудността да се съставят реалистични и полезни тестове. Необходимо е да се създадат релевантни начални условия, така че частта от приложението, която се тества, да има поведение на завършена пълна система. Ако тези начални условия не са създадени правилно, тестът няма да изпитва кода по релевантен начин, което намалява стойността на коректност на резултатите от компонентния тест.

За да се постигнат определени резултати от компонентното тестване, е нужна строга дисциплина по време на процеса на разработка. Важно е внимателно да се пазят записи не само на приложените тестове, но също така и на приложените промени по сорс кода на конкретната тествана и на всички останали единици в софтуера. Особено важно е използването на version control системи. Ако по-късна версия на даден компонент се провали на някой тест, който преди това е била преминала, version control системата може да предостави списък от промените в сорс кода, които са били направени между двата теста.

Важно нещо при компонентните тестове е да се имплементира устойчив тестови процес, за да се осигури ежедневен преглед и незабавно изпращане на грешките.[5] Ако такъв тестов процес липсва и не е интегриран в работния процес на екипа, разработваното приложение ще се развива извън тестовата рамка и по този начин ще се увеличи наличието на грешки и съответно ще се намали ефективността на тестовата рамка.

Вградените софтуерни системи за компонентно тестване представляват интересно предизвикателство, тъй като софтуерът е бил разработен на различна платформа от тази, на която евентуално ще работи след това, няма как да се пусне тестова програма в реалната среда на изпълнение.

Приложения

Екстремно програмиране

Компонентното тестване играе ключова роля в т.нар. екстремно програмиране, което разчита на автоматзирана тестова рамка, която може да бъде или чужда (напр. xUnit), или създадена от екипа разработчици.

Екстремното програмиране използва компонентните тестове за test-driven разработка. Разработчикът пише компонентен тест, който показва или някакво изискване към софтуера или дефект. Този тест ще се провали, защото или изискването към софтуера още не е имплементирано или защото той целенасочено показва дефект в съществуващия код. След това разработчикът пише най-простият възможен код, за да направи така, че тестът да бъде преминат заедно с другите тестове.

По-голямата част от кода в системата се подлага на компонентно тестване, но не и всички пътеки в този код. В екстремното програмиране се прилага стратегията „тествай всичко, което би могло да се счупи“, за разлика от традиционното „тествай всяка пътека на изпълнение“. Това помага на разаботчиците да разработват оп-малко тестове от обикновено, когато се прилагат класически методи.

От решаващо значение е тестовият код да се разглежда като първокачествен елемент на проекта и да се изпълни със същото качество, както и самия код на имплеметацията. Кодът на компонентните тестове се създава в едно и също репозитори с кода, който ще бъде тестван. По този начин се осигурява по-прост и сигурен начин за разработка и рефакториране, опростена интегрция на кода, коректна документация. Компонентните тестове могат също да се прилагат като форма на регресионни тестове.

Техники

Компонентните тестове в общия случай са автоматизирани, но могат да се изплняват и ръчно. IEEE не дава приоритет на единия вид пред другия. Подходът на ръчното тетване може да включва инструктаж за прилагането му стъпка по стъпка. Целта на компонентното тестване е да се изолира някой компонент (единица) от програмата и да се установи нейната правилност. Автоматизираното тестване е ефикасно за постигането на тази цел и има редица предимства. По отношение на ръчното тестване, ако то не е планирано внимателно, може да се изпълни като интеграционен тест (на повече от един компонент) и по този начин да не доведе до постигането на целите на компонентното тестване.

За да се приложи напълно ефектът на изолирането при използването на автоматизирани компонентни тестове, компонентът, който се тества, се изпълнява в някаква рамка (unit test framework) извън неговата естествена среда. С други думи, компонентът се изпълнява извън програмата, от която той е част и където по начало е бил създаден и извикван. Прилагането на такъв подход на изолирано тестване разкрива наличието на излишни зависимости между тествания код и други компоненти в програмата. По този начин тези зависимости могат а бъдат премахнати.

При използването на автоматизирана тестова рамка, се разработват тестови критерии, за да се установи правилността на компонента. По време на изпълнението на теста, тестовата рамка пуска тестове, които се провалят по всички критерии и понякога автоматичо ги обозначава и докладва. В зависимост от изискванията за грешка, тестовата рамка може да прекрати последващото тестване.

Рамки за компонентно тестване (Unit testing frameworks)

Виж: Списък с рамки за компонентно тестване (List of unit testing frameworks)

Рамките за компонентно тестване обикновено са външни продукти, които не се разпространяват като част от компилаторния комплект. Те помагат за опростяването на процеса на компонентно тестване и са разработени за най-различни видове програмни езици. Някои видове тестови рамки включват възможности за приложение с отворен код, например различни code-driven тестови рамки, познати под общото название xUnit и комерсиални продукти като TBrun, JustMock, Isolator.NET, Isolator++, Parasoft Test (C/C++test, Jtest, dotTEST), Testwell CTA++ and VectorCAST/C++. Като цяло е взможно да се приложи компонентен тест бе използването на тестова рамка чрез написвне на клиент код, който да се изпълнява върху компонентите, правейки тестове и използвайки твърдения, улавяне на изключения или прилагайки други механизми за контрол и докладване на грешките. В някои тестови рамки липсват много от приложенията за компонентно тестване и те трябва да бъдат допълнително написани.

Рефакториране

Рефакториране представлява процеса на такава промяна в софтуерната система, при която външното поведение на кода не се променя, но вътрешната му структура се подобрява. Това е начин за систематично привеждане на кода в ред, при който шансовете за нови грешки са минимални. Всъщност, когато рефакторинг код, подобрите дизайна му, след като е написан.

„Подобряване на кода след написването му“ - необичайна фигура на речта. В нашето разбиране за разработка на софтуер първо създаваме дизайна на системата и след това пишем кода. Първо се създава добър дизайн, а след това се извършва кодиране. С течение на времето кодът се променя и целостта на системата, съответствието на нейната структура с първоначално създадения дизайн, постепенно се влошава. Кодът бавно се изплъзва от дизайн до хакване.

Рефакторингът е обратната практика. С негова помощ можете да вземете лош проект, дори хаотичен и да го прекроите в добре проектиран код. Всяка стъпка от този процес е проста до крайност. Полето се премества от един клас в друг, част от кода се взема от метода и се поставя в отделен метод, някои кодове се движат в йерархията в една или друга посока. Комбинираният ефект от такива малки промени обаче може драстично да подобри дизайна. Това е точно обратното на обичайната постепенна разбивка на дадена програма.

При извършване на рефакторинг се оказва, че съотношението на различните етапи на работа се променя. Проектирането се извършва непрекъснато по време на разработката и не се изпълнява предварително. При внедряването на системата става ясно как да се подобри нейният дизайн. Продължаващото взаимодействие води до създаването на програма, чието качество на проекта остава високо, докато развитието продължава.

1. След като установихте, че е необходимо да добавите нова функционалност към програмата, но програмният код не е структуриран по начин, удобен за добавяне на тази функционалност, първо рефакторирайте програмата, за да опростите извършването на необходимите промени и едва след това добавете функцията.
2. Преди да започнете рефакторинг, уверете се, че имате надежден тестов пакет. Тези тестове трябва да са самотествани.
3. Когато се приложи рефакторинг, програмата се променя на малки стъпки. Грешката се открива лесно.
4. Всеки може да напише код, разбираем за компютър, но само добрите програмисти пишат код, който хората разбират.

Най-важният урок, който този пример трябва да научи, е ритъмът на рефакторинг: тестване, малки промени, тестване, малки промени, тестване, малки промени. Именно този ритъм прави рефакторинга бърз и надежден.

„Инкрементални промени“

„Инкременталната“ или Постепенната промяна е един от ключовите принципи на Екстремно програмиране. Тази статия представя a методология и казус на постепенни промени с помощта на малко приложение, написано на UML и Java. Понятията за домейни играят ключова роля в тази методология. Ключови думи Допълнително програмиране, Екстремно програмиране, еволюция на софтуера, концепции за домейни, разпространение на промените 1.

ВЪВЕДЕНИЕ

Екстремното програмиране се основава, наред с няколко други неща, за бързата доставка на малки версии. Те предоставят ценна обратна връзка за посоката и напредък на проекта и позволяват навременни корекции на целите и графиците на проекта. По този метод, Екстремното програмиране значително намалява присъщото риск в проекта. Тази статия се занимава с постепенна промяна, която е основа на малки издания. Постепенна промяна въвежда нова функционалност в програмата. то е различни от други видове промени, които запазват, оттегляне или модифициране на съществуваща функционалност. Докладът представя методология за постепенно нарастване промяна.

Методологията е насочена повече към начинаещ а не при опитен програмист. Занимава се с концепции за домейни и тяхната роля в постепенни промени. Раздел 2 обяснява методологията. Раздел 3 описва казусът. Раздел 4 обобщава опита. Раздел 5 свързва тази работа с други изследвания и раздел 6 съдържа заключения и бъдеща работа.

2 ОСНОВИ НА МЕТОДОЛОГИЯТА

Постепенната промяна добавя нова функционалност към приложение. Забелязахме, че той изпълнява един или няколко тясно свързани концепции за домейн. Например Приложението Point-of-Sale трябва да се справя с няколко форми на плащането и следователно има постепенна промяна, която въвежда „кредитни карти“, „чекове“ и др. Също така забелязахме, че новите понятия, въведени в постепенните промени не са съвсем нови. Обикновено те са вече присъства в кода, макар и само в примитивен или имплицитна форма. Например, преди въвеждането на кредитни карти и чекове, понятието „плащане“ вече беше част от програмата. Концепцията обаче беше в примитивна форма, позволяваща само пари в брой, представени като справедливи едно число. Постепенната промяна реализира съответните понятия изрично и изцяло. Следователно понятията за домейн играят съществена роля при избора и прилагане на допълнителни промени. Понятията, които са зависими една от друга, трябва да бъдат изпълнени по реда на тяхната зависимост. За например понятието „данък“ зависи от понятието „Артикул“, защото различните артикули могат да имат различен данък ставки и данък без артикул е безсмислен.

Следователно, изпълнението на „артикул“ трябва да предшества прилагане на „данък“. Ако няколко понятия са взаимно зависими, те трябва да бъдат внедрени в едно и също постепенна промяна. Взаимно независими понятия могат да бъдат въведени в произволен ред, но е препоръчително да ги въведете в ред на важност за потребителя. Например в Програмата на точка на продажба е по-важно да се занимавате правилно с данъци, отколкото да издържат няколко касиера. An заявление с правилна подкрепа за данъци вече е използваем в магазини с един касиер. Обратен ред на постепенните промени биха отложили използваемостта на програма. Всяка постепенна промяна следва следната последователност стъпки:

1. Изберете концепциите на домейна, които ще бъдат въведени или доразвити.

2. Изберете тестовите случаи за новите концепции и внедряване на новите концепции като нови класове.

3. В стария код намерете къде вече са понятията настояще (често примитивно или имплицитно).

4. Рефакторирайте стария код, ако старите концепции са делокализирани и могат да бъдат локализирани в по-малко класове.

5. Променете старите класове, които съдържат стари концепции, така че те взаимодействат правилно с новите класове. 6. Разпространете промяната чрез останалия стар код доколкото е необходимо

Изпитна тема № 11:

Разработка на софтуер План-тезис:

Инструменти за разработка. Техники за продуктивно използване на интегрирана среда за разработка. Използване на външни библиотеки. Управление на пакети. Свързване на приложения с бази от данни. Създаване на приложения с няколко потребителски интерфейса. Примерна приложна задача:

Инструменти за разработка

Инструментите за разработка на софтуер са важни за вас, в противен случай не бихте попаднали в тази статия. Ефективният инструмент може значително да увеличи производителността на екипа. Не става въпрос само за инструменти за програмиране, но и за комуникация и управление на ресурси.

Нов инструмент за разработване на софтуер отнема време за внедряване, свикване и разбиране. Има много възможности, когато става въпрос за избор на платформа за разработка. Тази статия предоставя списък с често използвани инструменти, използвани в отделите за разработка.

1. Sublime Text

Sublime Text е инструмент за разработка на софтуер за редактиране на изходен код. Има много клавишни комбинации, които ви позволяват лесно да отваряте файлове и да превключвате между редовете. Можете да редактирате много редове наведнъж поради опцията за множествен избор. Можете също така лесно да превключвате между проекти. API на приставката също е силен.

2. Инструмент за разработване на софтуер Intellij

Intellij е интегрирана среда за разработка. IDE интегрира всички необходими инструменти. Така че има редактор на код, компилатор, дебъгер и др. Intellij завършва автоматично и е наистина удобен за потребителя и лесен за използване. Също така помага за бързо навигиране през вашия код и анализ на грешки. Това значително увеличава производителността.

3. Docker (Докър)

Docker предлага платформа за контейнериране на софтуер, която ви позволява да включите вашето приложение или софтуер в контейнер за файлове. Този контейнер може да бъде поставен и изнесен навсякъде. Ще намерите всичко необходимо за работа: код, системни библиотеки и др. Това означава, че софтуерът ще работи навсякъде по един и същ начин и е независим от средата.

4. Git

Git е система за контрол на версиите с отворен код за софтуерни проекти. Разработчикът трябва да прави чести промени в кода, преди да е готова последната версия.

Система за контрол на версиите съхранява всяка направена промяна, за да може екипът да работи заедно. Съхранява се копие на кода на всеки разработчик. С Git работата в екип може да се синхронизира, кодът се актуализира по всяко време.

5. GitHub

GitHub е хранилище на Git в Интернет. Това е като dropbox за софтуерни проекти, където можете да намерите код. Когато качвате проект, имате избор да го направите публичен или частен. Това е чудесно място за мрежи, срещи със съмишленици и споделяне на проекти. Общността е огромна и броят на проектите е още по-голям.

6. Jira

Jira е предназначена за Agile екипи за планиране, проследяване и пускане на софтуер. Очевидно е, че докато Agile екипите използват Jira. Той поддържа управление на проекти. Jira е изключително адаптивна и има мощни функции за всеки етап от развитието. Можете да управлявате напредъка, да визуализирате текущата работа и да генерирате отчети в един инструмент.

7. Jenkins (Дженкинс)

Jenkins е сървър за автоматизация с отворен код, по-точно сървър за непрекъсната интеграция. Можете да си представите този инструмент за разработка на софтуер като посредник между вашия код и вашия сървър за изграждане. Дженкинс периодично проверява за промени на вашия сървър и ги изпраща на сървъра за изграждане. Инструментът има чудесни приставки, които улесняват живота.

8. Slack

Slack е приложение, което може да се използва за комуникация в екип. Страхотен е, защото може да се използва навсякъде. Използването значително намалява вътрешното използване на електронна поща. Slack е много практичен: имате „канали“, които създавате за проекти, екипи, теми и др. Можете лесно да навигирате между каналите. Освен това имате възможност да изпращате директни лични съобщения до членовете на вашия екип. Можете да споделяте и коментирате всякакви файлове или да маркирате хора. Можете също да търсите в цялата си история.

9. Препълване на стека

Stack Overflow всъщност не е инструмент за разработка на софтуер. Това е сайт с въпроси и отговори с най-голямата общност от програмисти. В тази библиотека ще намерите всички отговори на вашите въпроси. От промяна на цвета на текста до промяна на ядрото на Linux. Това е чудесно място за учене и споделяне на знания.

10. YouTube

С YouTube можете да преглеждате уроци и съвети от други ИТ специалисти. Но също така и семинари, които сме пропуснали.

Интегрираната среда за разработка (Преговор)

Интегрираната среда за разработка (на английски: integrated development environment, IDE) e софтуерно приложение, което предоставя цялостна среда за разработване на софтуер на програмистите.

IDE обикновено се състои от редактор на код, инструменти за автоматизиране построяването на изходното приложение и дебъгер. Повечето съвременни интегрирани среди предлагат автоматично предсказване и дописване на кода.

Някои IDE-та включват компилатор, интерпретатор или и двете, като NetBeans и Eclipse; други нямат тази функционалност, например SharpDevelop и Lazarus. Не съществува добре дефинирана граница между интегрирана среда за разработка и друг вид инструмент за създаване на софтуер. В някои случаи в пакета са включени система за контрол на версиите или различни инструменти за дизайн на графичен интерфейс. Много съвременни среди за разработка имат също речник на класоветеи и диаграма с тяхната йерархията, за улеснение при обектно-ориентирано програмиране.

Техники за продуктивно използване на интегрирана среда за разработка

Клавишните комбинации ще ви спестят време, като ви позволяват да избягвате преместването на мишката. Нищо голямо нещо не казваш? Отнема няколко секунди или така всеки път, когато преместите ръката си от клавиатурата, намерите мишката, върнете очите си обратно към екрана, докато премествате мишката от редактора в менюто, щракнете, за да отворите меню, преместете мишката надолу до елемент от менюто, щракнете отново и преместете ръцете си обратно към клавиатурата. Загубили сте време и нервна енергия, нито една от които никога няма да се върнете. Палците ви вече са на клавиатурата, ако просто редактирате кода, така че просто напишете CTRL / Shift / O, за да коригирате всички импорти, или Alt / Shift / X, за да стартирате програмата си. И не използвайте менюто File-> Save, преди да стартирате, и дори не използвайте CTRL / S, за да запазвате, преди да стартирате, защото всички разумни IDE по подразбиране автоматично записват файлове, преди да ги стартирате - d’oh!

Тук стриктно се спазва нескромно нареченото правило на Ян:

Никога не въвеждайте повече от първите 3 или 4 знака от всяко име, което е известно на IDE.

Въведете @ T R CTRL / Space и Eclipse ще въведе анотацията @Transactional вместо вас. Въвеждайки syso CTRL / Space, за вас ще се въведе System.out.println (). Eclipse нарича последния шаблон. Много полезни предварително дефинирани шаблони се доставят с Eclipse и можете да добавите свой собствен.

Винаги, когато използвате елемент от менюто, погледнете дали има клавишна комбинация. Те се показват в менюто, за да можете да ги запомните. Научете по един на ден и сте на път!

Максималният фактор?

Рефакторинга ще ви спести време, като изпълнява сложни за вас операции. Искате да преименувате публичен метод, използван от дузина други класове във вашия проект. Въвеждате името на метода, където е дефинирано. Но сега трябва да влезете във всеки от тези други файлове (които ще показват червени, веднага щом запишете файла с дефиницията), и да направите промяната там. Дори ако сте започнали с използване на търсене и замяна на дефиницията, пак ще трябва да отворите всички останали файлове в редактора. Още по-лошо, какво ще стане, ако името, което замествате, е подниз от друг метод? Какво ще кажете за промяна на едното, но не и на другото в коментарите на кода?

Решението? Рефактор-> Преименуване. IDE знае целия код във всички класове във вашия проект. Изберете старото име, въведете CTRL / Shift / R и въведете новото име. Бум! Готово сте през целия проект.

Друг пример: добавяне на разпечатка за отстраняване на грешки на стойност, която е част от сложен израз. Бихте могли да направите това на ръка, но защо бихте? Изберете частта, която искате да видите, натиснете Alt / Shift / L (Извличане на локална променлива) и изберете име за новата си локална променлива. За да го отпечатате, използвайте шаблона SYSO Control / Space. За да отмените това, изберете променливата и използвайте Alt / Shift / I, за да я вмъкнете. Искате ли да извадите малко код от един метод, за да направите нов метод за повторна употреба или за тестване? Изберете редовете на кода и направете CTRL / Shift / M.

Меню за рефакторинг

IDE разполагат с Меню за рефакторинг

Има много полезни рефакторинга и те са склонни да имат подобни имена в различни IDE поради каталог, наречен Refactorings. Определям рефакторинга като „промяна, запазваща поведението, която подобрява кода“. Рефакторинга, извършен от IDE, е много по-малко вероятно да внесе грешки, отколкото разработчикът, който прави същата задача на ръка. Така че, използвайте рефакторинга!

Плугини – Приставки (Plugins)

Приставките ще ви спестят време, като изпълнявате по-големи и / или по-специализирани задачи. Повечето IDE днес се състоят от малък двигател, с по-голямата част от функционалността в приставки. Например в Eclipse, Java Editor е плъгин, както и почти всичко, с което взаимодействате. Голяма колекция от приставки идва с пълната инсталация на типичния IDE. Приставките могат да показват информация или да ви позволяват да я редактирате или обработвате по някакъв начин. Съществуват специализирани приставки за:

* много езици за програмиране
* редактиране на диаграми като UML и потоци от процеси (в повечето случаи също генериране на кода, който диаграмите представляват)
* взаимодействие с проследяващи програмни грешки и непрекъснати работни процеси на интеграция
* изграждане и опаковане на различни видове приложения и за интегриране с Maven или Gradle до същото
* и още много.

Като един пример, приставката Maven (M2E) поддържа пътя на класа на вашия проект от файла за изграждане на Maven, като елиминира необходимостта от поддържане на две версии на този списък с Jar файлове. Възможностите на приставките за разширяване на съществуваща IDE са почти безкрайни. Клиентът на Eclipse Marketplace (в менюто Помощ) каталогизира по-голямата част от тях и ще инсталира всеки от тях с едно или две щраквания.

Погледнете напред към производителността

Виж. Трябва да отделите малко време, за да спестите малко време. Но си струва. Ще се възползвате всеки път, когато пишете или поддържате код. Имам референтна карта за преки пътища на Eclipse.

Следващия път, когато шефът ви отсъства на конференция, прекарайте един час в проучване на тези преки пътища, рефакторинг и плъгини. Докато шефът се върне, вие ще бъдете много по-продуктивни. Ще отделите по-малко време за работа с мрънкане и ще имате повече време да се съсредоточите върху алгоритмите и архитектурата. И мечтайте да станете шеф.

Използване на външни библиотеки

В областта на програмирането, библиотеката е колекция от помощни ресурси под формата на код и данни, съхранявани в енергонезависимата памет, предназначени за използване от независими програми, например от операционната система или от програми за разработка на софтуер. Това позволява споделяне и промяна на данни и код на модулен принцип.

Повечето компилиращи програмни езици имат стандартна библиотека, макар че софтуерните разработчици могат да създават собствени.

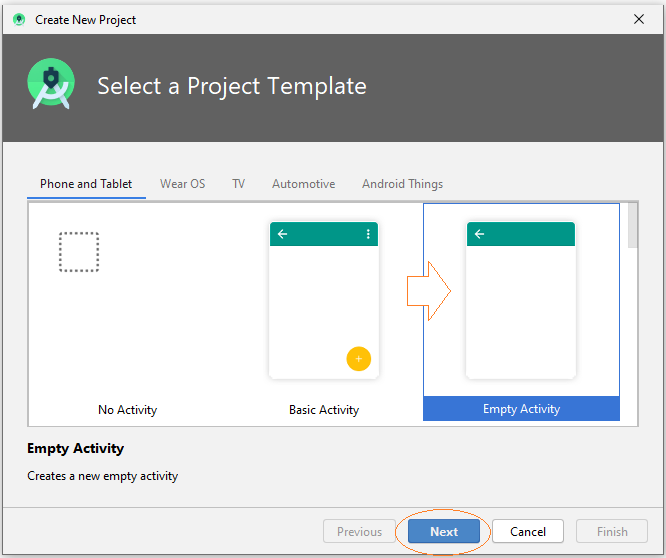
В този документ ще ви насоча към 3 начина за използване на външната библиотека:

Добавете вашите jar файлове в папката libs на проекта и го декларирайте като библиотека за използване.

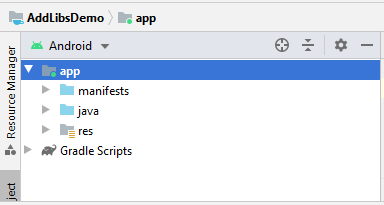
Създайте Android модул и копирайте вашия jar файл в този модул и след това декларирайте проекта си с помощта на новосъздадения модул.

Декларирайте и използвайте отдалечена библиотека.

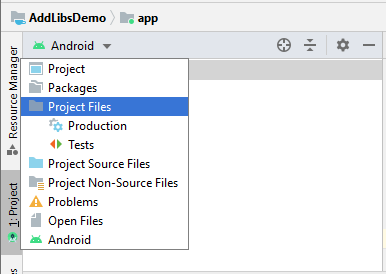
Create a project named **AddLibsDemo**:



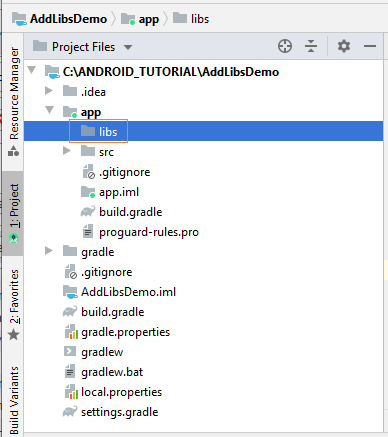
OK, your project has been created.

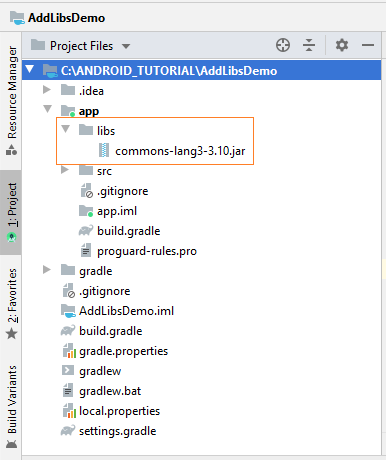


Change to **Project** tab:



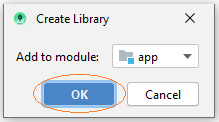
Copy jar file to **libs** folder:



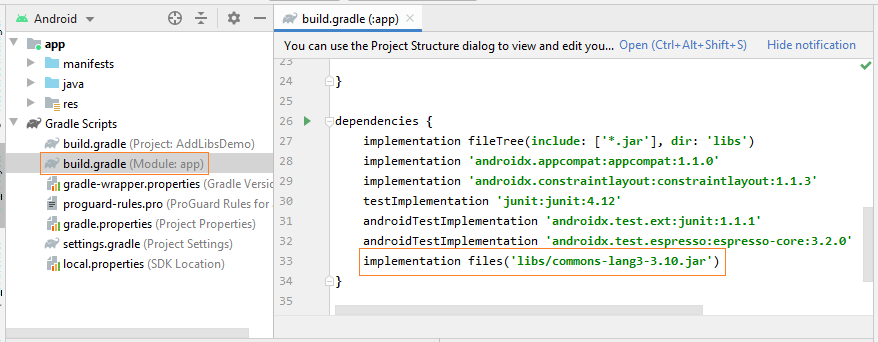


Right click jar files, select **"Add as Library.."**:





Turn back to "**Android"**Tab, you can see that your library has been already declared in  ***build.grade (Module: app)***



Управление на пакети

Мениджър на пакети или система за управление на пакети е колекция от софтуерни инструменти, която автоматизира процеса на инсталиране, надстройка, конфигуриране и премахване на компютърни програми за операционната система на компютъра по последователен начин.

Какво точно представлява мениджърът на пакети?

Вече срещнахме npm, но отстъпвайки от самия npm, мениджърът на пакети е система, която ще управлява зависимостите на вашия проект.

Мениджърът на пакети ще предостави метод за инсталиране на нови зависимости (наричани също "пакети"), ще управлява къде пакетите се съхраняват във вашата файлова система и ще предлага възможности за публикуване на собствени пакети.

На теория може да не ви е необходим мениджър на пакети и бихте могли ръчно да изтеглите и съхраните зависимостите на вашия проект, но мениджърът на пакети ще се справя безпроблемно с инсталирането и деинсталирането на пакети. Ако не сте използвали такъв, ще трябва да обработите ръчно:

Намиране на всички правилни JavaScript файлове.

Проверявайте ги, за да се уверите, че нямат известни уязвимости.

Изтеглете ги и ги поставете на правилните места във вашия проект.

Писане на кода за включване на пакета (ите) във вашето приложение (това обикновено се прави с помощта на JavaScript модули, друга тема, която си струва да се прочете и разбере).

Правенето на едно и също нещо за всички зависими пакети, от които може да са десетки или стотици.

Премахване на всички файлове отново, ако искате да премахнете пакетите.

В допълнение, мениджърите на пакети се справят с дублиращи се зависимости (нещо, което става важно и често срещано при разработката отпред).

В случай на npm (и базирани на JavaScript и Node мениджъри на пакети) имате две възможности за това къде да инсталирате вашите зависимости. Както споменахме в предишната статия, зависимостите могат да бъдат инсталирани глобално или локално към вашия проект. Въпреки че обикновено има повече плюсове за глобално инсталиране, плюсовете за локално инсталиране са по-важни - като преносимост на кода и заключване на версии.

Например, ако вашият проект се разчита на Webpack с определена конфигурация, бихте искали да се уверите, че ако сте инсталирали този проект на друга машина или сте се върнали към него много по-късно, конфигурацията ще продължи да работи. Ако е инсталирана различна версия на Webpack, тя може да не е съвместима. За да се смекчат тези зависимости се инсталират локално към проект.

За да видите, че локалните зависимости наистина блестят, всичко, което трябва да направите, е да опитате да изтеглите и стартирате съществуващ проект - ако той работи и всички зависимости работят веднага, можете да благодарите за факта, че кодът е преносим.

Забележка: npm не е единственият наличен мениджър на пакети. Успешен и популярен алтернативен мениджър на пакети е Прежда. Преждата разрешава зависимостите, използвайки различен алгоритъм, който може да означава по-бързо потребителско изживяване. Има и редица други нововъзникващи клиенти, като pnpm.

Регистри на пакети

За да работи мениджърът на пакети, той трябва да знае откъде да инсталира пакети и това се предлага под формата на регистър на пакети. Регистърът е централно място, на което се публикува пакет и по този начин може да се инсталира от него. npm, освен че е мениджър на пакети, е и името на най-често използвания регистър на пакети за JavaScript пакети. Регистърът на npm съществува на npmjs.com.

npm не е единствената опция. Можете да управлявате свой собствен регистър на пакети - продукти като Microsoft Azure ви позволяват да създавате прокси в npm регистъра (така че можете да замените или заключите определени пакети), GitHub предлага и услуга за регистрация на пакети и вероятно ще има повече опции, които се появяват като време продължава.

Важното е да сте сигурни, че сте избрали най-добрия регистър за вас. Много проекти ще използват npm и ние ще се придържаме към това в нашите примери в останалата част от модула.

Ефективният инструмент може значително да увеличи производителността на екипа

C # Връзка с база данни: Как да свържете SQL Server (пример)

Достъпът до данни от база данни е един от важните аспекти на всеки език за програмиране. Абсолютна необходимост е всеки език за програмиране да има способността да работи с бази данни. C # не е по-различен.

Може да работи с различни видове бази данни. Той може да работи с най-често срещаните бази данни като Oracle и Microsoft SQL Server.

Също така може да работи с нови форми на бази данни като MongoDB и MySQL.

В този урок за свързване на C # sql ще научите -

Основи на свързаността с бази данни

Как да свържете C # към база данни

Достъп до данни с SqlDataReader

C # Вмъкване в база данни

Актуализиране на записите

Изтриване на записи

Свързване на контроли за данни

C # DataGridView

Основи на свързаността с бази данни

C # и .Net могат да работят с повечето бази данни, като най-често срещаните са Oracle и Microsoft SQL Server. Но с всяка база данни логиката зад работата с всички тях е предимно една и съща.

В нашите примери ще разгледаме работата на Microsoft SQL Server като наша база данни. За целите на обучението може да се изтегли и използва Microsoft SQL Server Express Edition, който е безплатен софтуер за бази данни, предоставен от Microsoft.

При работа с бази данни по-долу са представени понятията, които са общи за всички бази данни.

Връзка - За да работите с данните в база данни, първата очевидна стъпка е връзката. Връзката с база данни обикновено се състои от посочените по-долу параметри.

Име на базата данни или Източник на данни - Първият важен параметър е името на базата данни, към което трябва да се установи връзката. Всяка връзка може да работи само с една база данни наведнъж.

Удостоверения - Следващият важен аспект е потребителското име и паролата, които трябва да се използват за установяване на връзка с базата данни. Той гарантира, че потребителското име и паролата имат необходимите привилегии за свързване с базата данни.

Незадължителни параметри - За всеки тип база данни можете да посочите незадължителни параметри, за да предоставите повече информация за това как .net трябва да обработва връзката с базата данни. Например, може да се посочи параметър за това колко дълго връзката трябва да остане активна. Ако не се извършва операция за определен период от време, тогава параметърът ще определи дали връзката трябва да бъде затворена.

Избор на данни от базата данни - След като връзката е установена, следващият важен аспект е извличането на данните от базата данни. C # може да изпълни командата за избор "SQL" срещу базата данни. Операторът „SQL“ може да се използва за извличане на данни от конкретна таблица в базата данни.

Вмъкване на данни в базата данни - C # може да се използва и за вмъкване на записи в базата данни. Стойностите могат да бъдат посочени в C # за всеки ред, който трябва да бъде вмъкнат в базата данни.

Актуализиране на данни в базата данни - C # може да се използва и за актуализиране на съществуващи записи в базата данни. Нови стойности могат да бъдат посочени в C # за всеки ред, който трябва да бъде актуализиран в базата данни.

Изтриване на данни от база данни - C # може да се използва и за изтриване на записи в базата данни. Изберете команди, за да посочите кои редове трябва да бъдат изтрити, могат да бъдат посочени в C #.

Добре, сега, когато видяхме теорията на всяка операция, нека да преминем към следващите раздели, за да разгледаме как можем да извършваме операции с база данни в C #.

SQL команда в c #

SqlCommand в C # позволява на потребителя да прави заявки и да изпраща командите към базата данни. SQL командата се определя от обекта на SQL връзка. Използват се два метода, метод ExecuteReader за резултатите от заявката и ExecuteNonQuery за команди за вмъкване, актуализиране и изтриване. Това е методът, който е най-подходящ за различните команди.

Как да свържете C # към база данни

Нека сега разгледаме кода, който трябва да се поддържа, за да се създаде връзка с база данни. В нашия пример ще се свържем с база данни, която носи името Demodb. Идентификационните данни, използвани за свързване с базата данни са дадени по-долу

Потребителско име -

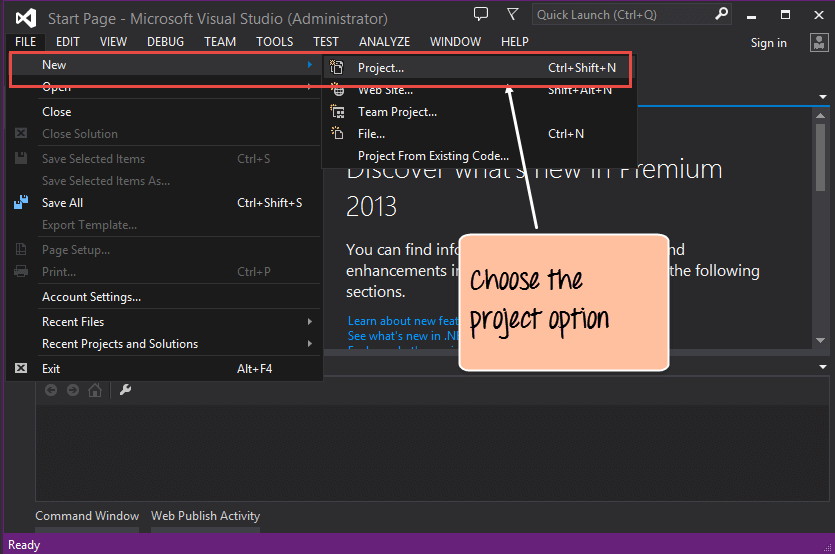
Парола - демо123

Ще видим просто приложение за Windows формуляри за работа с бази данни. Ще имаме прост бутон, наречен "Свързване", който ще се използва за свързване с базата данни.

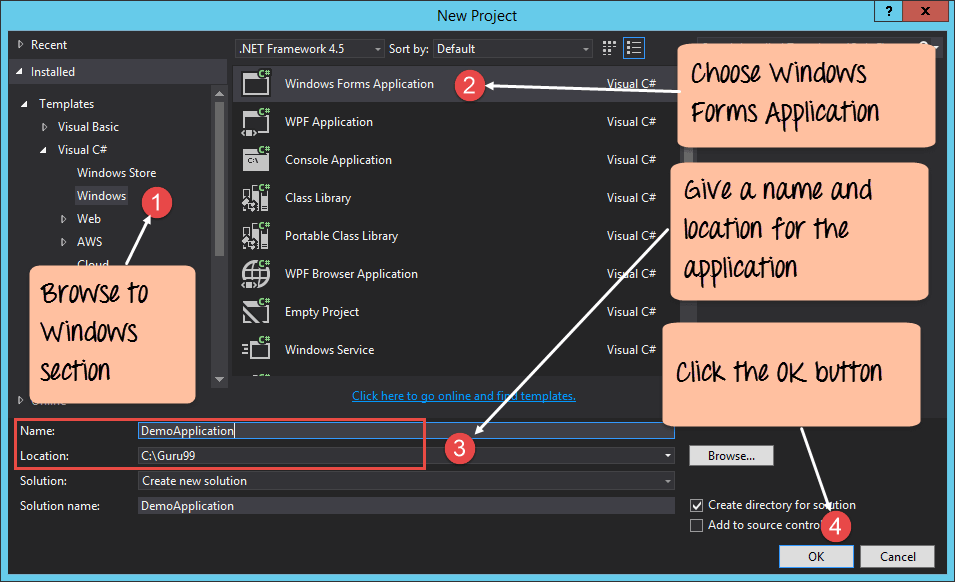
Така че нека следваме стъпките по-долу, за да постигнем това

Стъпка 1) Първата стъпка включва създаването на нов проект в Visual Studio. След стартирането на Visual Studio трябва да изберете опцията от менюто New-> Project.

**Step 1)**The first step involves the creation of a new project in Visual Studio. After launching Visual Studio, you need to choose the menu option New->Project.

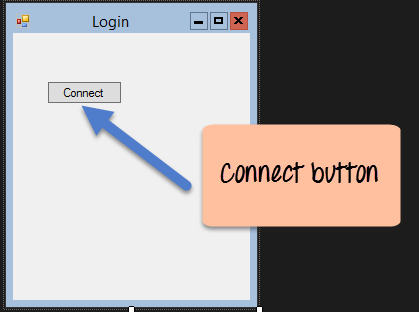
[](https://www.guru99.com/images/c-sharp-net/052716_0506_CAccessData1.png)

**Step 2)** The next step is to choose the project type as a Windows Forms application. Here, we also need to mention the name and location of our project.

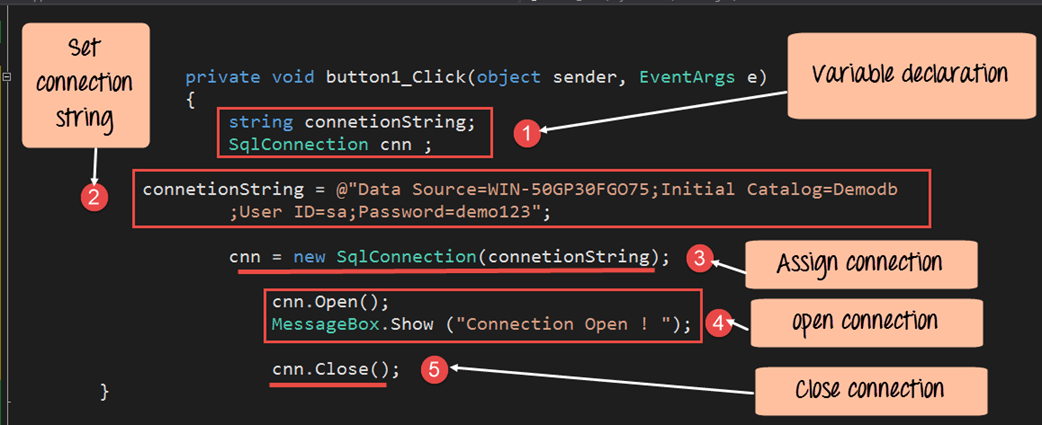
[](https://www.guru99.com/images/c-sharp-net/052716_0506_CAccessData2.png)

1. In the project dialog box, we can see various options for creating different types of projects in Visual Studio. Click the Windows option on the left-hand side.
2. When we click the Windows options in the previous step, we will be able to see an option for Windows Forms Application. Click this option.
3. We then give a name for the application which in our case is "DemoApplication". We also need to provide a location to store our application.
4. Finally, we click the 'OK' button to let Visual Studio to create our project.

**Step 3)** Now add a button from the toolbox to the Windows form. Put the text property of the Button as Connect. This is how it will look like

[](https://www.guru99.com/images/c-sharp-net/052716_0506_CAccessData3.png)

**Step 4)** Now double click the form so that an event handler is added to the code for the button click event. In the event handler, add the below code.

[](https://www.guru99.com/images/c-sharp-net/052716_0506_CAccessData4.png)

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Data.SqlClient;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace DemoApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

string connetionString;

SqlConnection cnn;

connetionString = @"Data Source=WIN-50GP30FGO75;Initial Catalog=Demodb;User ID=sa;Password=demol23";

cnn = new SqlConnection(connetionString);

cnn.Open();

MessageBox.Show("Connection Open !");

cnn.Close();

}

}

}

Създаване на приложения с няколко потребителски интерфейса

В днешно време крайните потребители могат да взаимодействат с услуга и информация, използвайки различни видове изчислителни платформи, включително традиционни офис настолни компютри, интелигентни телевизори, таблети и мобилни телефони. Това позволява на потребителите да мигрират своите задачи от един потребителски интерфейс към друг на различни устройства или платформи. Например потребителят може да търси ресторант от конкретна услуга и след това да превключи до изображението на услугата от мобилния им телефон, за да намерите информацията за връзка с ресторанта, и след това може да транзитира, за да използва таблет, за да напише отзив за ресторанта. Това води до нов потребител изживяваща тема, при която потребителят взаимодейства с множество потребителски интерфейси (MUI), за да постигне цели хоризонтално (през платформи). Този тип MUI достъп е различен от традиционния потребител опит, включващ взаимодействие с един потребителски интерфейс (вертикално взаимодействие). Има нови аспекти в MUI взаимодействието, включително превключване на процес от един потребителски интерфейс към друг, мигриране на знания и задачи от един потребителски интерфейс към друг.

Има четири разпространени типа потребителски интерфейс и всеки има редица предимства и недостатъци:

1. Интерфейс на командния ред.
2. Интерфейс, управляван от меню.
3. Графичен потребителски интерфейс.
4. Сензорен графичен потребителски интерфейс.

Какво е услуга? Услугата се отнася до софтуер и хардуер, в който една или повече услуги могат да бъдат използвани за поддръжка на нуждите и функциите на бизнеса. Има два основни типа услуги: атомни и композитни. Атомната услуга е самостоятелна функция, която не изисква използването на други услуги. Композитната услуга е съвкупност от атомни или други композитни услуги, които могат изискват използването на друга услуга, съдържаща се в същата композитна услуга.

2.2. Какво е крос-платформа? Терминът междуплатформена платформа може да се използва за характеризиране на различни единици в компютърните науки. За Например, хардуерни устройства, като компютърни монитори, могат да бъдат описани като междуплатформени като те могат да работят с всяка операционна система. По подобен начин могат да бъдат езици за програмиране, като C описани като междуплатформени, тъй като те могат да се използват за писане на софтуер за използване във всяка операционна система. Освен това терминът може да се използва за означаване на софтуер, който може да работи на повече от една платформа. За целите на тази статия използваме термина междуплатформена платформа, за да се позова на услуга, която може да бъде достъпни и използвани на две или повече изчислителни платформи.

2.3. Свързване на услуги Уеб услугите предоставят технологиите за свързване на услугите заедно. За различни платформи услуги, уеб услугата може да бъде определена като система, която може да бъде проектирана да поддържа оперативно съвместима комуникация от приложение към приложение през мрежа. Оперативна съвместимост може се отнасят както за синтактична оперативна съвместимост, така и за семантична оперативна съвместимост. Синтактичен оперативната съвместимост зависи от определени формати за данни и комуникационни протоколи, за да се гарантира комуникация и обмен на данни между разнородни софтуерни приложения. Със синтактични оперативна съвместимост, няма гаранция за последователни интерпретации на обменяните данни от един приложение към друг. Семантичната оперативна съвместимост се отнася до способността на различни услуги в цялата платформи за тълкуване на обменяната информация смислено и точно. Има множество технологии на уеб услуги за услуги за свързване, включително използването на SOAP, WSDL, UDDI, REST, XML и JSON, които са обяснени накратко, както следва:

1. Прост протокол за достъп до обекти (SOAP) е протокол за активиране на комуникацията между приложенията. Компютърни науки и информационни технологии (CS & IT) 45

2. Езици за описание на уеб услуги (WSDL) се използва за дефиниране на интерфейси на уеб услуги, типове данни и съобщения и картографиране на протоколи.

3. Universal, Description, Discovery и Integration (UDDI) е регистър на уеб услуги и механизъм за откриване, използван за сортиране на бизнес информация и извличане на указатели към интерфейс на уеб услуга.

4. Разширяем език за маркиране (XML) предоставя език за дефиниране на данни и обработвайки го.

5. Представителният държавен трансфер (REST) ​​е алтернатива на SOAP, който е разработен на a набор от принципи, описващи как се дефинират и адресират мрежовите ресурси.

6. JSON (JavaScript Object Notation) е алтернатива на XML, който използва двойки име / стойност вместо тагове, както се използват в XML.

2.4. Конфигуриране на междуплатформени услуги Услугата на различни платформи има за цел да осигури всеобхватна и синергична подкрепа за човешките дейности в различни контексти на употреба.

Feiner представи концепцията за хибридни потребителски интерфейси, в която множество хетерогенни дисплеи и устройства за взаимодействие се използват синергично, за да се възползват характеристиките за всеки от тях. Услугите между устройствата могат да бъдат конфигурирани въз основа на различен потребител и / или бизнес нужди, като се имат предвид различни ограничения и възможности на устройствата. Конфигурация на междуплатформената услуга се отнася до начина, по който устройствата са организирани и услугата е доставяни през тези устройства. Преди да обсъдите организацията на устройството и предоставянето на услуги, трябва да изясним концепцията за синергична специфичност, която е свързана с различни методи за конфигуриране.

Изпитна тема № 12:

Операционни системи План-тезис:

Структура на компютърните системи и операционни системи. Процеси и памет. Команди и команден интерпретатор. Пакетни системи и инсталиране на софтуер в операционните системи. Примерна приложна задача

Структура на компютърните системи

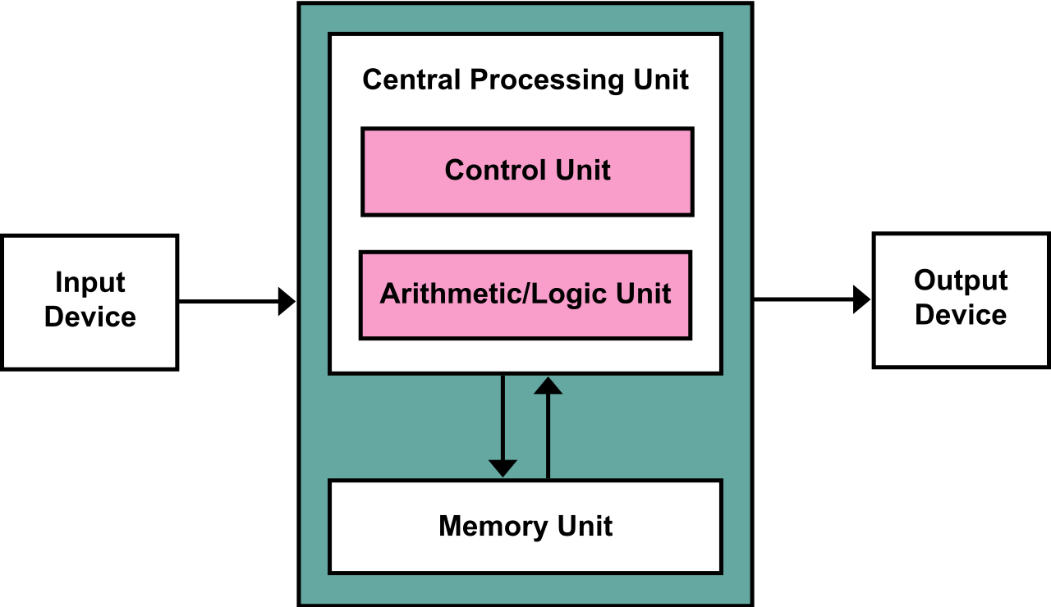
Компютърният хардуер (в миналото известен в България повече като компютърна апаратна част или компютърно апаратно осигуряване) е съвкупността от всички физически компоненти на компютъра, както и от всички периферни устройства като монитор, мишка, принтер, клавиатура и други.

Най-общо хардуерът обхваща всички компютри на дадено място, било то настолни, мобилни или сървърни машини. Състои се от няколко основни компонента, които съставляват всеки един компютър: централен процесор, дънна платка, оперативна памет. Към тях се добавят входно-изходни устройства, които преди години са били в един вид – например четец на перфокарти, телетип и принтер – а днес най-често са клавиатура, мишка и монитор, както и всевъзможни други устройства, които разширяват функционалността на компютъра: скенери, принтери, акустични системи, джойстик, графичен таблет, модем и други.

* Централен процесор.
* Оперативна памет.
  + Енергозависима.
* Входно/изходни устройства.
  + Външни запаметяващи устройства.
  + Комуникационни и мрежови устройства.
  + Терминали.
* Системна шина.

- Осъществява връзката между процесора, паметта и входно/изходните устройства.

## **Архитектура на фон Нойман**



Архитектура на фон Нойман

Основата на всички модерни компютри е архитектурата на фон Нойман, описана подробно през 1945 година на хартия от унгарския математик Джон фон Нойман. Той описва архитектурата на електронен цифров компютър с подразделения от централен процесор, следван от аритметично логическо устройство и процесорни регистри, контролер, съдържащ регистър с инструкции, и програмен брояч, памет в която да бъдат запаметени данните и инструкциите, външно съхранявани данни и входно-изходни механизми. Значението на термина се е развило до „програмируем компютър“, в които изпълняването на инструкция и операция с данни не могат да се появят по едно и също време, защото делят една и съща шина. Това се нарича фон Нойман „bottleneck“ и често ограничава производителността на системата.

Дефиниция за операционна система

Операционната система (ОС) е основна част от компютърния системен софтуер, която управлява и координира ресурсите на хардуера и софтуера и обслужва изпълняваните компютърни програми. Приложният софтуер обикновено има нужда от ОС, за да работи.

ОС разпределя задачите по време, планира ефикасното използване на ресурсите на системата и може да включва специализиран софтуер за изчисление на ресурси: стойност на процесорното време, използвана памет, ресурси за печат и други. При входно-изходните операции и динамично разпределение на паметта ОС действа като посредник между приложния софтуер и хардуера, макар че приложните команди обикновено се изпълняват директно, чрез системни повиквания към функции на ОС. ОС се среща на повечето устройства с компютри – от мобилни телефони през игрални конзоли до уеб сървъри и суперкомпютри.

* Операционната система може да се разглежда като:
* програма, която управлява и разпределя ресурсите на компютърната система;
* слой, който предоставя абстрактен интерфейс към хардуерните компоненти на компютъра.

Примери за популярни съвременни ОС са: AIX, Android, BlackBerry, BSD, Chrome OS, iOS, GNU/Linux, macOS, QNX, Microsoft Windows, SteamOS, Windows Phone и z/OS. Първите девет имат общ произход, свързан с UNIX. Други популярни ОС в реално време са FreeRTOS, Micrium и VxWorks.

Процесор

Процесорът е електронна схема, предназначена да извършва аритметически и логически действия/операции върху данни, „подадени“ му от външен източник – обикновено памет. Понятието често се използва за обозначаване на централния процесор на дадена компютърна система, но в много случаи последните използват и допълнителни специализирани процесори. Съвременните процесори са с тактова честота до 5 GHz. Създаването на процесори с по-големи тактови честоти е трудно по физични причини, защото драстично се повишава консумацията на енергия, както и отделянето на топлинна енергия.

Алтернативно решение за повишаване на бързодействието на компютрите са многоядрените процесори. Те предлагат възможност няколко ядра в един процесор да работят заедно. Ядрата може да са 2, 4, 6, 8 или повече. Бързодействието на такъв процесор се измерва, като се умножи честотата по броя на ядрата и по коефициент за инструкции за цикъл.

Разрядността е характеристика на процесорите, определяща броя битове, които процесорът може да обработи и предаде едновременно. Най-често използваните процесори са с разрядност 32 или 64 бита.

Кеш паметта е високопроизводителна памет, която съхранява най-често използваните команди или данни. Измерва се в МВ и съвременните процесори имат до 32 МВ кеш памет.

* Процесорът изпълнява инструкциите на програмата.
* Инструкциите на програмата и данните се съхраняват в оперативната памет.
* Регистрите на процесора са високоскоростна памет, разположена в самия процесор.
* Данните трябва да са в регистрите на процесора, за да може аритметико-логическия блок да ги обработва.
* За да се изпълни дадена инструкция тя първо трябва да се извлече (fetch) и декодира (decode).

Памет

Компютърната памет в компютрите е средата за съхранение на обработваните данни в течение на определено време. Притежава йерархична структура и обикновено е съставена от няколко запаметяващи устройства с различни характеристики.

Основната задача на компютърната памет е да съхранява информация, получена в резултат на външно въздействие и представена в подходящ вид. В случая на двоично представяне това означава записване на две състояния – 0 и 1 и може да се онагледи със състоянието „включено/изключено“ на ключ или реле. За управление на паметта се въвежда вътрешна организация, като тя се разделя на участъци (клетки на английски: memory cell) с определен размер (дължина или разредност) и адрес (пореден номер), и към тях се отправят команди от процесора. Достъпът до паметта се осъществява от специални механизми, които могат да четат и записват.

Съществуват различни видове компютърна памет и класификацията ѝ се прави според нейната функционалност, като включва освен нейната физическата реализация, така и структурата от данни, която в повечето случаи се реализира програмно. Запаметяващите устройства имат различна класификация.

Пакетни системи и инсталиране на софтуер в операционните системи.

Инсталирането (или настройката) на компютърна програма (включително драйвери на устройства и плъгини) е актът за подготвяне на програмата за изпълнение. Инсталацията се отнася до конкретната конфигурация на софтуер или хардуер с оглед да го направи използваем с компютъра. За инсталирането му е необходимо меко или цифрово копие на софтуера (програмата). Има различни процеси на инсталиране на софтуер (програма). Тъй като процесът варира за всяка програма и всеки компютър, програмите (включително операционните системи) често се доставят с инсталатор, специализирана програма, отговорна за извършване на каквото е необходимо (вижте по-долу) за инсталацията. Инсталацията може да е част от по-голям процес на внедряване на софтуер.

Инсталацията обикновено включва код (програма), който се копира / генерира от инсталационните файлове в нови файлове на локалния компютър за по-лесен достъп от операционната система, създаване на необходими директории, регистриране на променливи на средата, осигуряване на отделна програма за деинсталация и т.н. обикновено се копира / генерира на множество места, деинсталирането обикновено включва повече от просто изтриване на програмната папка. Например, файловете на системния регистър и другият системен код може да се наложи да бъдат модифицирани или изтрити за пълно деинсталиране.

Някои компютърни програми могат да бъдат изпълнени, като просто ги копирате в папка, съхранена на компютър и ги изпълните. Други програми се доставят във форма, неподходяща за незабавно изпълнение и поради това се нуждаят от процедура за инсталиране. Веднъж инсталирана, програмата може да се изпълнява отново и отново, без да е необходимо да се преинсталира преди всяко изпълнение.

Общите операции, извършвани по време на инсталирането на софтуер, включват:

Уверете се, че са изпълнени необходимите системни изисквания

Проверка за съществуващи версии на софтуера

Създаване или актуализиране на програмни файлове и папки

Добавяне на конфигурационни данни като конфигурационни файлове, записи в системния регистър на Windows или променливи на средата

Направете софтуера достъпен за потребителя, например чрез създаване на връзки, преки пътища или отметки

Конфигуриране на компоненти, които се изпълняват автоматично, като демони или услуги на Windows

Извършване на активиране на продукта

Актуализиране на версиите на софтуера

Изпитна тема № 13:

Вградени системи План-тезис:

Основи на електротехниката и електрониката. Електронни елементи: резистори, кондензатори, светодиоди, бутони и потенциометри. Аналогово/цифрови входове и изходи. Сензори. Управление на периферия. Широчинно импулсна модулация. Примерна приложна задача:

Основи на електротехниката

Електротехниката е инженерна дисциплина, занимаваща се с проучване, проектиране и приложение на оборудване, устройства и системи, които използват електричество, електроника и електромагнетизъм. Той се появява като идентифицируема професия през втората половина на 19 век след комерсиализацията на електрическия телеграф, телефона и производството, разпределението и използването на електрическа енергия.

Понастоящем електротехниката е разделена на широк спектър от области, включително компютърно инженерство, системно инженерство, енергетика, телекомуникации, радиочестотно инженерство, обработка на сигнали, инструментариум, електроника и оптика и фотоника. Много от тези дисциплини се припокриват с други инженерни отрасли, обхващащи огромен брой специализации, включително хардуерно инженерство, силова електроника, електромагнетика и вълни, микровълново инженерство, нанотехнологии, електрохимия, възобновяеми енергии, мехатроника и наука за електрическите материали. [A]

Електроинженерите обикновено са завършили електротехника или електронно инженерство. Практикуващите инженери могат да имат професионална сертификация и да бъдат членове на професионален орган или международна организация по стандартизация. Те включват Международната електротехническа комисия (IEC), Института на инженерите по електротехника и електроника (IEEE) и Института по инженерство и технологии (IET) (по-рано IEE).

Електроинженерите работят в много широк спектър от индустрии и необходимите умения също са променливи. Те варират от теория на веригите до управленски умения на ръководител на проекти. Инструментите и оборудването, от които може да се нуждае отделен инженер, са по същия начин променливи, вариращи от обикновен волтметър до сложен софтуер за проектиране и производство.

Eлектроника

Електрониката включва физика, инженерство, технологии и приложения, които се занимават с излъчването, потока и контрола на електроните във вакуум и материя. Той използва активни устройства за управление на електронния поток чрез усилване и коригиране, което го отличава от класическата електротехника, която използва пасивни ефекти като съпротивление, капацитет и индуктивност за управление на текущия поток.

Електрониката оказа голям ефект върху развитието на съвременното общество. Идентифицирането на електрона през 1897 г., заедно с последващото изобретение на вакуумната тръба, която може да усили и коригира малки електрически сигнали, откриха областта на електрониката и електронната ера. Това разграничение започва около 1906 г. с изобретението на Лий Де Форест на триода, което прави електрическо усилване на слаби радиосигнали и аудио сигнали възможно с немеханично устройство. До 1950 г. тази област се нарича „радиотехника“, тъй като основното й приложение е проектирането и теорията на радиопредавателите, приемниците и вакуумните лампи.

Терминът "полупроводникова електроника" се появява след като първият работещ транзистор е изобретен от Уилям Шокли, Уолтър Хаузър Братейн и Джон Бардин в Bell Labs през 1947 г. MOSFET (MOS транзисторът) е изобретен по-късно от Мохамед Атала и Дауон Канг в Bell Labs през 1959 г. MOSFET е първият наистина компактен транзистор, който може да бъде миниатюризиран и масово произвеждан за широка гама от приложения, революционизирайки електронната индустрия и играещ централна роля в микроелектронната революция и цифровата революция. Оттогава MOSFET се превърна в основен елемент в повечето съвременни електронни съоръжения и е най-широко използваното електронно устройство в света.

Електрониката се използва широко в обработката на информация, телекомуникациите и обработката на сигнали. Способността на електронните устройства да действат като превключватели прави възможна цифровата обработка на информацията. Технологиите за взаимно свързване като платки, технология за опаковане на електроника и други разнообразни форми на комуникационна инфраструктура допълват функционалността на веригата и трансформират смесените електронни компоненти в обикновена работеща система, наречена електронна система; примери са компютри или системи за управление. Електронната система може да бъде компонент на друга инженерна система или самостоятелно устройство. От 2019 г. повечето електронни устройства използват полупроводникови компоненти, за да извършват електронно управление. Обикновено електронните устройства съдържат верига, състояща се от активни полупроводници, допълнени с пасивни елементи; такава схема е описана като електронна схема. Електрониката се занимава с електрически вериги, които включват активни електрически компоненти като вакуумни тръби, транзистори, диоди, интегрални схеми, оптоелектроника и сензори, свързани пасивни електрически компоненти и технологии за взаимно свързване. Нелинейното поведение на активните компоненти и способността им да контролират електронните потоци прави възможно усилването на слабите сигнали.

Изследването на полупроводникови устройства и свързаните с тях технологии се счита за клон на твърдотелната електроника.

Резистор

Резисторът, наричан също (на жаргон) съпротивление, е двуизводен пасивен електронен компонент, чиято основна характеристика е електрическото съпротивление. При подаване на напрежение U между двата извода на резистора през него протича ток I, който е правопропорционален на напрежението. Отношението на напрежението към тока се нарича съпротивление (R):



Резисторите са сред най-често срещаните компоненти в електрическите вериги, като повечето електронни устройства съдържат голям брой от тях. Те се изработват от различни химически вещества във вид на компактни елементи, или в някои случаи – от навит проводник с голямо специфично съпротивление. Резистори се използват и в интегралните схеми, особено при аналоговите устройства, като могат да бъдат интегрирани и в хибридни или печатни схеми.

Съгласно международните стандарти на IEC резисторът в електронни схеми се означава като празен правоъгълник, а съгласно стандартите на ANSI (прилагани в САЩ и някои други страни) – като зигзаговидна линия. Второто означение може също да се срещне в много стари български или съветски принципни схеми.

Единицата за измерване на електрическото съпротивление на резисторите е ом, наречена в чест на немския физик Георг Симон Ом. Бележи се със символа (гръцката буква) Ω. За удобство се използват и кратни единици на основната – милиоми (mΩ), килооми (kΩ), мегаоми (MΩ) и т.н. Най-широко разпространен начин за отбелязване на стойността на един резистор е чрез цветен код. Тази маркировка се състои от четири или пет (при толеранс по-малък от 5%) цветни точки или ленти, нанесени върху корпуса на резистора. За измерването им се използват омметри или мултиметри („мултицети“).

Реалните резистори се класифицират според два основни критерия:

1) Според тяхната точност (или реципрочната – толеранс). Условно се разделят на точни – с толеранс 0,1%, 0,2%, 0,5%, 1% и 2% и такива за обща употреба – с толеранс 5%, като съществуват и такива с толеранс 10% и 20%, които вече не се произвеждат.

2) Според максималната разсейвана мощност.

Кондензатор

Кондензаторът е пасивен електронен компонент с два извода, предназначен за временно съхранение на електрически заряд чрез енергията на електрическото поле.

Устройството на използваните в практиката кондензатори варира в широки граници, но те почти винаги съдържат поне два проводника, разделени от непроводящ диелектрик. При наличие на потенциална разлика между двата проводника в диелектрика възниква статично електрическо поле, в резултат на което от едната му страна се натрупва положителен, а от другата – отрицателен заряд.

Идеалният кондензатор се характеризира с една постоянна величина, наричана електрически капацитет и измервана във фаради. Капацитетът е отношението на електрическия заряд на всеки от проводниците към потенциалната разлика между тях. При реалните кондензатори през диелектрика протичат известни токове, а големината на електрическото поле е ограничена поради пробивното напрежение на диелектрика. Освен това проводниците предизвикват нежелана индуктивност и съпротивление, които понякога оказват чувствително влияние върху теоретичния модел на работа на кондензаторите.

Кондензаторите намират широко приложение в различни електрически схеми за филтриране на определени честоти, за изглаждане на изходното напрежение на захранващи блокове, Подобряване фактора на мощността в мрежата, в трептящи кръгове използвани в радиотехниката, и за много други цели.

Светодиоди

Светодиодът или светоизлъчващият диод на английски: light-emitting diode, LED е полупроводников диод, който се състои от p-n преход, който излъчва некохерентна светлина в тесен спектър, когато през него протича електрически ток в права посока. Обикновено светодиодът съдържа един или няколко излъчващи светлина кристали, разположени в един корпус с леща, която формира светлинния поток.

Цветът на излъчване на светодиода се определя както от състава на използвания полупроводников материал, така и от легиращите примеси. Цветът често се дава в нанометри (nm), което е дължината на вълната на светлината. Светодиодите не са напълно монохроматични, а излъчват в тесен диапазон на спектъра.

Разработването на светодиоди исторически започва с червени и инфрачервени прибори, направени от галиев арсенид (GaAs). Постиженията в науката за материалите позволяват да се произвеждат прибори в по-късовълновия диапазон на спектъра, излъчващи светлина с различни цветове.

Бутон/Прекъсвач

Електрическия прекъсвач (автоматичен тип) е автоматично управляван ключ, който се използва в бита и съвременното електрическо оборудване, за предпазване на електрическа верига от пренапрежение, ток с високи стойности или късо съединение.

Потенциометърът

Потенциометърът е резистор с 3 извода, с който е възможно при промяната на съпротивлението чрез плъзгащ контакт в електрическата верига, да се променя изходното електрическото напрежение в предварително конструктивно зададени граници. Плъзгащият контакт на този пасивен компонент е единия от изходните електроди и работи като делител на напрежение. Ако се използват само двата му края действа като регулируем резистор (реостат).

Аналогово-цифров преобразувател

Аналогово-цифров преобразувател (съкратено АЦП; на английски: Analog-to-digital converter – съкратено ADC). Електронно устройство което преобразува аналоговият сигнал на входа си в двоичен код на изхода си. Основни параметри са разрядност на изхода и време за преобразуване. От първото зависи точността на устройството (напр. то може да е 8, 10, 12, 16 битово). От второто зависи максималната честота на входния сигнал, който може да се преобразува.

Теоретична характеризация

На теоретично ниво аналогово-цифровите преобразуватели могат да бъдат характеризирани с параметри определящи тяхната точност, линейност, зашуменост, бързина и д.р. В последващата секция са описани някои фундаментални особености на аналогово-цифровите преобразуватели.

Резолюция

Резолюцията на преобразувателите обобщено се свежда до това колко дискретни кода може представи за даден обхват на входния аналогов сигнал. Дискретните нива на изхода на аналогово-цифровите преобразуватели образуват така наречена грешка от квантуване.

Цифров сигнал

Цифров сигнал е аналогов сигнал, който е дискретизиран във времето и квантуван по ниво. За определен период от време цифровият сигнал има краен брой стойности (отчети), които заемат определено ниво от скалата на квантуване, която също е с краен брой стойности. От броя на тези стойности зависи с колко бита ще се кодира всяка стойност на цифровия сигнал. Колкото повече нива има в скалата на квантуване, толкова „по-вярно“ се преобразува аналоговият сигнал в цифров (по-високо качество), но и обемът на информацията е по-голям. Обратно, ако държим на малък обем информация, а качеството на преобразуването не ни е толкова важно, то ще ни трябва скала на квантуване с малък брой нива.

Аналогично е положението и с честотата на дискретизация (гъстотата на отчетите). При повече отчети (по-голяма честота на дискретизация) – по-добро качество, а с по-малко отчети при същия сигнал – по-ниско качество и по-голяма загуба на информация.

На практика обаче всеки сигнал в природата е аналогов. Това се обуславя от преносната среда (кабел, фидер, вълновод, естествена среда (за радиовръзка)), която променя определените параметри на сигнала. Така се наблюдават изменения в големината и формата на импулсите, размивания, закъснение на сигнала, смущения от други сигнали и източници и др.

Аналогов и цифров вход / изход

Arduino може да вкарва и извежда аналогови сигнали, както и цифрови сигнали.

Аналоговият сигнал е такъв, който може да приеме произволен брой стойности, за разлика от цифров сигнал, който има само две стойности: HIGH и LOW. За измерване на стойността на аналоговите сигнали, Arduino има вграден аналогово-цифров преобразувател (ADC). ADC превръща аналоговото напрежение в цифрова стойност. Функцията, която използвате за получаване на стойността на аналогов сигнал, е analogRead (pin). Тази функция преобразува стойността на напрежението на аналогов входен щифт и връща цифровата стойност от 0 до 1023, по отношение на референтната стойност. Референцията е 5V за повечето Arduinos, 7V за Arduino Mini и Nano и 15V за Arduino Mega. Той има един параметър, който е номерът на ПИН.

Arduino няма вграден цифрово-аналогов преобразувател (DAC), но може да модифицира импулс-ширина (PWM) цифров сигнал, за да постигне някои от функциите на аналогов изход. Функцията, използвана за извеждане на PWM сигнал, е analogWrite (пин, стойност). pin е номерът на ПИН, използван за изхода на PWM. стойността е число, пропорционално на работния цикъл на сигнала. Когато стойността = 0, сигналът винаги е изключен. Когато стойността = 255, сигналът винаги е включен. На повечето дъски на Arduino функцията PWM е налична на щифтове 3, 5, 6, 9, 10 и 11. Честотата на PWM сигнала на повечето пинове е приблизително 490 Hz. На Uno и подобни табла щифтове 5 и 6 имат честота приблизително 980 Hz. Пинове 3 и 11 на Leonardo също работят на 980 Hz.

За да настроите аналогов входен сигнал, който варира от 0 до 1023 до PWM изходен сигнал, който варира от 0 - 255, можете да използвате функцията (стойност, от ниска, от висока, до ниска, до висока) . Тази функция има пет параметъра, една е променливата, в която се съхранява аналоговата стойност, докато другите са 0, 1023, 0 и 255 съответно.

Сензор

Сензор (на английски: sensor) или датчик (на руски: датчик) е първичен преобразувател на физични или химични въздействия в удобен за използване електрически сигнал. Тези устройства представляват неизменна част от системите за автоматизирано управление и регулиране. Разликата при сензора спрямо измервателният уред, че сензорът сам по себе си няма зададени предварително фиксирани стойности на отчитаната величина. Сензорът обикновено е част от системата на много измервателни уреди, но той не работи самостоятелно. За да се възприеме информацията от него, първо е необходимо неговият сигнал да бъде усилен и разчетен в системата. Сензори са интегрирани в множество устройства, които използваме всеки ден, като например магнитният сензор в мобилни телефони, сензорът за движение др. Възможните им приложения са практически неограничени.

Чувствителността на сензора е индикатор за големината на промяната на изходния сигнал при дадена промяна на входната величина. Биологичният еквивалент на сензора е рецепторът.

Настройка на периферни устройства

Настройка на периферни устройства

Име Описание Arduino-Switch

цифров изходен шаблон, зададен през изводи 8 до 13.

Вижте използването в #Digital IO. Arduino-Shutter Превключва шаблона на цифровите изходи през щифтове 8 до 13. Изключва всички щифтове, когато затворът е затворен, и възстановява стойността, зададена в състояние на превключване, когато затворът е отворен.

Arduino-Input Reports, както цифровото, така и аналоговото (0-1023) състояние на аналоговите входни щифтове от 0 до 5. Arduino-DAC {1,2} (Не се използва обикновено) Запазено за TLV5618 AOTF периферна. TLV5618 е чип, действащ като ЦАП (цифрово-аналогов преобразувател), който може да бъде добавен като щит към Arduino.

По този начин Arduino може да се използва за управление на аналогови устройства като AOTF или сцени. Вижте #DAC. Описание на свойствата за инициализация Pin Прочита всички 6 входни щифта, когато е зададено на All, в противен случай чете индивидуалния номер на 0-5 Pin. Издърпващ резистор Изберете дали Arduino да използва своя вътрешен 20 kOhm вътрешен издърпващ резистор. Ако не сте сигурни, изберете Включено.

Широчинно импулсна модулация

Модулацията представлява пренасяне на спектъра на сигнала, носител на информация във високочестотната област. Това се постига чрез изменение на параметрите на високочестотен сигнал, наречен носещ, под въздействието на сигнала, носител на информация, наречен модулиращ. Трябва да се отбележи, че понятието „високочестотна област“ е относително и само изразява зависимостта:

f0 >> FМ,

където f0 е честотата на носещия сигнал;

Fм e честотата на модулиращия сигнал.

Основното уравнение за модулиран сигнал е:

S(t)=Akcos(ωct + φ), където: Ak – амплитудата на сигнала; ωct – честотата на сигнала; φ – началната фаза.

За формирането на различни видове модулации е необходимо да се измени някой от параметрите на носещия сигнал. В зависимост от изменяния параметър съществуват следните видове модулации:

Амплитудна модулация (АМ) – при нея се променя амплитудата на носещия сигнал пропорционално на амплитудата на модулиращия сигнал;

Честотна модулация (ЧМ или FM – frequency modulation) – променя се честотата на носещия сигнал пропорционално на амплитудата на модулиращия сигнал;

Импулсна модулация

Фазова модулация – променя се положението (ъгълът) на синусоидата на носещия сигнал пропорционално на амплитудата на модулиращия сигнал. Ъгловата модулация е с два подвида – честотна (ЧМ) и фазова модулация (ФМ). В GSM връзките се използва GMSK.

Делта модулация

Импулсна модулация – при нея носещият сигнал се излъчва под формата на поредица от импулси и се променят параметрите на тази поредица (амплитуда, честота, фаза, широчина) пропорционално на амплитудата на модулиращия сигнал. Различаваме:

а) Амплитудно-импулсна модулация (АИМ)

б) Честотно-импулсна модулация (ЧИМ)

в) Фазово-импулсна модулация (ФИМ)

Квадратурна амплитудна модулация. QAM

г) Широчинно-импулсна модулация (ШИМ)

д) Делта модулация

е) Импулсно-кодова модулация (ИКМ)

ж) Квадратурна амплитудна модулация (QAM)

\* Модулация с непрекъсната вълна – CW

Импулсната модулация се използва основно в системите за телеметрия и управление. Морзовото излъчване НЕ Е импулсна модулация. Делта модулацията се различава съществено от останалите видове импулсна модулация. Тя се използва основно за компресиране на информацията с цел намаляване на обема ѝ.

Все повече в съвременната техника се използва модулацията на цифрови сигнали, която се нарича манипулация. Както при аналоговата модулация, така и при нея параметърът, чието изменение е носител на информацията, може да бъде амплитудата, честотата или фазата.

Изпитна тема № 14

Функционално програмиране План-тезис:

Понятие за „странични ефекти“ в програмирането. Входно/изходни операции от различен тип - конзола, файл, база от данни, мрежа. „Състояние“ на програмата. Глобално и локално състояние. Функции, стойности, цикли и рекурсия. Работа със списъци: „глава“ и „опашка“. Функции от по-висок ред. Анонимни функции. Примерна приложна задача: По време на теоретичния изпит се предоставя непълен/неработещ/некоректен програмен фрагмент на приложната задача. Предоставеният фрагмент да се приведе в работещ вид.

Понятие за „странични ефекти“ в програмирането

В компютърните науки се казва, че операция, функция или израз имат страничен ефект, ако модифицират някаква стойност (стойности) на променливата на състоянието извън нейната локална среда, тоест има наблюдаем ефект, освен връщането на стойност (основния ефект) на призоваващият операцията. Данните за състоянието, актуализирани „извън“ на операцията, могат да се поддържат „вътре“ в обект с състояние или по-широка система с състояние, в рамките на която се извършва операцията. Примерите за странични ефекти включват модифициране на нелокална променлива, модифициране на статична локална променлива, модифициране на изменяем аргумент, предаден чрез препратка, извършване на I / O или извикване на други странични ефекти. При наличие на странични ефекти поведението на програмата може да зависи от историята; т.е. редът на оценяване е от значение. Разбирането и отстраняването на грешки във функцията със странични ефекти изисква познания за контекста и възможните му истории.

Степента на използване на страничните ефекти зависи от парадигмата на програмиране. Императивното програмиране обикновено се използва за създаване на странични ефекти, за актуализиране на състоянието на системата. За разлика от тях декларативното програмиране обикновено се използва за отчитане на състоянието на системата, без странични ефекти.

При функционалното програмиране страничните ефекти се използват рядко. Липсата на странични ефекти улеснява официалните проверки на програмата. Функционални езици като Standard ML, Scheme и Scala не ограничават страничните ефекти, но е обичайно програмистите да ги избягват. Функционалният език Haskell изразява странични ефекти като I / O и други изчисления с състояние, използващи монадични действия.

Програмистите на асемблерен език трябва да са наясно със скритите странични ефекти - инструкции, които модифицират части от състоянието на процесора, които не са споменати в мнемониката на инструкцията. Класически пример за скрит страничен ефект е аритметична инструкция, която имплицитно модифицира кодовете на условията (скрит страничен ефект), докато изрично модифицира регистър (явния ефект). Един потенциален недостатък на набор от инструкции със скрити странични ефекти е, че ако много инструкции имат странични ефекти върху едно състояние, като кодове на условия, тогава логиката, необходима за последователно актуализиране на това състояние, може да се превърне в пречка за изпълнението. Проблемът е особено остър при някои процесори, проектирани с конвейер (от 1990 г.) или с изпълнение извън поръчката. Такъв процесор може да изисква допълнителни схеми за управление за откриване на скрити странични ефекти и спиране на конвейера, ако следващата инструкция зависи от резултатите от тези ефекти.

Входно/изходни операции от различен тип

При изчисленията, вход / изход (I / O или неофициално io или IO) е комуникацията между система за обработка на информация, като компютър, и външния свят, вероятно човек или друга система за обработка на информация. Входовете са сигналите или данните, получени от системата, а изходите са сигналите или данните, изпратени от нея. Терминът може да се използва и като част от действие; да "изпълнява I / O" е да се извърши операция по въвеждане или извеждане.

I / O устройствата са части от хардуера, използвани от човек (или друга система) за комуникация с компютър. Например клавиатурата или компютърната мишка са входно устройство за компютър, докато мониторите и принтерите са изходни устройства. Устройствата за комуникация между компютри, като модеми и мрежови карти, обикновено изпълняват както входни, така и изходни операции.

Определянето на устройство като вход или изход зависи от перспективата. Мишките и клавиатурите извършват физически движения, които човешкият потребител извежда, и ги преобразуват във входни сигнали, които компютърът може да разбере; изходът от тези устройства е входът на компютъра. По същия начин принтерите и мониторите приемат сигнали, които компютрите извеждат като вход, и ги преобразуват в представяне, което човешките потребители могат да разберат. От гледна точка на човешкия потребител, процесът на четене или гледане на тези представяния получава изход; този тип взаимодействие между компютрите и хората се изучава в областта на взаимодействието човек-компютър. Допълнително усложнение е, че устройство, традиционно считано за устройство за въвеждане, например четец на карти, клавиатура, може да приема команди за управление, например, да избира стекер, да показва индикатори на клавиатурата, докато устройство, традиционно считано за изходно устройство, може да предоставя данни за състоянието, напр. , нисък тонер, без хартия, задръстване на хартия.

В компютърната архитектура комбинацията от процесора и основната памет, към която процесорът може да чете или пише директно с помощта на отделни инструкции, се счита за мозък на компютъра. Всяко прехвърляне на информация към или от комбинирания процесор / памет, например чрез четене на данни от дисково устройство, се счита за I / O. Процесорът и неговите поддържащи схеми могат да предоставят картографирани I / O, които се използват при компютърно програмиране на ниско ниво, например при изпълнението на драйвери на устройства, или могат да осигурят достъп до I / O канали. Входно-изходният алгоритъм е проектиран да използва локалността и да се представя ефективно при обмен на данни с вторично устройство за съхранение, като дисково устройство.

Входно-изходните (I / O) системи прехвърлят информация между основната памет на компютъра и външния свят. Входно-изходната система се състои от I / O устройства (периферни устройства), I / O контролни устройства и софтуер за извършване на I / O транзакция (и) чрез последователност от I / O операции. I / O устройствата могат да бъдат класифицирани като серийни, т.е.могат да прехвърлят битови потоци един бит в даден момент или паралелно. Паралелните устройства имат по-широка шина за данни и следователно могат да прехвърлят данни с думи от един или повече байта. Както всяка друга дейност в компютърна система, I / O е съгласувана работа както на хардуер, така и на софтуер. Софтуерът, който се изпълнява за извършване на I / O транзакция за конкретно I / O устройство, се нарича драйвер на устройство. Пример за такава I / O транзакция е четенето на блок данни от диск в паметта. Софтуерът за това е просто последователност от I / O операции (инструкции) за прехвърляне на данни между периферните устройства и основната памет (q.v.) и за да се даде възможност на централния процесор (CPU) да контролира периферните устройства, свързани към него. По този начин I / O операциите са от два класа: операции за контрол и операции за прехвърляне на данни.

Функционално програмиране

В компютърните науки функционално програмиране е парадигма за програмиране – стил за изграждането на структурата и елементите на компютърни програми, който третира като изчислява оценката на математически функции и избягва променящите състоянието непостоянни данни. Това е декларативна парадигма за програмиране, което означава, че програмирането се извършва с изрази. Във функционален код, изходната стойност на функция зависи само от аргументите, които са вложени по време на функцията, така призованата функция е два пъти с една и съща стойност за един аргумент Х и тя ще произведе същия резултат (Х) всеки път. Премахване на нежелани реакции, т.е. промени в състоянието, което не зависи от входовете на функцията, може да направи много по-лесно разбирането и да се предскаже поведението на една програма, което е един от основните мотиви за развитието на функционално програмиране.

Функционално програмиране има своите корени в ламбда пресмятането, което е официална система, разработена през 1930 г., за да разследва изчислимост на дефиниция на функция, приложение функция и рекурсия. Много функционални езици за програмиране могат да бъдат разглеждани като изградени върху ламбда пресмятането. Друга добре позната декларативна парадигма за програмиране е логическото програмиране което, се основава на релацията.

За разлика, императивното програмиране променя състоянието, чрез команди в началния език, най-прост пример за това е задачата. Императивното програмиране наистина има функции, не в математическия смисъл, а в смисъл на подпрограми. Те могат да имат странични ефекти, които могат да променят стойността на програмата. Функции без стойности за връщане могат да доведат до това. Поради това, че те нямат референтна прозрачност, т.е. същия израз в друг език може да доведе до различни стойности по различно време в зависимост от състоянието на изпълняващата програмата.

Функционални програмни езици, особено изцяло функционални такива като Hope и Rex, са използвани повече в академичните среди, отколкото в развитието на търговски софтуер. Въпреки това, видни функционални езици за програмиране като Common Lisp, Scheme, Clojure, Language Wolfram Language, Racket, Erlang, OCaml, Haskell, и F# са били използвани в промишлени и търговски заявления от голямо разнообразие от организации. Функционално програмиране се подкрепя и в някои специфични програмни езици за домейн като R, J, K и (programming language from Kx Systems) Q от Kx Systems (финансов анализ), XQuery / XSLT (XML), и Opal. Широко разпространени специфични програмни езици за домейн като SQL и Lex / Yacc използват някои елементи на функционалното програмиране.

Програмиране на функционален стил може да се осъществи в езици, които не са специално предназначени за функционално програмиране. Например, императивния Perl език за програмиране е бил обект на една книга, описваща как да се прилагат концепции на функционалното програмиране. Това е вярно и за езика PHP. C # 3.0 и Java 8 добавят конструкции за улесняване на функционалния стил. Езикът Julia също предлага функционални възможности за програмиране. Интересен случай е този на Scala – често се пише във функционален стил, но присъствието на странични ефекти и непостоянни състояния го провежда в сивата зона между императивни и функционални езици.

„Състояние“ на програмата

В компютърното програмиране „Състоянието“, „Statement-а“ или „Изявлението“ е синтактична единица на императивен език за програмиране, който изразява някакво действие, което трябва да бъде извършено. Програма, написана на такъв език, се формира от последователност от един или повече твърдения. Изявлението може да има вътрешни компоненти (напр. Изрази).

Много езици за програмиране (напр. Ada, Algol 60, C, Java, Pascal) правят разлика между изрази и дефиниции / декларации. Дефиниция или декларация определя данните, върху които дадена програма трябва да работи, докато декларация определя действията, които трябва да се предприемат с тези данни.

Изявленията, които не могат да съдържат други твърдения, са прости; тези, които могат да съдържат други твърдения, са сложни.

Появата на изявление (и наистина програма) се определя от неговия синтаксис или граматика. Значението на дадено твърдение се определя от неговата семантика.

В каузалната област глобалното състояние е съвкупност от локални състояния, които са едновременно едно с друго. Под едновременно разбираме, че няма две състояния, които да имат причинно-следствена връзка помежду си.

Глобално и локално състояние

Глобалното състояние е набор от локални състояния, които се появяват едновременно. Това определение се основава на физическото време. Използваме фразата „времева област“, за да се позовем на такова определение. Възможно е различно определение на глобално състояние въз основа на „причинно-следствената област“ (свят, основан на причинно-следствената връзка). В каузалната област глобалното състояние е съвкупност от локални състояния, които са едновременно едно с друго. Под едновременно разбираме, че няма две състояния, които да имат причинно-следствена връзка помежду си. Глобално състояние във времевия домейн е и глобално състояние в причинно-следствената област; ако две състояния възникнат едновременно, тогава те не могат да имат никаква причинно-следствена връзка. Обратното обаче не е вярно; две едновременни състояния не могат да възникнат едновременно при дадено изпълнение.

В каузалната област глобалното състояние е съвкупност от локални състояния, които са едновременно едно с друго. Под едновременно разбираме, че няма две състояния, които да имат причинно-следствена връзка помежду си.

add :: Integer -> Integer -> Integer --function declaration

add x y = x + y --function definition

main = do

putStrLn "The addition of the two numbers is:"

print(add 2 5) --calling a function

Функциите

Функциите играят основна роля в Haskell, тъй като това е функционален език за програмиране. Подобно на други езици, Haskell има собствена функционална дефиниция и декларация.

Декларацията за функция се състои от името на функцията и нейния списък с аргументи, заедно с нейния изход.

Определението на функцията е мястото, където всъщност определяте функция.

Нека вземем малък пример за добавяне на функция, за да разберем подробно тази концепция.

The addition of the two numbers is:

7

Тук сме декларирали нашата функция в първия ред, а във втория ред сме написали нашата действителна функция, която ще вземе два аргумента и ще даде един изход от цял тип.

Както повечето други езици, Haskell започва да компилира кода от основния метод. Нашият код ще генерира следния изход –

fact :: Int -> Int

fact 0 = 1

fact n = n \* fact ( n - 1 )

main = do

putStrLn "The factorial of 5 is:"

print (fact 5)

Съвпадащ модел

Pattern Matching е процес на съвпадение на специфичен тип изрази. Това не е нищо друго освен техника за опростяване на вашия код. Тази техника може да бъде внедрена във всеки тип клас Type. If-Else може да се използва като алтернативна опция за съвпадение на шаблони.

Pattern Matching може да се разглежда като вариант на динамичен полиморфизъм, където по време на изпълнение могат да се изпълняват различни методи в зависимост от техния списък с аргументи.

Обърнете внимание на следния блок с кодове. Тук използвахме техниката на съвпадение на образци, за да изчислим факториал на число.

Стойност

Стойността е резултат от оценяване на израз. В Haskell няма разлика между изрази, чийто резултат е функция, и тези, които връщат някои стойности на данните. Името на функцията вече е валиден израз, лесно е да се конструират анонимни функции и т.н.

Цикли

Нека извадим последния ред поотделно:

cycle xs = xs 'where xs' = xs ++ xs '

Сега, нека се опитаме да го намалим:

cycle xs = xs ++ (xs ++ (xs ++ (xs ++ ...)))

Виждате, че се разширява безкрайно. Но имайте предвид, че не така се намаляват изразите в Haskell. Изразите ще бъдат намалени до WHNF, когато се изисква. И така, нека изискаме някои стойности от функцията цикъл:

ghci > take 1 $ cycle [1..]

This is how take function is implemented:

take n \_ | n <= 0 = []

take \_ [] = []

take n (x:xs) = x : take (n-1) xs

Сега, поради съвпадение на модела - стойността n ще бъде оценена първо. Тъй като вече е в нормална форма, не е необходимо да се прави по-нататъшно намаляване и ще се проверява дали е по-малко или равно на нула. Тъй като състоянието е неуспешно, то ще премине към второто условие. Тук ще бъде проверен вторият аргумент, за да се види дали е равен на []. Както обикновено, haskell ще го оцени на WHNF, което ще бъде 1: \_. Тук \_ представлява thunk. Сега целият израз ще бъде намален до 1: вземете 0 \_. Тъй като тази стойност трябва да бъде отпечатана в ghci, целият 1: take 0 \_ ще бъде намален отново. Следвайки отново подобни стъпки, както по-горе, ще получим 1: [], което намалява до [1]. Следователно цикълът [1,2,3] ще бъде намален до WHNF във формата (1: xs) и в крайна сметка ще бъде намален до [1].

Рекурсия

Ще споменем накратко рекурсията в предишната глава. В тази глава ще разгледаме по-отблизо рекурсията, защо е важно за Хаскел и как можем да намерим много кратки и елегантни решения на проблемите, като мислим рекурсивно.

Ако все още не знаете какво е рекурсия, прочетете това изречение. Хаха! Само се шегувам! Рекурсията всъщност е начин за дефиниране на функции, при които функцията се прилага в собствената си дефиниция. Дефинициите в математиката често се дават рекурсивно. Например, последователността на Фибоначи се дефинира рекурсивно. Първо, дефинираме първите две числа на Фибоначи нерекурсивно. Казваме, че F (0) = 0 и F (1) = 1, което означава, че 0-то и 1-во число на Фибоначи са съответно 0 и 1. Тогава казваме, че за всяко друго естествено число това число на Фибоначи е сумата от предишните две числа на Фибоначи. Така че F (n) = F (n-1) + F (n-2). По този начин F (3) е F (2) + F (1), което е (F (1) + F (0)) + F (1). Тъй като сега стигнахме до само нерекурсивно дефинирани числа на Фибоначи, можем спокойно да кажем, че F (3) е 2. Като елемент или два в дефиниция на рекурсия, дефинирани нерекурсивно (като F (0) и F ( 1) тук) се нарича още условие на ръба и е важно, ако искате вашата рекурсивна функция да прекрати. Ако не бяхме дефинирали F (0) и F (1) нерекурсивно, никога нямаше да получите решение нито едно число, защото щяхте да достигнете 0 и след това щяхте да преминете в отрицателни числа. Изведнъж бихте казали, че F (-2000) е F (-2001) + F (-2002) и все още няма да има край!

Работа със списъци: „глава“ и „опашка“

Given any list xs, how do I...?

Basics

* *Get the size of the list.*

length xs

* *Turn a list backwards.*

reverse xs

Finding / searching

* *Get the Nth element out of a list.*

xs !! n

Indexes are zero based, so [1,2,3] !! 0 will result in 1.

(Related: head xs returns the *first* element of the list.)

(Related: last xs returns the *last* element of the list.)

* *Get a list of all elements that match some condition.*

filter my\_test xs

(Returns everything that *passes* the test.)

* *Find the highest/lowest element of a list.*

minimum xs

maximum xs

(Works not just for numbers but *anything* that is a member of the Ord class. In particular, that includes characters and strings.)

Adding

* *Add an element to the start of a list.*

new\_element : xs

* *Add an element to the end of a list.*

xs ++ [new\_element]

* *Insert an element into the middle of a list.*

Generally, you will have to split the list into two smaller lists, put the new element to in the middle, and then join everything back together. For example:

**let** (ys,zs) **=** splitAt n xs **in** ys ++ [new\_element] ++ zs

* *Join two lists together.*

list1 ++ list2

Deleting

* *Delete the first N elements from a list.*

drop n xs

(Related: tail xs removes just *one* element.)

(Related: init xs removes just the *last* element.)

* *Make a new list containing just the first N elements from an existing list.*

take n xs

* *Split a list into two smaller lists (at the Nth position).*

splitAt n xs

(Returns a tuple of two lists.)

* *Delete the just Nth element of a list.*

This is tricky. AFAIK, there is no built-in function that does this. You have to split the list in two, remove the element from one list, and then join them back together, like this:

**let** (ys,zs) **=** splitAt n xs **in** ys ++ (tail zs)

(Related: tail xs removes the *first* element.)

(Related: init xs removes the *last* element. Slow if the list is big.)

* *Delete elements that meet some condition.*

Haskell has a function called filter which will do this for you. Beware though: it should really be named 'select' instead. For example, filter odd xs returns a list of *odd* numbers. That is, it deletes everything that is *not* odd.

Testing various conditions

* *Check if a list is empty.*

null xs

* *Find out whether any list element passes a given test.*

any my\_test xs

* *Check whether all list elements pass a given test.*

all my\_test xs

Modifying the list or its elements

* *Apply a function to all list elements.*

map my\_function xs

* *Apply a function to just some elements of a list.*

Assuming you only want to apply function f to elements for which function p returns true, you can do this:

map (\x **->** **if** p x **then** f x **else** x) xs

* *Convert a list of foos into a list of bars.*

Find or write a function to convert foo into bar, and then apply it to the whole list using map.

* *Number the elements of a list (so I can process each one differently according to its position).*

zip xs [0..]

(For example, zip ['a','b','c'] [0..] gives [('a',0),('b',1),('c',2)].)

* *Apply a list of functions to a single element to get a list of results.*

It's not in the book, but it's easy when you know how:

map ($ my\_element) xs

* *Total up a list of numbers.*

sum xs

(Related: product xs will *multiply* all the elements together instead of adding them.)

* *Sort a list.*

You'll need to import Data.List first, but then you can just do sort xs.

* *Find out if some item is in a list.*

my\_element `elem` xs

Lists and IO

* *Execute a list of IO actions.*

Turn a list of IO actions into one IO action that returns a list of results: sequence xs

Prelude> sequence [putStr "hello ", putStrLn "world"]

hello world

[(),()]

(Note: you might want to use sequence\_ instead, like in the above case, if your actions only return ())

* *Execute an IO action on each element of a list.*

You could map the IO function over your list (resulting in a list of actions) and then perform them using the trick above. But it's much simpler to do this:

mapM my\_action xs

or

mapM\_ my\_action xs

where

mapM f xs **=** sequence (map f xs) and similarly for sequence\_.

Notes about speed

Haskell lists are ordinary single-linked lists. (Look up the term in any book on data structures.) This gives them certain speed properties which are well worth knowing.

Fast operations

The following operations are always 'fast':

* Prepend 1 element (the : operator)
* head (get first element)
* tail (remove first element)

Slower operations

Any function that does something with the Nth element or the first N elements generally gets slower as N increases. The following all slow down as n gets larger:

* xs !! n
* take n xs
* drop n xs
* splitAt n xs

Any function which needs to process the entire list obviously gets slower as the list gets bigger. The following all slow down as the list xs gets larger:

* length xs
* list1 ++ list2 (speed depends *only* on the size of list1)
* last xs
* map my\_fn xs
* filter my\_test xs
* zip my\_fn list1 list2 (speed depends on the *smallest* of the two lists - as does the size of the result list!)
* x `elem` xs
* sum xs
* minimum xs
* maximum xs

Функции от по-висок ред

Функциите на Haskell могат да приемат функции като параметри, а функциите за връщане като възвръщаеми стойности. Функция, която изпълнява някое от тези, се нарича функция от по-висок ред. Функциите от по-висок ред не са само част от преживяването на Haskell, те са почти изживяването на Haskell. Оказва се, че ако искате да дефинирате изчисления, като дефинирате какво е това, вместо да дефинирате стъпки, които променят някакво състояние и може би ги цикли, функциите от по-висок ред са необходими. Те са наистина мощен начин за решаване на проблеми и мислене за програми.

Карирани функции

Всяка функция в Haskell официално приема само един параметър. И така, как е възможно да сме дефинирали и използвали няколко функции, които вземат повече от един параметър досега? Е, това е хитър трик! Всички функции, които досега са приемали няколко параметъра, са карирани функции. Какво означава това? Ще го разберете най-добре на пример. Да вземем нашия добър приятел, функцията max. Изглежда, че отнема два параметъра и връща този, който е по-голям. Правейки max 4 5 първо създава функция, която приема параметър и връща или 4, или този параметър, в зависимост от това кой е по-голям. След това 5 се прилага към тази функция и тази функция дава желания резултат. Това звучи като залък, но всъщност е наистина страхотна концепция. Следните две повиквания са еквивалентни:

1. ghci> max 4 5
2. 5
3. ghci> (max 4) 5
4. 5

Поставянето на интервал между две неща е просто приложение на функцията. Пространството е нещо като оператор и има най-висок приоритет. Нека разгледаме вида на макс. Това е max :: (Ord a) => a -> a -> a. Това може да се запише и като max :: (Ord a) => a -> (a -> a). Това може да се прочете като: max взема a и връща (това е ->) функция, която взема a и връща a. Ето защо типът на връщане и параметрите на функциите са просто разделени със стрелки.

И така, как това е полезно за нас? Просто казано, ако извикаме функция с твърде малко параметри, ние получаваме обратно частично приложена функция, което означава функция, която приема толкова параметри, колкото сме пропуснали. Използването на частично приложение (ако извиквате функции с твърде малко параметри) е чудесен начин да създавате функции в движение, за да можем да ги предадем на друга функция или да ги заредим с някои данни.

Aнонимната функция

Анонимна функция (или т.нар. функционален литерал или ламбда функция) в програмирането е дефинирана функция, която не е обвързана с идентификатор, тоест не е именувана.

Най-често анонимните функции могат да бъдат използвани:[1]

като аргументи, които се прехвърлят към функции от по-висок ред, или

за построяване на резултата на функция от по-висок ред, която трябва да върне функция.

Ако дадена функция се използва само веднъж или ограничен брой пъти, е подходящо да бъде създадена като анонимна, тъй като това я прави синтактично по-лека, отколкото именувана функция. Във функционалните езици и други езици с функции от първи клас анонимните функции се използват навсякъде, където те изпълняват същата роля за вида на функцията, както правят литералите за други типове данни.

Произходът на анонимните функции е свързан с работата на Алонсо Чърч и неговата теория за ламбда смятане, (създадена преди ерата на компютрите), в която всички функции са анонимни.[2] В няколко програмни езика анонимните функции се представят чрез използването на ключовата дума ламбда, а анонимните функции често са посочени като ламбди или ламбда абстракции. Анонимните функции се въвеждат за пръв път в програмния език Lisp през 1958 г. Оттогава непрекъснато се увеличава броя на съвременните езици, които поддържат анонимни функции.

Анонимните функции са форма на вложена функция, в която е позволен достъпа до променливи в обхвата на съдържащата функция. Ето защо анонимните функции трябва да бъдат имплементирани чрез използването на затваряне. За разлика от именуваните вложени функции, те не могат да бъдат рекурсивни без помощта на т.нар. оператор на неподвижната точка (т.нар. анонимна неподвижна точка или анонимна рекурсия).

An anonymous function is a [function](https://wiki.haskell.org/Function) without a name. It is a [Lambda abstraction](https://wiki.haskell.org/Lambda_abstraction) and might look like this: \x **->** x + 1. (That backslash is Haskell's way of expressing a λ and is supposed to look like a Lambda.)

## **Examples**

That is a nameless function which increments its parameter, x. So in [Hugs](https://wiki.haskell.org/Hugs) or [GHCi](https://wiki.haskell.org/GHC/GHCi), I might say:

Prompt> (\x **->** x + 1) 4

5 **::** Integer

Or I could make a nameless function of two parameters, x and y: \x y **->** x + y, which could be applied like so:

Prompt> (\x y **->** x + y) 3 5

8 **::** Integer

You could also name a lambda expression if you wanted to for some reason:

addOne **=** \x **->** x + 1

Of course, there are a variety of better ways to write that in Haskell, but you get the idea.

## **But why bother?**

Sometimes it is more convenient to use a lambda expression rather than giving a function a name. This is often the case when using map and foldl / foldr. So if I wanted to add one to each element of a list, here's one way to do it (without anonymous functions):

addOneList lst **=** map addOne' lst

**where** addOne' x **=** x + 1

But here's another way, where we pass the anonymous function into map rather than any named function.

addOneList' lst **=** map (\x **->** x + 1) lst

For completeness it's worth mentioning that this could be better written using a [section](https://wiki.haskell.org/Section_of_an_infix_operator), in [pointfree](https://wiki.haskell.org/Pointfree) style:

addOneList'' **=** map (+1)

Изпитна тема № 15:

Интернет програмиране План-тезис:

Мрежови протоколи. Комплект протоколи TCP/IP. Сокети. HTTP протокол. GET и POST методи. Формуляри. Семантични елементи и най-често използвани тагове в HTML. Стилизиране на уеб страници чрез CSS. HTTP сървъри и сервиране на статични HTML файлове. Примерна приложна задача: По време на теоретичния изпит се предоставя непълен/неработещ/некоректен програмен фрагмент на приложната задача. Предоставеният фрагмент да се приведе в работещ вид. Условие: Създайте двукомпонентно конзолно приложение за изпращане на съобщения (клиент и сървър), като изпратените съобщения от клиента се получават от сървъра, комуникацията е еднопосочна и се извършва, посредством сокети.

Мрежови протоколи

Мрежов протокол е система от стандартни правила за обмяната на информация между процеси или компютри, които са свързани в една компютърна мрежа, и то по начин, гарантиращ успешната връзка между два или повече комуникационни апарата (крайни устройства). Комуникацията в тази мрежа се базира на множество протоколи с различни функции и се осъществява посредством обмяната на съобщения (пакети). Описаните в протокола правила дефинират каква информация се предоставя в пакетите и в какъв формат, за да се приеме от комуникационните партньори.

Най-често използваните протоколи са протоколите за пренасяне на данни през мрежа. Съществуват 2 модела за класификация на тези протоколи – OSI и TCP/IP.

Типично съдържание на пакет

Пакетите съдържат:

* Адресите (IP адрес или хардуерен) на подателя и получателя
* тип на пакета (например дали се отнася за създаване или прекратяване на връзка)
* дължина на пакета
* сума за проверка на целостта на пакета (на английски: checksum)

Тази информация се поставя преди или след данните в даден пакет. Освен това някои протоколи дефинират определени секвенции от съобщения, които трябва да бъдат изпълнени, за да бъде създадена връзката между комуникационните партньори. Тази информация, наричана оувърхед (на английски: Overhead), натоварва допълнително мрежата, но е необходима за правилното доставяне на пакетите. Съществуват олекотени протоколи с по-малко допълнителна информация, но те са считани за несигурни и се ползват само при определени обстоятелства.

Дефиниции в мрежовите протоколи

Брой на комуникационните партньори: Ако получателят е само един – комуникацията е от тип уникаст (на английски: Unicast), при комуникация между повече партньори – тип мултикаст (на английски: Multicast).

Ако информацията се предава само в едната посока, връзката се нарича симплекс (на английски: Simplex). Ако съобщенията в двете посоки се редуват последователно, връзката е от тип полудуплекс (англ. Halfduplex). Ако информацията се предава едновременно и в двете посоки на комуникация, връзката се нарича пълен дуплекс (на английски: Fullduplex).

Роля на партньорите: ако са с еднакви роли, връзката е симетрична (също peer-to-peer), в противен случай – асиметрична. Типичен пример за асиметрична връзка е модела клиент-сървър (на английски: Client-Server-System), при който един от партньорите приема и обработва запитвания от много клиенти, които са инициаторите на връзката.

Ако при дадено запитване се изисква отговор (например за създаване на връзка), комуникацията се нарича синхронна, в противен случай – асинхронна.

Информацията може да е в пакети или във вид на непрекъснат поток от отделни знаци (на английски: streaming).

При протоколите с повече оуверхед се ползват определени поредици от пакети за дефиниране на начало и край на връзка. Такива протоколи осигуряват транспорта на пакети в определен ред и повторното им пращане, ако не стигнат до получателя. Други протоколи не предлагат такива функции, но имат по-малко оувърхед и комуникацията е по-бърза.

Функции на модерните протоколи

* Сигурно и надеждно осъществяване на връзка между комуникационните партньори.
* Надеждно доставяне на пакетите на желаните получатели.
* Повторно пращане на неполучени пакети.
* Ползване на контролната сума за проверка на целостта на пакета.
* Събирането на пакетите в правилната поредица за изграждането на цялостната информация (като се има предвид, че информацията се разделя при транспортиране на множество малки пакети).
* Възпрепятстване на нежелан достъп и промяна на информацията (кодиране).

Примери

Най-добре познатата сфера на употреба на мрежовите протоколи е интернет:

• отваряне на страници – протоколите HTTP или HTTPS

• изпращане на електронна поща – протокол SMTP

• сваляне на данни – FTP, HTTP или HTTPS

Протоколите се базират отчасти един на друг.

Комплект протоколи TCP/IP

TCP/IP (на английски: Transmission Control Protocol / Internet Protocol) или още семейство Интернет протоколи (на английски: Internet protocol suite) е концептуален модел на семейство от протоколи за комуникация между компютрите, който се използва в Internet и в почти всички други съвременни компютърни мрежи. Този модел се състои от много протоколи, но тъй като ключова роля имат протоколите TCP и IP, името се определя от тях.

Моделът TCP/IP е създаден през 1980 г. заради необходимостта от единен начин за комуникация между компютрите, като по този начин предоставя възможност мрежите да бъдат свързвани помежду си. В модела TCP/IP информацията се пренася под формата на пакети. Всеки пакет се състои от 2 части – заглавна част или още заглавка (от английски: header) и данни. Често е наричан Интернет модел, а в ранните години на Интернет и DoD модел, тъй като разработването на мрежовия модел е финансирано от DARPA, агенция на Министерство на отбраната на САЩ.

## **Протоколни слоеве в TCP/IP**

Моделът TCP/IP се състои от протоколи, които са групирани на базата на предназначението си в 4 „слоя“:

* Приложен слой (Application layer)
* Транспортен слой (Transport layer)
* Мрежови слой (Network/Internet layer)
* Канален слой (Physical layer)

По същество пакетът комуникациони протоколи TCP/IP е практическото приложение на абстракцията OSI модел. Така че можем да съпоставим слоевете от TCP/IP модела със слоевете на OSI модела.

Моделът TCP/IP има следните слоеве:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **OSI** | **TCP/IP** |  |
| 7 | [Application](https://bg.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD_%D1%81%D0%BB%D0%BE%D0%B9_%D0%BD%D0%B0_OSI_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B0) | [Application](https://bg.wikipedia.org/w/index.php?title=%D0%9F%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD_%D1%81%D0%BB%D0%BE%D0%B9_%D0%BD%D0%B0_TCP/IP_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B0&action=edit&redlink=1) | [HTTP](https://bg.wikipedia.org/wiki/HTTP), [FTP](https://bg.wikipedia.org/wiki/FTP) |
| 6 | [Presentation](https://bg.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%B8%D1%82%D0%B5%D0%BB%D0%B5%D0%BD_%D1%81%D0%BB%D0%BE%D0%B9_%D0%BD%D0%B0_OSI_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B0) | [ASCII](https://bg.wikipedia.org/wiki/ASCII), [Unicode](https://bg.wikipedia.org/wiki/Unicode) |
| 5 | [Session](https://bg.wikipedia.org/wiki/%D0%A1%D0%B5%D1%81%D0%B8%D0%B5%D0%BD_%D1%81%D0%BB%D0%BE%D0%B9_%D0%BD%D0%B0_OSI_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B0) | [DNS](https://bg.wikipedia.org/wiki/DNS), [SSL](https://bg.wikipedia.org/wiki/SSL) |
| 4 | [Transport](https://bg.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BF%D0%BE%D1%80%D1%82%D0%B5%D0%BD_%D1%81%D0%BB%D0%BE%D0%B9_%D0%BD%D0%B0_OSI_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B0) | [Transport](https://bg.wikipedia.org/w/index.php?title=%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BF%D0%BE%D1%80%D1%82%D0%B5%D0%BD_%D1%81%D0%BB%D0%BE%D0%B9_%D0%BD%D0%B0_TCP/IP_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B0&action=edit&redlink=1) | [TCP](https://bg.wikipedia.org/wiki/TCP), [UDP](https://bg.wikipedia.org/wiki/UDP), [RTP](https://bg.wikipedia.org/wiki/RTP), [SCTP](https://bg.wikipedia.org/w/index.php?title=SCTP&action=edit&redlink=1) |
| 3 | [Network](https://bg.wikipedia.org/wiki/%D0%9C%D1%80%D0%B5%D0%B6%D0%BE%D0%B2_%D1%81%D0%BB%D0%BE%D0%B9_%D0%B2_OSI_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B0) | [Network](https://bg.wikipedia.org/w/index.php?title=%D0%9C%D1%80%D0%B5%D0%B6%D0%BE%D0%B2_%D1%81%D0%BB%D0%BE%D0%B9_%D0%BD%D0%B0_TCP/IP_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B0&action=edit&redlink=1) | [IP](https://bg.wikipedia.org/wiki/IP), [ICMP](https://bg.wikipedia.org/wiki/ICMP), [IGMP](https://bg.wikipedia.org/wiki/IGMP) |
| 2 | [Data Link](https://bg.wikipedia.org/wiki/%D0%9A%D0%B0%D0%BD%D0%B0%D0%BB%D0%B5%D0%BD_%D1%81%D0%BB%D0%BE%D0%B9_%D0%BD%D0%B0_OSI_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B0) | [Physical](https://bg.wikipedia.org/w/index.php?title=%D0%A4%D0%B8%D0%B7%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8_%D1%81%D0%BB%D0%BE%D0%B9_%D0%BD%D0%B0_TCP/IP_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B0&action=edit&redlink=1) | [Ethernet](https://bg.wikipedia.org/wiki/Ethernet), [Token ring](https://bg.wikipedia.org/wiki/Token_ring) |
| 1 | [Physical](https://bg.wikipedia.org/wiki/%D0%A4%D0%B8%D0%B7%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8_%D1%81%D0%BB%D0%BE%D0%B9_%D0%BD%D0%B0_OSI_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B0) | [T1](https://bg.wikipedia.org/w/index.php?title=T1_(%D1%81%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82)&action=edit&redlink=1), [E1](https://bg.wikipedia.org/w/index.php?title=%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82_E1&action=edit&redlink=1) |
|  |  |  |  |

Канален слой

Той е комбинация между каналния и физичния слой на OSI модела.

Протоколи: ARP, MAC, NDP и др.

Мрежови слой

Чрез мрежовия слой пакетът достига до желаното място.

Протоколи: IP, ICMP, IPsec и др.

Транспортен слой

Транспортният слой отговаря за начина на транспортиране на информацията – обикновено тя трябва да бъде разделена на малки части, за да може да бъде пренесена по мрежата.

Протоколи: TCP, UDP, SCTP и др.

Приложен слой

Приложният слой отговаря за съхраняването на данните, които са специфични за определен приложен софтуер.

Протоколи: HTTP, FTP, DNS, IRC, IMAP, SMTP, POP3, TLS/SSL, SSH, Telnet и много други

Сокети

**сървър** - слуша мрежата и очаква клиент да подаде заявка за свързване.Понятие свързано със програма а не с машина.  
**клиент**- подава заявка към сървъра за установяване на връзка.

### Sockets и ports

**port** - цяло число от тип  "int", софтуерна абстракция, второ ниво на адресиране в една машина за идентифициране  на програмата сървър съответстваща на осигуряваната услуга (service). Портовете  1 - 1024 са резервирани за стандартни услуги.  
Всяка програма сървър е свързана със свой собствен порт.

**socket** - софтуерна абстракция за представяне на двата "терминала" за връзка между машините.  
    **ServerSocket** - свързва програмата сървър със съответния порт, слуша мрежата.  
    **Socket** - използван от програмата клиент за свързване със сървъра и обмен на информация. Използвана от програмата сървър за обмен на информация след установяването на връзка.

**процедури в програмата клиент**:

* създаване на обект *InetAddress*(машината - сървър трябва да бъде в мрежата);
* създаване на обект *Socket* ( програмата сървър трябва да работи - да е свързана с порта);
* създаване на обекти  *InputStream* et *OutputStream* (или съответно *Reader* et *Writer*) ;
* четене от *InputStream*(*Reader*) и писане в *OutputStream*(*Writer*);
* информиране на сървъра за край на връзката и затваряне на *създадения Socket*.

**процедури в програмата сървър**:

* създаване на  обект*ServerSocket* - за да бъде запазен порт и да се слуша мрежата;
* след получаването на заявка от страна на клиента - създаване на *Socket;*
* създаване на *InputStream* и  *OutputStream* (или  съответно *Reader* и *Writer*) ;
* четене от *InputStream*(*Reader*) и писане в *OutputStream*(*Writer*).
* затваряне на създадения *Socket* след отвързването на клиента.

HTTP протокол

Протокол за пренос на хипертекст (на английски: Hypertext Transfer Protocol, HTTP) е мрежов протокол, от приложния слой на OSI модела, за пренос на информация в компютърни мрежи. Създаден като средство за публикуване на HTML страници, протоколът довежда до формирането на Световната мрежа. Разработването на протокола е дело на Уеб консорциума (на английски: World Wide Web Consortium) и IETF (на английски: Internet Engineering Task Force) и завършва с публикуването на серия от документи (заявления за обсъждане) (на английски: Request for Comment, RFC), от които RFC 2616 (от юни 1999) е със статут на стандарт и описва HTTP/1.1.

GET Заявка

Методът HTTP GET изисква представяне на посочения ресурс. Заявките, използващи GET, трябва да се използват само за искане на данни (те не трябва да включват данни).

Забележка: Изпращането на тяло / полезен товар в GET заявка може да накара някои съществуващи внедрения да отхвърлят заявката - макар че не е забранено от спецификацията, семантиката е недефинирана. По-добре е просто да избягвате изпращането на полезни товари в GET заявки.

Заявката има основен номер

Успешният отговор има тяло Да

Сейф Да

Идемпотентно Да

Може да се кешира Да

Позволено в HTML форми Да

Синтаксис

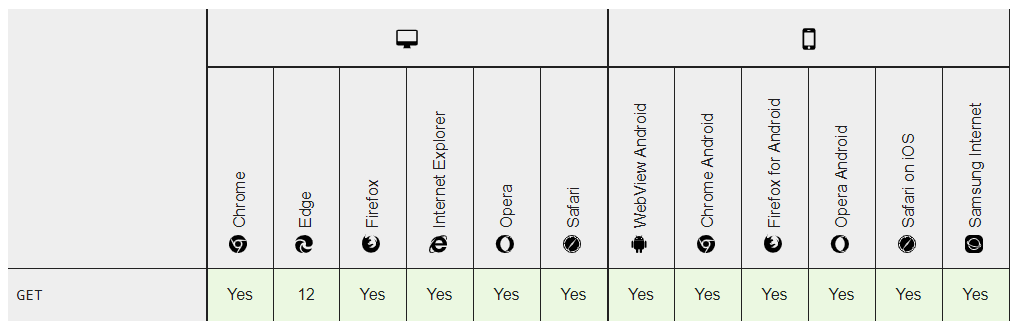
ВЗЕМЕТЕ /index.html

Спецификации

Спецификация Заглавие

RFC 7231, раздел 4.3.1: GET Hypertext Transfer Protocol (HTTP / 1.1): Семантика и съдържание

Съвместимост на браузъра



POST Заявка

Методът HTTP POST изпраща данни към сървъра. Типът на тялото на заявката се посочва от заглавката Content-Type.

Разликата между PUT и POST е, че PUT е идемпотентен: извикването му веднъж или няколко пъти последователно има същия ефект (това не е страничен ефект), където последователният идентичен POST може да има допълнителни ефекти, като предаване на поръчка няколко пъти.

POST заявка обикновено се изпраща чрез HTML форма и води до промяна на сървъра. В този случай типът съдържание се избира чрез поставяне на адекватния низ в атрибута enctype на елемента <form> или атрибута formenctype на елементите <input> или <button>:

application / x-www-form-urlencoded: ключовете и стойностите са кодирани в кортежи ключ-стойност, разделени с '&', с '=' между ключа и стойността. Небуквено-цифровите знаци както в ключовете, така и в стойностите са кодирани в проценти: това е причината този тип да не е подходящ за използване с двоични данни (вместо това използвайте multipart / form-data)

multipart / form-data: всяка стойност се изпраща като блок от данни („част от тялото“), с дефиниран от потребителя агент („граница“), разделящ всяка част. Ключовете са дадени в заглавката Content-Disposition на всяка част.

текст / обикновен

Когато заявката POST се изпраща чрез метод, различен от HTML форма - например чрез XMLHttpRequest - тялото може да приеме всякакъв тип. Както е описано в спецификацията HTTP 1.1, POST е създаден, за да позволи унифициран метод за покриване на следните функции:

Анотация на съществуващите ресурси

Публикуване на съобщение до табло за обяви, дискусионна група, пощенски списък или подобна група статии;

Добавяне на нов потребител чрез модал за регистрация;

Предоставяне на блок данни, като резултат от подаване на формуляр, в процес на обработка на данни;

Разширяване на база данни чрез операция за добавяне.

Заявката има тяло Да

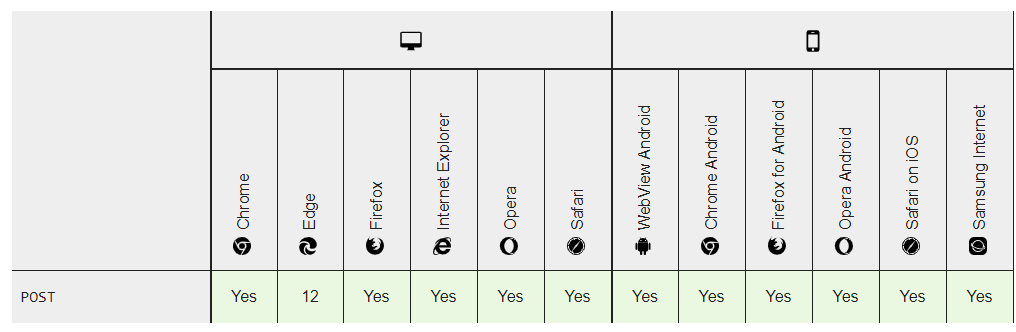
Успешен отговор има тяло Да

Сейф №

Idempotent No.

Кешируемо само ако е включена информация за свежест

Позволено в HTML форми Да



GET и POST методи

Какво е HTTP?

Протоколът за трансфер на хипертекст (HTTP) е предназначен да даде възможност за комуникация между клиенти и сървъри.

HTTP работи като протокол за заявка-отговор между клиент и сървър.

Пример: Клиент (браузър) изпраща HTTP заявка към сървъра; след това сървърът връща отговор на клиента. Отговорът съдържа информация за състоянието на заявката и може също да съдържа исканото съдържание.

HTTP методи

ВЗЕМЕТЕ

ПОСТ

СЛАГАМ

ГЛАВА

ИЗТРИЙ

КРЪПКА

НАСТРОИКИ

Двата най-често срещани HTTP метода са: GET и POST.

Методът GET

GET се използва за искане на данни от определен ресурс.

GET е един от най-често срещаните HTTP методи.

Имайте предвид, че низът на заявката (двойки име / стойност) се изпраща в URL адреса на GET заявка:

/test/demo\_form.php?name1=value1&name2=value2

Някои други бележки относно заявките за GET:

GET заявките могат да бъдат кеширани

GET заявките остават в историята на браузъра

GET заявките могат да бъдат маркирани

GET заявките никога не трябва да се използват при работа с чувствителни данни

Заявките за GET имат ограничения за дължина

GET заявките се използват само за искане на данни (не се променят)

Методът POST

POST се използва за изпращане на данни към сървър за създаване / актуализиране на ресурс.

Данните, изпратени до сървъра с POST, се съхраняват в тялото на заявката на HTTP заявката:

POST /test/demo\_form.php HTTP / 1.1

Водещ: w3schools.com

name1 = value1 & name2 = value2

POST е един от най-често срещаните HTTP методи.

Някои други бележки относно POST заявките:

POST заявките никога не се кешират

POST заявките не остават в историята на браузъра

POST заявките не могат да бъдат маркирани

POST заявките нямат ограничения за дължината на данните

Методът PUT

PUT се използва за изпращане на данни към сървър за създаване / актуализиране на ресурс.

Разликата между POST и PUT е, че заявките за PUT са идемпотентни. Това означава, че извикването на една и съща заявка PUT няколко пъти винаги ще доведе до един и същ резултат. За разлика от това, извикването на POST заявка многократно има странични ефекти от създаването на един и същ ресурс няколко пъти.

Методът HEAD

HEAD е почти идентичен с GET, но без тялото на отговора.

С други думи, ако GET / потребители върне списък с потребители, тогава HEAD / потребители ще направят същата заявка, но няма да върнат списъка с потребители.

HEAD заявките са полезни за проверка на това, което GET заявка ще върне, преди действително да направите GET заявка - като преди да изтеглите голям файл или тяло на отговор.

Методът DELETE

Методът DELETE изтрива посочения ресурс.

Методът ОПЦИИ

Методът OPTIONS описва комуникационните опции за целевия ресурс.

Сравнете GET срещу POST

Следващата таблица сравнява двата HTTP метода: GET и POST.

Формуляри

Следващият откъс е от д-р Иван Миснер. и книгата на Брайън Хилиард Networking Like a Pro. Купете го сега от Amazon | Barnes & Noble | IndieBound

Бизнес професионалистите, които нямат много свободно време, често ни питат кои мрежови групи осигуряват най-големия удар за парите си. Има пет основни типа и това, което работи най-добре, зависи от бизнеса, в който се занимават, и перспективите, които искат да срещнат.

Ето кратко описание на най-познатите типове.

Свързани: Поправете „Прекъсването на мрежата“ и продължете работата си по мечта

1. Случайни контактни мрежи

Това са общи бизнес групи, които позволяват на много хора от различни припокриващи се професии. Тези групи обикновено се срещат ежемесечно и често държат миксери, където всички се смесват неофициално. Те могат също така да провеждат срещи, на които гост-лектори да представят важни бизнес теми или да обсъждат въпроси, свързани със законодателството, общностните въпроси или местните бизнес програми.

Най-добрите примери за тези групи са хилядите търговски камари, действащи в Северна Америка и другаде по света. Те предлагат на участниците възможност да осъществят ценни контакти с много други бизнесмени в общността. Посещавайки камерни събития, можете да установите първоначални контакти, които ще бъдат ценни в други аспекти на развитието на вашия бизнес с препоръки.

Но тъй като организациите за случайни контакти не са пригодени основно за да ви помогнат да получите препоръки, трябва да положите усилия, за да ги накарате да работят. Например, можете да се включите доброволно за камарен посланик, позиция, която изисква малко време, но осигурява голяма експозиция. Заседанието в комитети ви помага да опознаете по-добре членовете. Най-вече трябва да присъствате редовно на събития, за да можете да се възползвате от всяка възможност за укрепване на отношенията, които създавате.

2. Силни контактни мрежи

Организациите, чиято цел е основно да помогнат на членовете да обменят бизнес препоръки, са известни като силни групи за препоръки за контакт. Някои от тези групи се срещат седмично, обикновено на обяд или закуска. Повечето от тях ограничават членството до един член за професия или специалност.

Силните мрежи за контакти осигуряват силно фокусирани възможности за вас и вашите сътрудници да започнете да разработвате вашите маркетингови кампании за препоръки. Няма да срещнете стотици бизнесмени в този тип група, но всички членове ще носят вашите визитки наоколо, където и да отидат. Нетният резултат е като да имате до 50 продавачи, които работят за вас! С програма като тази ще установите мощни дългосрочни отношения, които ще се окажат безценни.

Ако обмисляте група за силни контакти, ще искате да имате предвид няколко неща:

Трябва да имате график, който ви позволява да присъствате на всички или почти на всички срещи. Редовното присъствие е от жизненоважно значение за създаването на връзка с други членове на групата и за опознаването на техния бизнес.

Трябва да се чувствате комфортно, като отидете на събитие в мрежа и търсите перспективи, които могат да помогнат на други членове на вашата група. Една добра мрежа от силни контакти обикновено проследява обема на бизнеса, който се провежда. Ако не „дърпате тежестта си“, ще бъдете помолени да напуснете или рефералите ще спрат да ви идват по пътя.

3. Клубове за обществени услуги

За разлика от по-ориентираните към бизнеса групи, групите услуги не са създадени предимно за мрежи за препоръчани потребители; техните дейности са насочени към обслужване на общността. Въпреки това, като отделяте време и усилия за граждански каузи, вие създавате трайни взаимоотношения, които разширяват и задълбочават вашите лични и бизнес мрежи. Ако влезете не за да се възползвате, а за да допринесете, натрупаният от вас социален капитал в крайна сметка ще ви възнагради по други начини и от други посоки - бизнес сред тях.

4. Професионални сдружения

Членовете на професионални асоциации обикновено са от един специфичен вид индустрия, като банкиране, архитектура, персонал, счетоводство или здравеопазване. Основната цел на професионалната асоциация е да обменя информация и идеи.

Вашата цел при включването в такива мрежи е да се присъедините към групи, съдържащи вашите потенциални клиенти или целеви пазари. Една проста процедура за насочване на ключови групи е да попитате най-добрите си клиенти или клиенти към кои групи принадлежат.

Много групи ограничават членството си до онези, които имат конкретни браншови данни, а продавачите не са добре дошли. Въпреки това, за да генерират повече доходи или да дадат на пълноправните си членове добре закръглени списъци с потенциални доставчици, все повече асоциации са създали асоциирана категория членове, чиито членове не са активни в бизнеса или професията, за която е създадена групата.

В този тип мрежи ви препоръчваме да се откроите, като намерите начини да помогнете, без да продавате на членове. Като пример, ако сте били консултант в социалните медии и сте се присъединили към асоциация на професионални бизнес треньори, вместо да се опитвате да ги „продавате“ на своите услуги, какво ще кажете за доброволческа дейност, за да управлявате социалните платформи на асоциацията? Поемането на отговорност за техните страници във Facebook и LinkedIn би било чудесна позиция

Семантични елементи

Семантичен HTML не е нова версия или различна версия на HTML. Семантичния HTML е най-обикновен HTML. Семантичен го прави начина, по който той бива използван. Семантичен буквално означава „значещ“. Като цяло HTML не е нещо предназначено за крайния потребител пред екрана, а за разработчиците и машините. За да бъде „значещ“ за тях, той трябва да е максимално добре структуриран и да описва максимално точно съдържанието което представя. Той конкретизира вида на съдържанието и указва неговата йерархия. Един сайт може да бъде изграден единствено със <span></span> елементи и за крайния потребител това да не е проблем, но как ще се ориентира един разработчик, който реши да редактира сайта. Или когато така наречените уеб паяци (web crawler) минат от там, за тях всичко ще е еднакво и те няма да индексират съответния сайт. Поради тези причини семантичния HTML е много важен за SEO и лесното преработване на сайтове.

HTML съдържа в себе си концептуално семантичния модел, т.к. в един HTML документ, авторът може, наред с други неща, да започне със заглавие; да обработва параграфи, абзаци и под-теми; да добави акцент в текста; изображения; връзки към други страници; както и да използва различни видове списъци. Семантичният HTML е концепцията да се използват мета данни и подходящи тагове, така че да се обозначат различните формати и данни в HTML документа.

Цели на семантичения HTML

**Семантичния HTML** има за цел да развие и модернизира настоящите уеб стандарти, като дава възможност на потребителите да намерят, споделят, и комбинират дадената информация по-лесно. Това предоставя възможност хора, с помощта на тази технология да могат да използват Интернет за изпълнение на задачи като намиране на Български или чуждоезичен превод за известна книга, резервиране на книга от библиотеката, и търсене на най-ниската цена за даден продукт. Все пак компютрите, не могат да осъществят всички тези задачи без човешка насока, тъй като уеб страниците са предназначени да се четат от хора, а не от машини. Семантичният уеб е визия на информацията, която може лесно да се тълкува от машини, така че компютрите да могат да извършват повечето от стандартната работа за търсене, намиране, съчетаване на съществените данни в Интернет пространството.

Възможности и похвати на HTML

Файловете в един стандартен компютър могат да бъдат условно разделени на документи, позволяващи да бъдат четени от хора и електронно кодирани документи. Файлове, като електронната поща, доклади и брошури могат да бъдат четени от хората, докато данните, като календари, плейлисти и електронни таблици са представени използвайки приложна програма, която позволява да се гледа, търси и комбинира по различни начини в дадената апликация.

В момента, World Wide Web се базира основно на документи, написани на Hypertext Markup Language HTML, като конвенция, която се използва за кодиране на тялото на текста заедно с мултимедийни обекти, например изображения и интерактивни форми. Таговете на метаданните предоставят метод, чрез който компютрите да категоризира съдържанието на уеб страниците, например:

<**meta** name="keywords" content="semantic html, computer science, computer programming" />

<**meta** name="description" content="Courses of software engineering" />

<**meta** name="author" content="Ivo Nikolov" />

С HTML и на база на допълнително приложение (примерно уеб браузър), което да го визуализира, могат да бъдат направени разнообразни уеб страници, които дават възможност за запазване на информация за различни продукти, както и да бъде осъществявана тяхната покупко-продажба. Семантичният HTML по-скоро прилага традиционните HTML практики за маркиране, отколкото да специфицира оформлението на страниците директно.

Microformats (микроформатите) обогатяват синтаксиса на HTML, като създават машинно четимо семантично обозначение на различни обекти, като хора, коли, животни и много други.

Доброто семантично оформление в HTML подобрява достъпността на уеб документите, например: когато дадена апликация за четене на текст, или за работа с мултимедийни файлове, може правилно да установи структурата на даден документ, потребителя няма да губи време с ненужно препрочитане на повтаряща се и ненужна информация.[[2]](https://bg.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D0%BD%D1%82%D0%B8%D1%87%D0%B5%D0%BD_HTML#cite_note-2)

Семантични елементи в HTML5

* Елемент <header> – представлява заглавие на елемент или група от елементи. Тага <header> позволява използването на няколко заглавия в един документ за разлика от възможността която предоставя HTML4. Пример:

<**article**>

<**header**>

<**h1**>HTML</**h1**>

<**p**><**time** pubdate datetime="1990"></**time**></**p**>

</**header**>

<**p**>HTML5 is the fifth revision of the HTML standard (created in 1990 and standardized as HTML 4 as of 1997.....</**p**>

</**article**>

* Елемент <nav> – дефинира навигацията на сайта. Той е предназначен за големи блокове от навигационни връзки. Не всички връзки трябва да са в <nav> елемента, а само главното навигационно меню. Пример:

<**nav**>

<**a** href="/html/">HTML</**a**> |

<**a** href="/css/">CSS</**a**> |

<**a** href="/js/">JavaScript</**a**> |

<**a** href="/jquery/">jQuery</**a**>

</**nav**>

* Елемент <article> – представлява предмет/статия. Съдържанието в <article> трябва да има смисъл само по себе си и да може да се разпространява независимо и отделно от останалата част на сайта. Използва се най-често за публикация във форум; публикация в блог; новини; коментари и др. Пример:

<**article**>

<**h1**>HTML</**h1**>

<**p**>HTML5 is the fifth revision of the HTML standard (created in 1990 and standardized as HTML 4 as of 1997.....</**p**>

</**article**>

* Елемент </section/> е тематично групирано съдържание, обикновено със заглавие. Този елемент позволява влагане на нова сегмантична структура във вече съществуващ елемент. Пример:

<**section**>

<**h1**>WWF</**h1**>

<**p**>The World Wide Fund for Nature (WWF) is....</**p**>

</**section**>

* Елемент <aside> – той представя съдържание което е странично от основното съдържание на страницата. Съдържанието в <aside> се счита за неважно и може да бъде пропуснато, ако сайта се възпроизвежда на устройство с малък екран като телефон. Пример:

<**p**>My family and I visited The Epcot center this summer.</**p**>

<**aside**>

<**h4**>Epcot Center</**h4**>

<**p**>The Epcot Center is a theme park in Disney World, Florida.</**p**>

</**aside**>

* Елемент <footer> – може да съдържа информация за автора на страницата; навигация на сайта; авторските права върху използваните материали и връзки към други (подобни) публикации. Пример:

<**footer**>

<**p**>Posted by: John Williams</**p**>

<**p**><**time** pubdate datetime="2013-05-05"></**time**></**p**>

</**footer**>

Най-често използвани тагове

**Основни HTML тагове**

**<b> таг (препоръчва се да използвате css font-weight: bold)**

Удебелява текста.

Това е нормален текст**&nbsp;**– <**b**>а това е удебелен</**b**>.

Това е нормален текст – **а това е удебелен**.

**<i> таг (по-добрия вариант е да се използва CSS font-style: italic)**

Задава курсив/наклон на текста.

Той кръсти своята кола <**i**>Светкавицата</**i**>, защото беше много бърза.

Той кръсти своята кола *Светкавицата*, защото беше много бърза.

**<u> таг (препоръчва се да не го използвате – използвайте CSS text-decoration: underline)**

Подчертава текста.

Това е <**u**>подчертан текст</**u**>.

Това е подчертан текст.

**<strong> таг**

Указва важен текст.

<**strong**>Важен текст</**strong**>

**Важен текст**

**<sub> таг**

Дефинира текст под черта.

Текстът съдържа <**sub**>subscript</**sub**> текст.

Текстът съдържа subscript текст.

**<sup> таг**

Дефинира текст над черта.

Текстът съдържа <**sup**>superscript</**sup**> текст.

Текстът съдържа superscript текст.

**<del> таг**

Указва изтрит текст.

Моят любим цвят е <**del**>червен</**del**> син!

Моят любим цвят е ~~червен~~ син!

**<br> таг**

Указва нов ред. Няма таг за край.

Този текст съдържа <**br**> нов ред.

Този текст съдържа  
нов ред.

**<blockquote> таг**

Указва част от текста, който е цитат.

<**blockquote** cite="http://www.worldwildlife.org/who/index.html">

Когато често се срещаме с дадени хора, те стават част от живота ни.

И като станат част от живота ни, започват да се опитват да го променят.

И се сърдят, когато не правим това, което те изискват от нас.

Понеже всеки си мисли, че знае как другият трябва да живее живота си,

но всъщност никой не знае как трябва да живее своя собствен.

</**blockquote**>

Когато често се срещаме с дадени хора, те стават част от живота ни.

И като станат част от живота ни, започват да се опитват да го променят.

И се сърдят, когато не правим това, което те изискват от нас.

Понеже всеки си мисли, че знае как другият трябва да живее живота си, но всъщност никой не знае как трябва да живее своя собствен.

**<a> таг**

Указва линк към друга страница. Най-важният атрибут на този таг е *href*. Той посочва URL адреса, към който сочи линка:

<**a** href="https://bg.wikipedia.org">Уикипедия на Български</**a**>

Уикипедия на Български

**<img> таг**

Дефинира картинка в HTML страницата. Има два задължителни атрибута: *src* и *alt*. Атрибутът *src* указва URL адреса на картинката, *alt* – указва алтернативен текст на картинката, а *hight* и *width* – указват съответно височината и ширината на картинката в пиксели.

<**img** src="smiley.gif" alt="Smiley face" height="42" width="42">

Стилизиране с HTML

HTML (съкращение от термина на английски: HyperText Markup Language, произнасяно най-често като „ейч-ти-ем-ел“, в превод „език за маркиране на хипертекст“) е основният маркиращ език за описание и дизайн на уеб страници. HTML е стандарт в Интернет, а правилата се определят от международния консорциум W3C. Текущата версия на стандарта е HTML 5.0 (от 28 октомври 2014 г.)

Описанието на документа става чрез специални елементи, наречени HTML елементи или маркери, които се състоят от етикети или тагове (HTML tags) и ъглови скоби (като например елемента <html>). HTML елементите са основната градивна единица на уеб страниците. Чрез тях се оформят отделните части от текста на една уеб страница, като заглавия, цитати, раздели, хипертекстови препратки и т.н. Най-често HTML елементите са групирани по двойки <h1> и </h1>.

В повечето случаи HTML кодът е написан в текстови файлове и се хоства на сървъри, свързани към Интернет. Тези файлове съдържат текстово съдържание с маркери – инструкции за браузъра за това как да се показва текстът. Например <маркер> Някакъв текст. </край на маркера>. Предназначението на уеб браузърите е да могат да прочетат HTML документите и да ги превърнат в уеб страници. Браузърите не показват HTML таговете, а ги използват, за да интерпретират съдържанието на страницата.

**Основни елементи**

**<html> елемент**

Указва на браузъра, че това е HTML документ. Отбелязва началото и края на документа и съдържа всички други негови елементи (с изключение на <!DOCTYPE> елемента).

<**html**>

...

</**html**>

**<head> елемент**

Съдържа заглавието на документа, и може да съдържа стилове, скриптове, енкодинг и т.н.

<**html**>

<**head**>

<**title**>Заглавие на документа</**title**>

</**head**>

...

</**html**>

**<body> елемент**

Съдържа форматиране видимо за потребителя – текст, хиперлинк, картинки, таблици, бутони, параграфи и т.н.

<**html**>

<**head**>

<**title**>Заглавие на документа</**title**>

</**head**>

<**body**>

Съдържание на документа...

</**body**>

</**html**>

**<!DOCTYPE> елемент**

Декларира се първи, още преди <html> тага. Валидира документа. <!DOCTYPE> не е HTML таг. Той е инструкция за уеб браузъра – указва HTML версията, на която е написана страницата:

<!DOCTYPE HTML PUBLIC „-//W3C//DTD HTML 4.01//EN“ „http://www.w3.org/TR/html4/strict.dtd“>

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>Заглавие на документа</**title**>

</**head**>

<**body**>

<**h1**>Моето първо заглавие.</**h1**>

Съдържание на документа......

</**body**>

</**html**>

CSS

Каскадни таблици със стилове (CSS) е език на таблицата със стилове, използван за описване на представянето на документ, написан на език за маркиране като HTML. CSS е крайъгълна технология на World Wide Web, заедно с HTML и JavaScript.

CSS е създаден, за да даде възможност за разделяне на презентацията и съдържанието, включително оформление, цветове и шрифтове. Това разделяне може да подобри достъпността на съдържанието, да осигури повече гъвкавост и контрол при специфицирането на характеристиките на презентацията, да даде възможност на множество уеб страници да споделят форматирането, като посочи съответния CSS в отделен .css файл, което намалява сложността и повторяемостта на структурното съдържание, както и позволява .css файлът, който ще се кешира, за да се подобри скоростта на зареждане на страницата между страниците, които споделят файла и неговото форматиране.

Разделянето на форматирането и съдържанието също прави възможно представянето на една и съща страница за маркиране в различни стилове за различни методи за изобразяване, като например на екрана, в печат, с глас (чрез браузър, базиран на реч или четец на екран), и на Брайлова основа тактилни устройства. CSS също има правила за алтернативно форматиране, ако съдържанието е достъпно на мобилно устройство.

Каскадното име идва от посочената приоритетна схема, за да се определи кое правило на стила се прилага, ако повече от едно правило съответства на определен елемент. Тази каскадна приоритетна схема е предвидима.

Спецификациите на CSS се поддържат от консорциума на World Wide Web (W3C). Текстът / css от интернет носител (тип MIME) е регистриран за използване с CSS от RFC 2318 (март 1998 г.). W3C управлява безплатна услуга за проверка на CSS за CSS документи.

В допълнение към HTML, други езици за маркиране поддържат използването на CSS, включително XHTML, обикновен XML, SVG и XUL.

Използвайте

Преди CSS, почти всички презентационни атрибути на HTML документи се съдържаха в HTML маркирането. Всички цветове на шрифта, стилове на фона, подравняване на елементи, граници и размери трябва да бъдат изрично описани, често многократно, в HTML. CSS позволява на авторите да преместят голяма част от тази информация в друг файл, таблицата със стилове, което води до значително по-опростен HTML.

Например заглавия (елементи h1), подзаглавия (h2), подзаглавия (h3) и т.н. се дефинират структурно с помощта на HTML. При печат и на екрана изборът на шрифт, размер, цвят и акцент за тези елементи е презентационен.

Преди CSS, авторите на документи, които искаха да присвоят такива типографски характеристики, да речем, всички h2 заглавия трябваше да повтарят HTML презентационно маркиране за всяко появяване на този тип заглавия. Това направи документите по-сложни, по-големи и по-податливи на грешки и трудни за поддръжка. CSS позволява отделянето на презентацията от структурата. CSS може да дефинира цвят, шрифт, подравняване на текст, размер, граници, интервали, оформление и много други типографски характеристики и може да го направи независимо за екранни и печатни изгледи. CSS също дефинира невизуални стилове, като скорост на четене и акцент за четене на звуков текст. W3C вече оттегли използването на всички презентационни HTML маркировки.

Например, под HTML преди CSS, заглавният елемент, дефиниран с червен текст, ще бъде написан като:

<**h1**><**font** color="red">Chapter 1.</**font**></**h1**>

Using CSS, the same element can be coded using style properties instead of HTML presentational attributes:

<**h1** style="color: red;">Chapter 1.</**h1**>

The advantages of this may not be immediately clear but the power of CSS becomes more apparent when the style properties are placed in an internal style element or, even better, an external CSS file. For example, suppose the document contains the style element:

<**style**>

**h1** {

**color**: **red**;

}

</**style**>

All h1 elements in the document will then automatically become red without requiring any explicit code. If the author later wanted to make h1 elements blue instead, this could be done by changing the style element to:

<**style**>

**h1** {

**color**: **blue**;

}

</**style**>

rather than by laboriously going through the document and changing the color for each individual h1 element.

The styles can also be placed in an external CSS file, as described below, and loaded using syntax similar to:

<**link** href="path/to/file.css" rel="stylesheet" type="text/css">

This further decouples the styling from the HTML document and makes it possible to restyle multiple documents by simply editing a shared external CSS file.

HTTP сървър / Уеб сървър

Уеб сървър (на английски: web server, т.е. „звено, обслужващо Мрежата“) е приложна програма, сървър, която позволява на даден компютър да предоставя информация на други компютри, под формата на страници с хипертекст.

Уеб сървър предоставя информация на клиент под формата на HTML документ. За пренасянето на информацията се използва протокола HTTP. Често уеб сървър се използва за предоставяне на информация и в други формати – най-често това са изображения във форматите PNG, JPEG и GIF; XML документи и др.

Всеки един потребител, стига да има компютърна техника, мрежова техника и интернет доставчикът му позволява, може да стартира уеб сървър.

Типично съдържание на пакет

Пакетите съдържат:

* Адресите (IP адрес или хардуерен) на подателя и получателя
* тип на пакета (например дали се отнася за създаване или прекратяване на връзка)
* дължина на пакета
* сума за проверка на целостта на пакета (на английски: checksum)
* Тази информация се поставя преди или след данните в даден пакет. Освен това някои протоколи дефинират определени секвенции от съобщения, които трябва да бъдат изпълнени, за да бъде създадена връзката между комуникационните партньори. Тази информация, наричана оувърхед (на английски: *Overhead*), натоварва допълнително мрежата, но е необходима за правилното доставяне на пакетите. Съществуват олекотени протоколи с по-малко допълнителна информация, но те са считани за несигурни и се ползват само при определени обстоятелства.

Дефиниции в мрежовите протоколи

* Брой на комуникационните партньори: Ако получателят е само един – комуникацията е от тип уникаст (на английски: *Unicast*), при комуникация между повече партньори – тип мултикаст (на английски: *Multicast*).
* Ако информацията се предава само в едната посока, връзката се нарича симплекс (на английски: *Simplex*). Ако съобщенията в двете посоки се редуват последователно, връзката е от тип полудуплекс (англ. *Halfduplex*). Ако информацията се предава едновременно и в двете посоки на комуникация, връзката се нарича пълен дуплекс (на английски: *Fullduplex*).
* Роля на партньорите: ако са с еднакви роли, връзката е симетрична (също *peer-to-peer)*, в противен случай – асиметрична. Типичен пример за асиметрична връзка е модела клиент-сървър (на английски: Client *-Server-System*), при който един от партньорите приема и обработва запитвания от много клиенти, които са инициаторите на връзката.
* Ако при дадено запитване се изисква отговор (например за създаване на връзка), комуникацията се нарича синхронна, в противен случай – асинхронна.
* Информацията може да е в пакети или във вид на непрекъснат поток от отделни знаци (на английски: *streaming*).
* При протоколите с повече оуверхед се ползват определени поредици от пакети за дефиниране на начало и край на връзка. Такива протоколи осигуряват транспорта на пакети в определен ред и повторното им пращане, ако не стигнат до получателя. Други протоколи не предлагат такива функции, но имат по-малко оувърхед и комуникацията е по-бърза.

Функции на модерните протоколи

Сигурно и надеждно осъществяване на връзка между комуникационните партньори.

* Надеждно доставяне на пакетите на желаните получатели.
* Повторно пращане на неполучени пакети.
* Ползване на контролната сума за проверка на целостта на пакета.
* Събирането на пакетите в правилната поредица за изграждането на цялостната информация (като се има предвид, че информацията се разделя при транспортиране на множество малки пакети).
* Възпрепятстване на нежелан достъп и промяна на информацията (кодиране).

Примери

Най-добре познатата сфера на употреба на мрежовите протоколи е интернет:

* отваряне на страници – протоколите HTTP или HTTPS
* изпращане на електронна поща – протокол SMTP
* сваляне на данни – FTP, HTTP или HTTPS
* Протоколите се базират отчасти един на друг.

Сервиране на статични HTML файлове

Обслужване на статични HTML файлове локално

Като цяло можем да щракнем с десния бутон върху HTML файла и да кажем „отвори в браузъра“ и той ще отвори файла във вашия локален браузър (Chrome, Firefox, Edge, Safari), но всъщност не е точно това, което винаги искаме. по-скоро искаме нещо като обикновен уеб сървър, който да обслужва файловете от директория.

"http-сървърът" може да направи точно това по лесен начин.

npm инсталирайте http-сървър -g

След инсталирането можем да отидем в нашата статична директория на сайта и да стартираме

http-сървър

По подразбиране ще обслужва директорията на localhost 8080.

Също така можем да предадем опции за това.

ES2017 async / await с извличане в Redux

Когато трябва да използваме API на трети страни или REST API в нашето уеб приложение, трябва да изчакаме отговора, идващ от сървъра.

След това при успешното обратно извикване използваме тези данни и правим други операции. И ако имаме повече REST повиквания, всички знаем, че в крайна сметка това може да бъде ад за обратно повикване

ES2017 async / await

Но ако използваме синтаксис ES2017 async / await, можем да напишем кода си възможно най-изчистен с минимални редове код.

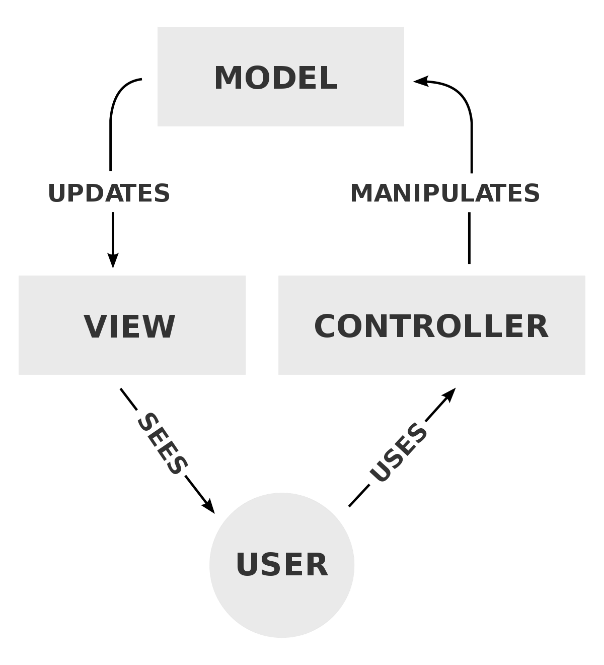
Основният пример за async / await е като по-долу-

|  |  |
| --- | --- |
|  | function resolveAfter2Seconds() { |
|  | return new Promise(resolve => { |
|  | setTimeout(() => { |
|  | resolve('resolved'); |
|  | }, 2000); |
|  | }); |
|  | } |
|  |  |
|  | async function asyncCall() { |
|  | console.log('calling'); |
|  | var result = await resolveAfter2Seconds(); |
|  | console.log(result); |
|  | // expected output: "resolved" |
|  | } |
|  |  |
|  | asyncCall(); |

Изпитна тема № 16:

Интернет програмиране План-тезис:

Уеб MVC платформа, архитектура и компоненти. Свързване на HTTP сървър с backend език за програмиране. Комуникация с база от данни. Шаблонни езици от страна на сървъра. Създаване на MVC уеб приложения. Управление на състоянието в уеб приложенията. Автентикация и оторизация. Сигурност на уеб приложенията и добре известни атаки. Примерна приложна задача:



Уеб MVC платформа

Използване на MVC в уеб приложения

Въпреки че първоначално е разработен за настолни компютри, MVC е широко приет като дизайн за приложения в World Wide Web на основните езици за програмиране. Създадени са няколко уеб рамки, които налагат модела. Тези софтуерни рамки се различават в своите интерпретации, главно по начина, по който отговорностите на MVC са разпределени между клиента и сървъра.

Някои мрежови MVC рамки възприемат тънък клиентски подход, който поставя почти целия модел, изглед и логика на контролера на сървъра. При този подход клиентът изпраща заявки за хипервръзка или подаване на формуляр до контролера и след това получава пълна и актуализирана уеб страница (или друг документ) от изгледа; моделът съществува изцяло на сървъра.

(.NET Core MVC)

.NET (по-рано наречен .NET Core) е безплатна и управлявана компютърна софтуерна рамка с отворен код за операционни системи Windows, Linux и macOS. Той е междуплатформен наследник на .NET Framework. Проектът е разработен предимно от служители на Microsoft чрез .NET Foundation и е издаден под лиценза MIT.

Езикова поддръжка

[](https://en.wikipedia.org/wiki/File:Overview_of_the_Common_Language_Infrastructure_2015.svg)

.NET използва Common Language Infrastructure (CLI)

.NET напълно поддържа C # и F # (и C ++ / CLI от 3.1; активиран само за Windows) и поддържа Visual Basic .NET (за версия 15.5 в .NET Core 5.0.100-preview.4 и някои стари версии, поддържани в стари .NET Core).

VB.NET се компилира и работи на .NET, но от .NET Core 3.1 отделното изпълнение на Visual Basic не е внедрено. Microsoft първоначално обяви, че .NET Core 3 ще включва изпълнението на Visual Basic, но след две години графикът за такава поддръжка беше актуализиран до .NET 5.

Архитектура на .NET

.NET поддържа четири сценария на различни платформи: уеб приложения на ASP.NET Core; приложения от командния ред; библиотеки; и универсални приложения на Windows Platform. Преди .NET Core 3.0 той не е прилагал Windows Forms или Windows Presentation Foundation (WPF), които представят стандартния GUI за настолен софтуер на Windows. Сега обаче .NET Core 3 поддържа настолни технологии Windows Forms, WPF и Universal Windows Platform (UWP). Също така е възможно да се напишат графични приложения на различни платформи, използвайки .NET с GTK # езиково обвързване за инструментариума на GTK джаджа.

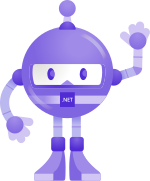
.NET поддържа използването на пакети NuGet. За разлика от .NET Framework, който се обслужва с помощта на Windows Update, .NET разчита на своя мениджър на пакети, за да получава актуализации. От декември 2020 г. обаче .NET актуализациите започнаха да се доставят и чрез Windows Update.

Двата основни компонента на .NET са CoreCLR и CoreFX, които са сравними с Common Language Runtime (CLR) и Framework Class Library (FCL) на внедряването на Common Language Infrastructure (CLI) на .NET Framework.

Като CLI внедряване на виртуална система за изпълнение (VES), CoreCLR е цялостна среда за изпълнение и виртуална машина за управлявано изпълнение на CLI програми и включва компилатор точно навреме, наречен RyuJIT. [a] .NET Core също съдържа CoreRT , .NET Nant Runtime, оптимизиран за интегриране в AOT компилирани собствени двоични файлове.

Като CLI изпълнение на основните стандартни библиотеки, CoreFX споделя подмножество от .NET Framework API, но също така идва със собствени API, които не са част от .NET Framework. Вариант на .NET библиотеката се използва за UWP.

Интерфейсът на командния ред .NET предлага точка за изпълнение на операционни системи и предоставя услуги за разработчици като компилация и управление на пакети. Mascot

[](https://en.wikipedia.org/wiki/File:Dotnet-bot.svg)

dotnet бот, талисманът на общността за .NET

Официалният талисман на .NET е .NET Bot (стилизиран като "dotnet bot" или "dotnet-bot"). Dotnet ботът служи като разработчик на резервни части за първоначално чекиране на изходния код на .NET, когато той е с отворен код. Оттогава се използва като официален талисман.

ASP.NET MVC

ASP.NET MVC е платформа, създадена от Microsoft, която служи за изработване на уеб приложения с шаблон за дизайн Модел-Изглед-Контролер (на английски: Model-View-Controller, MVC). Платформата използва C#, HTML, CSS, JavaScript и бази данни и е съвременно средство за уеб приложения, което обаче не замества изцяло уеб формите. Платформата включва нови тенденции в разработката на уеб приложения, притежава много добър контрол върху HTML и дава възможност за създаване на всякакви приложения. ASP.NET MVC може да бъде много лесно тествана и допълвана, защото е изградена от отделни модули, изцяло независими един от друг. Чрез платформата се създават цялостни приложения, които се стартират, а не единични скриптове (като при PHP например).

MVC модели

**Модел**

Моделът представлява част от приложението, което реализира домейн логиката, също известна като бизнес логика. Домейн логиката обработва данните, които се предават между базата данни и потребителския интерфейс. Например, в една система за инвентаризация, моделът отговаря за това дали елемент от склада е наличен. Моделът може да бъде част от заявлението, което актуализира базата данни когато даден елемент е продаден или доставен в склада. Често моделът съхранява и извлича официална информация в базата данни.

**Изглед модел (view model)**

Изглед модела позволява да се оформят няколко изгледа от един или повече модела от данни или източници в един обект. Този модел е оптимизиран за потребление и изпълнение.

**MVC модели**

MVC (model – view – control) предоставя различни видове и начини за това как да бъде изграден даден модел. Практиката е доказала, че поради липсата на много информация, практики и препоръки, е възможно да се получат разминавания за това как да бъде направен модел. Всъщност, истината е, че няма еднотипно решение, което да задоволява всички възникнали проблеми. Тук ще разгледаме някои от основните модели, които се използват, с описание на конкретен пример. Трябва да се отбележи, че това са модели, използвани при решаването на реални практически проблеми.

**Модел 1 – Обектен домейн модел използван директно като модел-изглед(view model)**

Представете си модел, който изглежда по следния начин:

**public** **class** **Motorcycle**

{

**public** string Make { **get**; **set**; }

**public** string Model { **get**; **set**; }

**public** int Year { **get**; **set**; }

**public** string VIN { **get**; **set**; }

}

Когато този код премине в изглед, той ни позволява да пишем HTML помощници в стила на нашия избор:

<%=Html.TextBox("Make") %>

<%=Html.TextBoxFor(m => m.Make) %>

Разбира се с този модел по подразбиране може да се върнем обратно при контролера:

**public** ActionResult Save(Motorcycle motorcycle)

Въпреки че този модел е доста опростен и работи изчистено и елегантно, той се „чупи“ много често дори и при по-опростени изгледи. Свързването към домейн модела в този случай не винаги е достатъчно за пълно представяна на изгледа.

**Модел 2 – Собствен модел, съдържащ обектния домейн модел**

Ако разгледаме пак мотоциклета от горния пример, в реалния живот е много по-реалистично изгледът за даден обект да се нуждае от повече от една характеристика, за да бъде представен правилно. Примерно, характеристиките от горния пример Make и Model ще бъдат извиквани от падащи менюта. Ето защо, този модел е предназначен да представи изгледа като съдържател на всички характеристики, необходими за правилното представяне:

**public** **class** **MotorcycleViewModel**

{

**public** Motorcycle Motorcycle { **get**; **set**; }

**public** SelectList MakeList { **get**; **set**; }

**public** SelectList ModelList { **get**; **set**; }

}

В този случай, контролерът има отговорност за това моделът MotorcycleView да е правилно попълнен с данни от хранилищата (Пример: получаване на Motorcycle от база данни, извличане на колекция Make/Model от база данни). Нашите HTML помощници леко се променят, защото сега се отнасят за MotorcycleMake, а не само за Make:

<%=Html.DropDownListFor(m => m.Motorcycle.Make, Model.MakeList) %>

Когато формата се публикува, все още е необходимо да има Save() метод:

**public** ActionResult Save([Bind(Prefix = "Motorcycle")]Motorcycle motorcycle)

В този пример трябва атрибутът Motorcycle да се използва като префикс към HTML елемента, от който се интересуваме Този модел е опростен и подходящ за много ситуации. Въпреки това, изгледът става по-сложен, а „чупенето“ отново е проблем тъй като често има разминаване между обектния домейн модел и отдадения модел!

**Модел 3 – Собствен модел, съдържащ обичайния модел – изглед**

Когато изгледът стане по-сложен, обикновено е трудно да се запази обектният модел на домейна в синхрон с работата на изгледа. Ако разгледаме пак горепосочения пример, нека да предположим, че се изисква да се предостави на потребителя поле, в което той да може да добави мотоциклет, ако желае. Когато формата се публикува, контролерът трябва да прецени според стойностите, кой изглед да бъде показан следващ. Последното нещо, което трябва да се направи е да се добави това свойство на нашия домейн – модел, тъй като това е строго изискване.

Нека информацията за мотоциклета да е MotorcycleData:

**public** **class** **MotorcycleData**

{

**public** string Make { **get**; **set**; }

**public** string Model { **get**; **set**; }

**public** int Year { **get**; **set**; }

**public** string VIN { **get**; **set**; }

**public** bool AddAdditionalCycle { **get**; **set**; }

}

Този модел е доста трудоемък и изисква слой за превеждане, за да се очертаят връзките назад и напред между Motorcycle и MotorcycleData, но в случая усилията си струват дори за по-сложни изгледи. Този модел със сериозно застъпен и препоръчан от авторите на книгата MVC in Action[]](https://bg.wikipedia.org/wiki/ASP.NET_MVC#cite_note-3). Идеите са разширени и в публикациите на Jimmy Bogard (един от съавторите на книгата) – How we do MVC – View Models.

В реалния живот, този модел може да е много бърз. Освен че трябва да се очертае връзката между Motorcycle и MotorcycleData, също така може да има номерирана колекция, която да бъде представена с падащо меню. Ако целият този код бъде подаден към контролера, той много бързо ще се бъде напълнен с много код, който се отнася само за изграждането на модел – изгледа, което не е желателно, защото контролерът трябва да не се препълва с излишна информация. Ето защо може да създаден отделен клас, чиято роля да бъде само изграждането на модел – изгледа:

**public** **class** **MotorcycleViewModelBuilder**

{

**private** IMotorcycleRepository motorcycleRepository;

**public** MotorcycleViewModelBuilder(IMotorcycleRepository repository)

{

**this**.motorcycleRepository = repository;

}

**public** MotorcycleViewModel Build()

{

*// code here to fully build the view model*

*// with methods in the repository*

}

}

Това позволява кодът на контролера да изглежда по следния начин:

**public** ActionResult Edit(int id)

{

var viewModelBuilder = **new** MotorcycleViewModelBuilder(**this**.motorcycleRepository);

var motorcycleViewModel = viewModelBuilder.Build();

**return** **this**.View();

}

Изгледът в този модел може да изглежда почити като изгледа в модел 2, но тук имаме предимството да знаем, че само преминаваме през данните, от които имаме нужда – нито повече, нито по-малко. Когато формата се публикува обратно, Save() методът в контролера ще изглежда по следния начин:

**public** ActionResult Save([Bind(Prefix ="Motorcycle")]MotorcycleData motorcycleData)

{

var mapper = **new** MotorcycleMapper(motorcycleData);

Motorcycle motorcycle = mapper.Map();

**this**.motorcycleRepository.Save(motorcycle);

**return** **this**.RedirectToAction("Index");

}

Тази имплементация е много подобна на модела описан от Jimmy Bogard, но без AutoMap. AutoMap ни позволява да запазваме част от кода извън контолера, което може да бъде много полезно.

В сложните реални ситуации, някой вариации на Модел 3 са най-добрия избор, тъй като този модел предоставя най-голяма гъвкавост и вариативност на разработчика.

**Съображения**

Как определяме кой е най-подходящият модел в конкретна ситуация? Ето няколко съображения, които да имате предвид:

**Повторна употреба на код (Re-use)**

Модел 1 и модел 2 са подходящи за употреба при повторение на код. Тук трябва да има връзка между изгледа и обектния домейн модел. Това води до по-кратък код, като картографски слой не е задължителен. Обаче, ако изгледът се различава от домейн модела (което е често срещано) модел 1 и модел 2 са неефективни.

**Несъответствия**

Често се получават несъответствия между нашия домейн модел и съображенията на изгледа. В такъв случай модел 3 е най-подходящ.

**Картографски слой**

Ако е използван обичайния модел 3, трябва да сте сигурни, че е създаден модел за картографкси слой. Въпреки че това означава повече код, който трябва да се напише, това предоставя повече гъвкавост, а също така има и библиотеки, които правят написването му по-лесно.

**Валидация**

Има много начини за валидация, като един от най-популярните е използването на библиотеки. Въпреки че типичните валидации (напр. задължителни полета), ще бъдат същите между домейн модела и изгледа, не всички валидации винаги ще съвпадат. Освен това, не винаги ще може да се контролира домейн модела (например в някой разработки домейн моделът е изложен чрез услуги, които разработчиците само използват). Ето защо има ограничения как валидациите да се свържат с тези класове. Възможно е да се използва отделен клас „мета данни“ за валидиране. Ето защо клас 3 предоставя абсолютен контрол над валидирането.

**Изводи**

До тук разгледаните модели бяха в контекста на ASP.NET MVC. Проблемът с това как най-добре да се справяме с изглед моделите са актуални и при други рамки, като например уеб формите. Ако е възможно да се свържете директно към домейн модела с обикновен клас, това е най-простото и лесно решение. Но когато сложността нараства, отделните модели предоставят разнообразие от възможности за действие.[[5]](https://bg.wikipedia.org/wiki/ASP.NET_MVC#cite_note-5)

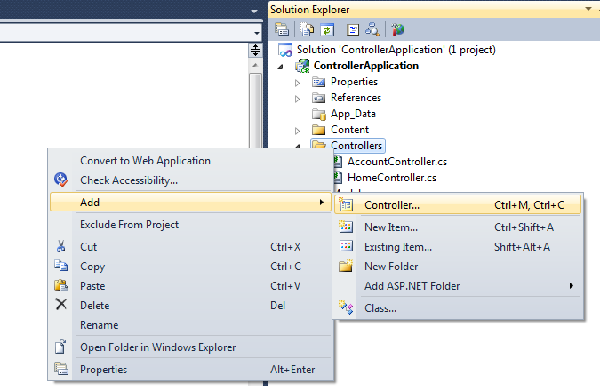
Контролери и действия

**Какво са контролери**

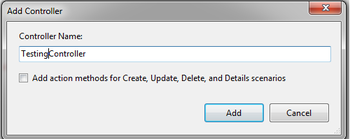
Контролерите са класове, които се създават в MVC приложението. Намират се в папка Controllers. Всеки един клас, който е от този тип, трябва да има име завършващо с наставка „Controller“. Контролерите обработват постъпващите заявки, въведени от потребителя и изпълняват подходящата логика за изпълнение на приложението. Класът контролер е отговорен за следните етапи на обработка:

* Намиране и извикване на най-подходящия метод за действие (action method) и валидиране, че може да бъде извикан.
* Взимането на стойности, които да се използват като аргументи в метода за действие.
* Отстраняване на всички грешки, които могат да възникнат по време на изпълнението метода за действие.
* Осигуряване на клас WebFormViewEngine по подразбиране за отваряне на страници с изглет от тип ASP.NET.

За да създадем контролер натискаме дясното копче на мишката върху папката „controllers“ в прозореца „solution explorer“ на Visual Studio, избираме „add“ и след това „controller“.

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:ExampleForControllers.png)

След появяването на прозореца „Add controller“ трябва да се въведе име на контролера. То трябва да завършва с наставката „controller“.

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:ExapleControllerVS.png)

Контролера е клас който се наследява от базовия клас System.Web.Mvc.Controller. Всеки публичен метод в контролера е показан като „controller action“. Ако искаме даден метод да не бъде извикан, трябва да сложим „NonAction“ атрибут върху неговото име. По подразбиране „Index()“ действието е извикано за контролера, когато друго такова не е изрично упоменато.

**using** **System**;

**using** **System.Collection.Generic**;

**using** **System.Linq**;

**using** **System.Web**;

**using** **System.Web.MVC**;

**namespace** **ControllerApplication.Controllers**

{

**public** **class** **TestingController** : Controller

{

*//*

*// GET: /Testing/*

**public** ActionResult Index() *//Default Action*

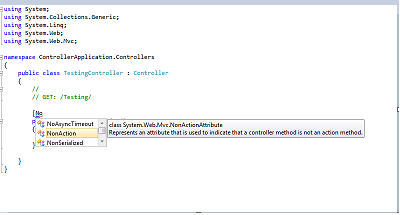
{

**return** View();

}

}

}

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:ExampleNoAction.png)

[[6]](https://bg.wikipedia.org/wiki/ASP.NET_MVC#cite_note-6)

**Методи за действие**

В ASP.NET приложения, които не ползват MVC Framework, взаимодействията с потребителя са организирани и контролирани чрез страници. За разлика от това в приложения с MVC framework взаимодействията с потребителя са организирани чрез контролери и методи за действие. Контролерът определя метода за действие и може да включва толкова методи за действие, колкото са необходими.

Методи за действие обикновено имат функции, които са пряко свързани с взаимодействието с потребителя. Примери за взаимодействие с потребителите са въвеждане на URL адрес в браузъра, кликване върху линк, и подаването на формуляр. Всяко едно от тези потребителски взаимодействия изпраща заявка към сървъра. Във всеки един от тези случаи, URL адреса от заявката съдържа информация, която MVC Framework използва за да включи метод за действие.[[7]](https://bg.wikipedia.org/wiki/ASP.NET_MVC#cite_note-7)

**using** **System**;

**using** **System.Collection.Generic**;

**using** **System.Linq**;

**using** **System.Web**;

**using** **System.Web.MVC**;

**namespace** **ControllerApplication.Controllers**

{

**public** **class** **TestingController** : Controller

{

*//*

*// GET: /Testing/*

**public** ActionResult Index() *//Default Action*

{

**return** View();

}

**public** ActionResult Testing1()

{

**return** View();

}

**public** ActionResult Testing2(string x)

{

**return** View();

}

}

}

В посочения горе пример имаме три действия (Index, Testing1, Testing2), които са дефинирани в контролер клас „TestingController“. За да извикаме различни действия, трябва да напишем следното в полето за адреси:

Начин на изписване: /{controller}/{action}/{id}

* /TestingController/Index :- Извиква Index() действие.
* /TestingController/Testing1 :- Извиква Testing1() действие.
* /TestingController/Testing2/pijush :- Извиква Testing2() действие с „pijush“ за Id параметър.
* /TestingController :- Извиква Index() действие.

**Списък резултати от дейстия.**

Action Results

* View():- Представлява ASP.NET MVC изглед т.е., когато вашия браузър връща HTML. Това е най-рапространения „ActionResult“, който може да върне един контролер.
* PartialView():- Извиква част от ASP.NET MVC изгледа.
* RedirectToAction():- Пренасочва от едно контролер действие в друго. Параметри с които може да се използва това действие са:

-actionName: Името на действието.

-controllerName: Името на контролера.

-routeValues: Стойностите които са предадени на действието.

Метода RedirectToAction() връща „302 found HTTP“ статус код на браузъра, за да може да направи пренасочването към новото действие. Едно от предимствата на това действие е, че когато се прави пренасочване в браузъра – полето за адреси се обновява с новия URL линк. Недостатък е, че трябва да се направи повторна заявка от браузъра към сървъра.

* Redirect(): Пренасочва към друг контролер или URL линк.
* Content(): Данни които постъпват към браузъра. Параметрите, които могат да се използват като аргументи, са:

-string: Изобразява string на браузъра.

-contentype: Типът MIME на данните (по подразбиране за text/html).

-contentEncoding: Текстовото кодиране на данните (например:-Unicode или ASCII)

* Json(): JavaScript object Notation (JSON) e изобретен от Douglas Crockford като по-лека алтернатива на XML за пращане на данни по интернет в Ajax приложения. Метода Json() използва клас в .NET framework наречен JavaScriptSerializer за да преобразува обект в JSON репрезентация.
* File(): Връщане на фаил от действие. Този метод приема следните параметри:

-filename, contentType, fileDownloadName, fileContents, fileStream.

* JavaScript(): Представлява JavaScript фаил.
* HandleUnknownAction(): Този метод се извиква автоматично когато контролера не може да намери ресурс. По подразбиране метода изкарва грешка „404 resource Not Found“. Съобщението за грашка може да бъде сменено с друго, това обаче изисква да се пренапише контролер класа.

Валидация

Валидирането на входните данни, въведени от потребителя (user input), позволява да се уверим, че те съответстват на модела на данните в ASP.NET MVC. По този начин се защитава приложението от потребителски грешки или злонамерени потребители. Има много начини да се включи валидиране в едно MVC приложение. Например, може да бъде използвана публично достъпна библиотека за **валидиране**, като Microsoft Enterprise Library Validation Block.

**Как работи валидирането**

Когато потребителят потвърди (submit) формуляр(form), данните от формуляра се предават на **метод за действие** (action method) с помощта на ViewDataDictionary колекция. ViewDataDictionary има ModelState свойство(property), което съдържа колекция от ModelState обекти. За всеки модел, който е дефиниран в MVC приложението, се създава съответния ModelState обект и добавя в колекцията. **Действието**, което получава данните от формуляра определя правилата за проверка, прилагани към формуляра с данни. Ако правилото е нарушено, действието използва ModelState свойство за да предаде информация за грешка обратно към **изгледа**.

**Пример за валидация**

Примерът показва клас Person, който дефинира свойства (properties) за съхраняване на лични данни:

**public** **class** **Person**

{

**public** string Name { **get**; **set**; }

**public** int Age { **get**; **set**; }

**public** string Street { **get**; **set**; }

**public** string City { **get**; **set**; }

**public** string State { **get**; **set**; }

**public** string Zipcode { **get**; **set**; }

**public** string Phone { **get**; **set**; }

**public** string Email { **get**; **set**; }

}

Следното дефинира форма за въвеждане на стойности и създаване на инстанция на класа Person. Ако потребителят въведе невалидна стойност за инстанцията на Person, изгледът се генерира отново, този път със съобщение за грешка, което е подадено на изгледа от свойство на ModelState:

<h2>Create</h2>

<%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try again.") %>

<% **using** (Html.BeginForm()) {%>

<fieldset>

<legend>Fields</legend>

<p>

<label **for**="Name">Name:</label>

<%= Html.TextBox("Name") %> Required

<%= Html.ValidationMessage("Name", "\*") %>

</p>

<p>

<label **for**="Age">Age:</label>

<%= Html.TextBox("Age") %> Required

<%= Html.ValidationMessage("Age", "\*") %>

</p>

<p>

<label **for**="Street">Street:</label>

<%= Html.TextBox("Street") %>

<%= Html.ValidationMessage("Street", "\*") %>

</p>

<p>

<label **for**="City">City:</label>

<%= Html.TextBox("City") %>

<%= Html.ValidationMessage("City", "\*") %>

</p>

<p>

<label **for**="State">State:</label>

<%= Html.TextBox("State") %>

<%= Html.ValidationMessage("State", "\*") %>

</p>

<p>

<label **for**="Zipcode">Zipcode:</label>

<%= Html.TextBox("Zipcode") %>

<%= Html.ValidationMessage("Zipcode", "\*") %>

</p>

<p>

<label **for**="Phone">Phone:</label>

<%= Html.TextBox("Phone") %> Required

<%= Html.ValidationMessage("Phone", "\*") %>

</p>

<p>

<label **for**="Email">Email:</label>

<%= Html.TextBox("Email") %> Required

<%= Html.ValidationMessage("Email", "\*") %>

</p>

<p>

<input type="submit" **value**="Create" />

</p>

</fieldset>

<% } %>

<div>

<%=Html.ActionLink("Back to List", "Index") %>

</div>

Ако възникне грешка при валидирането, **методът за действие** извиква метода AddModelError за да добави грешката в свързания обект ModelState. След като се изпълнят правилата за валидиране, методът за действие използва свойството IsValid на колекциата ModelStateDictionary за да се определи дали получената информация е в съответствие с **модела**:

[AcceptVerbs(HttpVerbs.Post)]

**public** ActionResult Create(Person person)

{

**if** (person.Name.Trim().Length == 0)

{

ModelState.AddModelError("Name", "Name is required.");

}

**if** (person.Age < 1 || person.Age > 200)

{

ModelState.AddModelError("Age", "Age must be within range 1 to 200.");

}

**if** ((person.Zipcode.Trim().Length > 0) && (!Regex.IsMatch(person.Zipcode, @"^\d{5}$|^\d{5}-\d{4}$")))

{

ModelState.AddModelError("Zipcode", "Zipcode is invalid.");

}

**if** (!Regex.IsMatch(person.Phone, @"((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}"))

{

ModelState.AddModelError("Phone", "Phone number is invalid.");

}

**if** (!Regex.IsMatch(person.Email, @"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$"))

{

ModelState.AddModelError("Email", "Email format is invalid.");

}

**if** (!ModelState.IsValid)

{

**return** View("Create", person);

}

people.Add(person);

**return** RedirectToAction("Index");

}

Изгледи

**Какво е изглед**

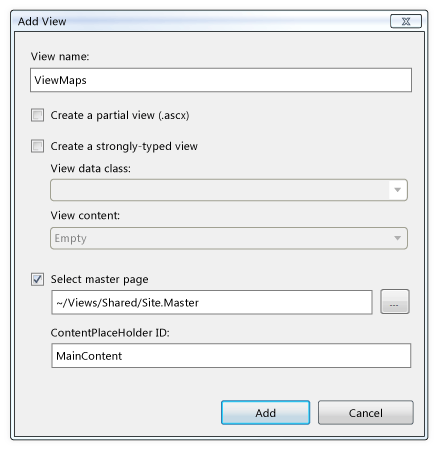
Изгледите са тези, които определят как ще бъде визуализиран потребителският интерфейс (UI) на приложението. В ASP.NET MVC се поддържат средства (engines) за генериране на изгледи.

**Начин на работа на изгледите**

Когато потребителт взаимодейства с изглед, данните се **рутират** от изгледа до метод за действие, който от своя страна може да създаде друг изглед. Едно MVC приложение може да има няколко контролери, всеки от които може да съдържа множество методи за действие, а всяко действие може да създаде различен изглед. Изгледите са организирани в папки, като името им се определя от това на свързания контролер (например изгледите от HomeController са в папката \Views\Home).

**Добавяне на изглед на MVC приложение в Visual Studio**

1. В Solution Explorer, щракнете с десния бутон на мишката върху папката, която носи името на нужния ви контролер. Например ако контролерът е с име HomeController, щракнете с десния бутон върху папката Home.
2. Натиснете Add, след което View. Отваря се следният диалогов прозорец:

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:AddView.png)

1. Попълнете име, вид изглед и т.н, в зависимост от това какъв изглед искате да създадете.
2. Натиснете Add[[9]](https://bg.wikipedia.org/wiki/ASP.NET_MVC#cite_note-9)

Рутиране

Рутирането е метод за валидация и обработка на заявката от потребителя. След като рутирането получи заявката, то проверява дали тази заявка отговаря на рутиращите шаблони и ако бъдат намерени съответствия, първият физически поставен в кода шаблон, отговарящ на дадената заявка, изпълнява функции за подаване на данните от заявката към съответните контролери, действия и параметри.

**Как се регистрира и дефинира рутирането**

При стартиране на **ASP.NET MVC** приложението първият файл, който се стартира, се нарича „**Global.asax**“ и съответно в него първият метод, който се извиква, е с име „Application\_start()“. В този метод има фунции за регистрация на **Area**, **WebApi**, **Filters**, **Routing** и **Bundle**. Именно тук се регистрира и нашето рутиране (Routing) чрез извикване на класа **RoutingConfig**, през метода **RegisterRoutes** и през него се регистрира рутирането в глобалната таблица **RouteTable**.

**public** **class** **MvcApplication** : System.Web.HttpApplication

{

**protected** **void** Application\_Start()

{

AreaRegistration.RegisterAllAreas();

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

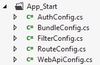
RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

AuthConfig.RegisterAuth();

}

}

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:App_Start-RouteConfig.png)

Конфиг файловете в папка Апп Старт

За да дефинирате рутиранете е нужно да конфигурирате файла **RouteConfig.cs**, който се намира в папката **App\_Start** заедно с останалите Config файлове.

Тук вече се конфигурира метода **RegisterRoutes**, който се извиква от „**Global.asax**“ файла. Този метод използва глобална таблица, която може да бъде извикана отвсякъде в нашето приложение и в нея има методи **IgnoreRoute** и **MapRoute**. **IgnoreRoute** методът се използва за да се пренебрегнат заявки, които не желаем да обработим. Чрез метода **MapRoute** ние регистрираме нашето рутиране като то си има **име**, **url-шаблон** и **стойности по подразбиране**.

**public** **static** **void** RegisterRoutes(RouteCollection routes)

{

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: **new** {

controller = "Home",

action = "Index",

id = UrlParameter.Optional

}

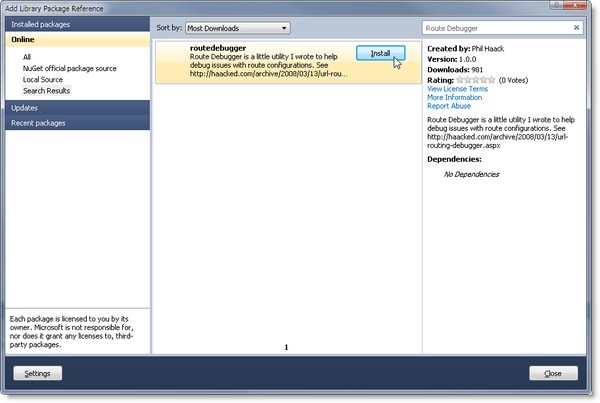
};

}

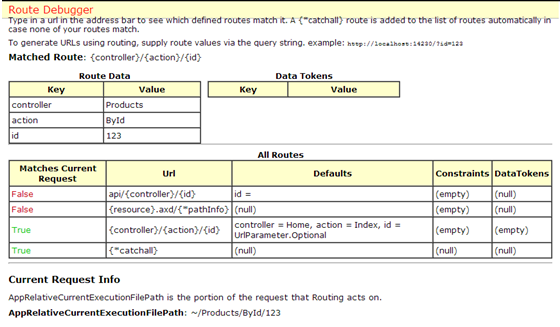
Шаблоните в рутирането са **case-insensitive**, което означава, че шаблонът ще разпознае данните без значение дали са с малки или главни букви. Ако при заявката никой от шаблоните не бъде обхванат, то тогава сървърът връща **error-404**.

**Дебъгване на рутирането**

**ASP.NET MVC** ни предоставя готов дебъгер който може да свалим през **NuGet packege** и в него потърсите **RouteDebugger**.

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:RouteDebuggerNugetInstall.png)

Дебъгерът се появява отдолу на страницата след нейното стартиране.

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:DebuggerPreview.png)

Ако желаете той да бъде спрян, е нужно да отворите **Web.Config** файла на вашето приложение и в него да промените този ред „<add key=“RouteDebugger:Enabled" value="**true**" />", като смените стойността **value** да е равна на **false** (<add key="RouteDebugger:Enabled" value="**false**" />).

<appSettings>

<**add** key="webpages:Version" **value**="1.0.0.0" />

<**add** key="ClientValidationEnabled" **value**="true" />

<**add** key="UnobtrusiveJavaScriptEnabled" **value**="true" />

<**add** key="RouteDebugger:Enabled" **value**="true" />

</appSettings>

Свързване на HTTP сървър с backend език за програмиране

Web Application сървър

За да завършим изследването на базите данни и Web, ще прегледаме основните сървъри за приложения в Web (Web Application Server) с подходящите за тях Web сървъри и компоненти за бази данни.

Нека най-напред допълним фигурата, която започнахме в частта Web сървър:

Както става ясно от картинката, а и ще бъде изяснено по-подробно надолу, Web Application сървъра или Сървъра за приложения, е свързващото звено между Web сървъра и базите данни. Как става самата връзка между тези компоненти, ще разгледаме след като се запознаем с част от Web application сървърите в момента, индивидуално и в детайли.

Web Application сървърите обикновено се свързват с Web сървъра чрез неговия Приложен Програмен Интерфейс (Application Programing Interface - API) и играят ролята на разширение на Web сървъра, което може да обменя информация с базата данни чрез ODBC. Алтернатива на тези инструменти е приложение, разработено с използване на традиционни езици за програмиране (като C++, Python, Perl или Java), достъпът до което се осъществява чрез CGI (Common Gateway Interface). Други възможности за програмиране включват Java Server Pages (JSP) или други Web сървър-специфични разширения, подобни на Microsoft-ските Active Server Pages и на Natescape-ския server-side JavaScript за бази данни.

Понеже обекта на нашето изследване са не Web Application сървърите сами по себе си, а как те се използват за създаване на динамични страници и свързването на тези страници с бази данни, ще разгледаме и основните програмни езици за създаването на Web сайтове.

Това, което ще имаме предвид тук под динамична страница е страница, която е конкретно създадена от сървъра в отговор на клиентска заявка и която може да бъде различна всеки път в зависимост от разнообразни обстоятелства. Важното в случая, е че сървърът не просто чете обикновена HTML страница или текстов файл от диска и го изпраща към клиента - той трябва самостоятелно да извърши някаква работа за да създаде страницата. Интерфейсът, използван за тези динамични страници, ще ги наричаме приложения, се нарича CGI (Common Gateway Interface), и беше обект на разглеждане в предната глава. Първите приложения всъщност са компилирани програми, обикновено написани на С или C++. Но необходимостта да се програмира на С и да се прекомпилира изпълнимата програма всеки път, когато се налага да се направи малка промяна на създадените от нея текст или форматиране, ограничава използването на CGI и динамични страници. Затова беше разработен начин за създаване на страниците чрез скриптов език - Practical Extraction and Reporting Language (практичен език за извличане и отчети), или Perl. Той позволява на създателите на информация да пишат код на език, който до голяма степен прилича на опростена версия на С или C++. Той ще бъде разгледан по-подробно малко по-нататък.

Perl е популярен език в Web, особено в базирани на Unix или Linux системи, обаче ако сте Web разработчик, който няма предишни познания по С или C++ ще е много трудно да се го разучите. Вместо него се появяват нови скриптови езици, които улесняват създаването на страници. На системи, базирани на Unix и Linux и техните варианти, става популярен един нов език, наречен РНР (Personal Home Page. Съществуват и езици, насочени конкретно към определени типове потребители: например TCL е проектиран за по-лесно извършване на сложни математически изчисления в научна среда.

След създаването на SAPI, Microsoft и други независими разработчици произведоха приложения, които се свързват към Web сървъра чрез него. Това е начинът- чрез разработката на Microsoft - ISAPI, Active Server Pages, ASP да се свързват към IIS, както и на други техники на Microsoft за динамично свързване от сървърната страна. /Преди ASP най-често използваното средство беше Internet Database Connector (IDC). Той разкри един цял нов свят за Web разработчиците, използващи платформи на Microsoft, улеснявайки създаването на динамични страници, които използват база данни./ Това е една отворена среда за разработване на динамични и мощни Web-базирани приложения, в които се използват HTML, скриптови и ActiveX компоненти. Active Server Pages поддържа VBScripts и Jscript.

Вследствие успеха на ASP, Microsoft разработи Windows Scripting Host (WSH) - технологията, която дава възможност за изпълнение на скриптове в Windows 95, 98 и NT 4. WSH е независим от езика и поддържа JScript, VBScript и други езици. WSH дава възможност за изпълнение на скриптове от работното поле на Windows или от конзолен (MS-DOS) прозорец. WSH скриптовете са завършени сами по себе си и не е необходимо да бъдат вграждани в HTML документи. WSH е технология, впечатляваща с това, че разширява възможностите на JScript отвъд Web, като пренася езика в работното поле на Windows и операционната система. WSH скриптовете могат да се използват вместо MS-DOS скриптовете, като дават възможност за пълно използване на предимствата на Windows GUI, ActiveX и функциите на операционната система. /WSH може да бъде свален безплатно от Web сайта на Microsoft на адрес http://msdn.microsoft.com/scripting/

Както и при ASP, aналогичен е случая при Cold Fusion страниците на Allaire, които се свързват към Web сървъра си посредством стандартния сървърен API, но освен това поддържат и CGI интерфейса.

Netscape също създадоха графична среда за разработване и управляване на Web сайтове – LiveWire - за използване в своите сървъри. Една от възможностите на LiveWire е поддръжката на сървърни програми, написани на езика JavaScript.

Съществуват обаче и начини, за създаване на динамични странци за Web, които не се нуждаят от специфичен интерфейс за свързване към Web сървъра – такива са Java сървлетите. Те комуникират с Web сървъра без да използват специфичен за него API. Като се смесят обикновен статичен HTML с динамично генерирано съдържание от сървлети, се получават Java Server Pages, JSP.

Продължителят на ASP от страна на Microsoft се появи съвсем наскоро и беше наречен ASP+ или ASP.Net. Ако сравним ASP и ASP+, ще забележим че има два основни разлики във вторите сървърни страници: ASP+ поддържа кодовно написани и компилируеми езици като например Visual Basic, C++, Perl, и поддържа сървърни контроли, които могат да разделят кода от съдържанието, позволявайки по този начин добавяне в страниците. И въпреки че ASP+ не е обратно съвместима с ASP технологията, е възможно да се развива заедно с нея.

А сега нека разгледаме малко по-подробно някои от начините за създаване на динамични страници и свързването на тези страници с бази данни през Web, споменати по-горе.

Комуникация с база от данни

За свързване с базата данни MySQL, трябва да имате инсталиран NuGet във вашето приложение. Отидете на Project root-> appsettings. json-> въведете низ за връзка, както е показано по-долу.

<https://www.youtube.com/watch?v=wM-kXgdHw5w>

Connection to MySQL With ASP.NET Core

In this article, we are going to understand how to connect .NET CORE Application with MySQL and read data from MySQL, using .NET Core MySQL connector.

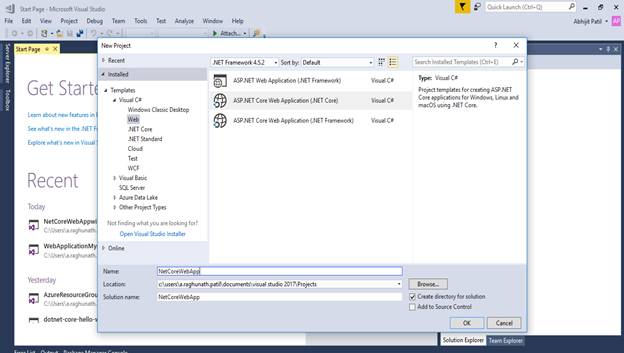
Prerequisites

1. MySQL (You can get MySQL IDE – MySQL Work Bench [here](https://dev.mysql.com/downloads/workbench/)).
2. NET CORE Environment Setup. (you can get it [here](https://www.microsoft.com/net/download/core)).
3. Visual studio 2015 or Visual Studio 2017(download VS 2017 [here](https://www.visualstudio.com/vs/whatsnew/)).
4. MySQL database.

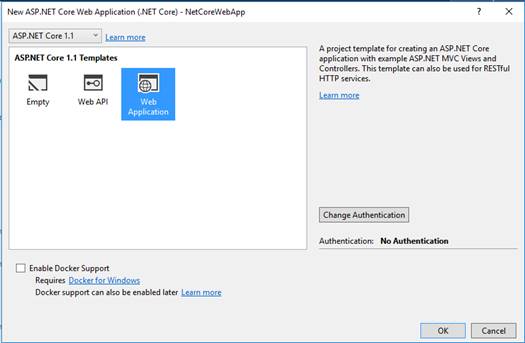
Steps to create a .NET Core Web app with VS 2017

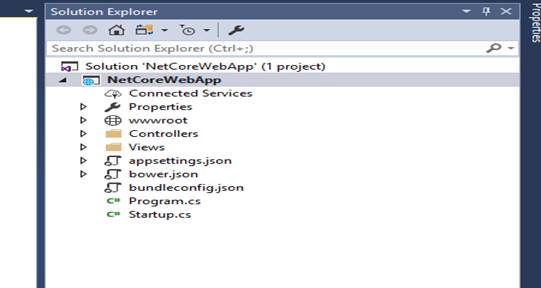
Go to File -> New Project -> Select Web from templates->choose ASP.NET Core Web Application (.NET Core).

Provide the name for the Web app and click OK.

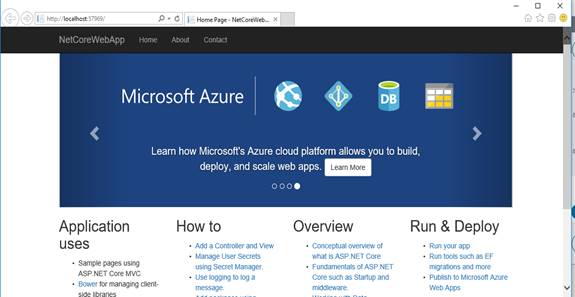


Select the Web Application template and click OK. It will create a .NET Core Web app for you.





Clean build and run the Application for testing.



Set up MySQL database

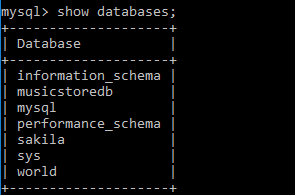
To create the MySQL database, we must-have IDE for MySQL or we need to have MySQL CLI. We can create the database, either with IDE or using CLI commands.

We can create the database in both ways. Let’s check out CLI first.

1. To create a database, using CLI, we first need to log in with the password, which we have provided at the time of installation.
2. After login, type the command “create database MusicStoreDB;”.

ASP.NET Core

1. To check if DB is created or not, run the command given below.



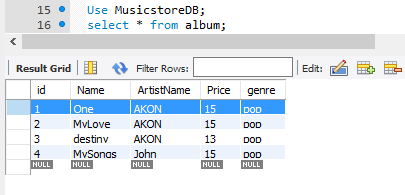
1. Similarly, we run SQL queries on CLI, create tables, and insert data into it.
2. Now, let's check with MySQL workbench IDE
3. Select the Query template. Write SQL query on query editor. Click the Lighting icon.

ASP.NET Core

on the top menu.



1. After creating a table, insert records in created DB, run Select query on the table.

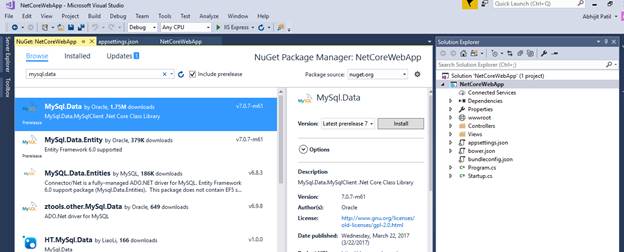


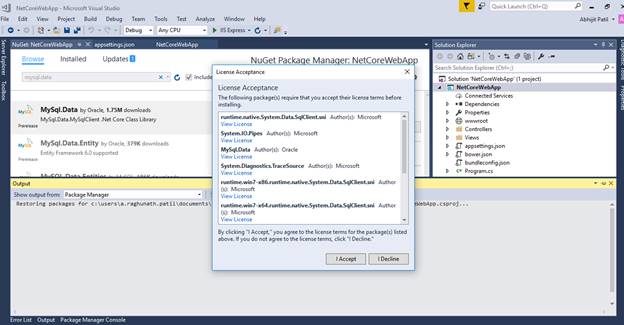
1. For now, we did for a single table only. We are done with the database creation and data insertion.

Connect with MySQL

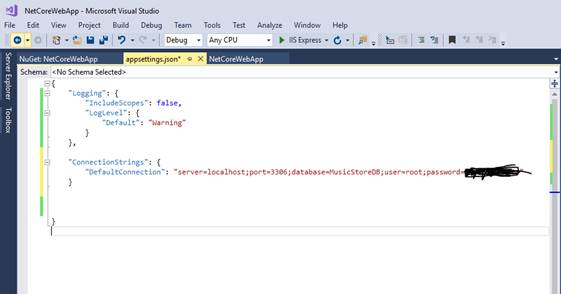
To connect with the MySQL database, we must have some NuGet installed in our Application.

1. Go to Application ->right click on project name-> select Manage Nuget Packages-> Type MySql .Data

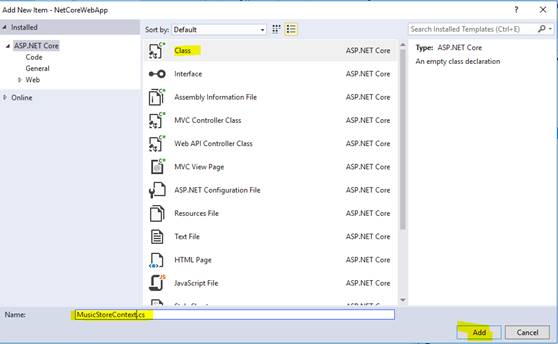


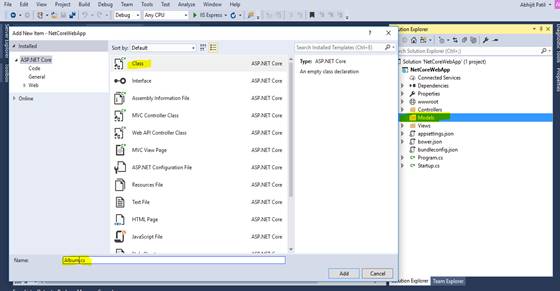


1. Go to Project root-> appsettings.json-> enter the connection string, as shown below.



1. Create new folder Models and add Class ‘MusicStoreContext’& ‘Album’ in it.

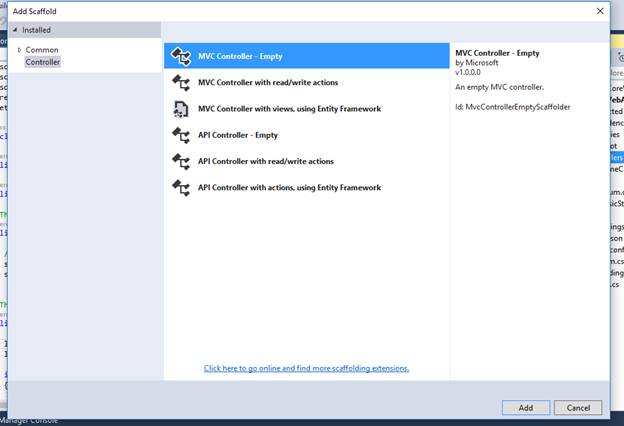




1. Add Album properties in an Album class.
   1. namespace NetCoreWebApp.Models
   2. {
   3. **public** **class** Album
   4. {
   5. **private** MusicStoreContext context;
   7. **public** **int** Id { get; set; }
   9. **public** string Name { get; set; }
   11. **public** string ArtistName { get; set; }
   13. **public** **int** Price { get; set; }
   15. **public** string Genre { get; set; }
   16. }
   17. }
2. Create a new MusicStoreContext class, which will contain the connections and MusicStore data entities, as shown below.
   1. **using** MySql.Data.MySqlClient;
   2. **using** System;
   3. **using** System.Collections.Generic;
   5. **namespace** NetCoreWebApp.Models
   6. {
   7. **public** **class** MusicStoreContext
   8. {
   9. **public** **string** ConnectionString { **get**; **set**; }
   11. **public** MusicStoreContext(**string** connectionString)
   12. {
   13. **this**.ConnectionString = connectionString;
   14. }
   16. **private** MySqlConnection GetConnection()
   17. {
   18. **return** **new** MySqlConnection(ConnectionString);
   19. }
   20. }
   21. }
3. To use context in our Application, we need to register instance as a Service in our Application. To register context, we need to add on line of code in ‘startup.cs’ file under ‘ConfigureServices’ method.
   1. using Microsoft.AspNetCore.Builder;
   2. using Microsoft.AspNetCore.Hosting;
   3. using Microsoft.Extensions.Configuration;
   4. using Microsoft.Extensions.DependencyInjection;
   5. using Microsoft.Extensions.Logging;
   6. using NetCoreWebApp.Models;
   7. namespace NetCoreWebApp
   8. {
   9. **public** **class** Startup
   10. {
   11. **public** Startup(IHostingEnvironment env)
   12. {
   13. **var** builder = **new** ConfigurationBuilder()
   14. .SetBasePath(env.ContentRootPath)
   15. .AddJsonFile("appsettings.json", optional: **false**, reloadOnChange: **true**)
   16. .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: **true**)
   17. .AddEnvironmentVariables();
   18. Configuration = builder.Build();
   19. }
   21. **public** IConfigurationRoot Configuration { get; }
   23. // This method gets called by the runtime. Use this method to add services to the container.
   24. **public** **void** ConfigureServices(IServiceCollection services)
   25. {
   26. // Add framework services.
   27. services.AddMvc();
   28. services.Add(**new** ServiceDescriptor(**typeof**(MusicStoreContext), **new** MusicStoreContext(Configuration.GetConnectionString("DefaultConnection"))));
   29. }
   31. // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
   32. **public** **void** Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
   33. {
   34. loggerFactory.AddConsole(Configuration.GetSection("Logging"));
   35. loggerFactory.AddDebug();
   37. **if** (env.IsDevelopment())
   38. {
   39. app.UseDeveloperExceptionPage();
   40. app.UseBrowserLink();
   41. }
   42. **else**
   43. {
   44. app.UseExceptionHandler("/Home/Error");
   45. }
   47. app.UseStaticFiles();
   49. app.UseMvc(routes =>
   50. {
   51. routes.MapRoute(
   52. name: "default",
   53. template: "{controller=Home}/{action=Index}/{id?}");
   54. });
   55. }
   56. }
   57. }

Fetch data from My SQL database

1. To get the data from the database, we need ‘GetAllAlbums()’ method, our DB context, add ‘GetAll Albums()’ method in “MusicStoreCotext” class.
   1. **public** List<Album> GetAllAlbums()
   2. {
   3. List<Album> list = **new** List<Album>();
   5. **using** (MySqlConnection conn = GetConnection())
   6. {
   7. conn.Open();
   8. MySqlCommand cmd = **new** MySqlCommand("select \* from Album where id < 10", conn);
   10. **using** (var reader = cmd.ExecuteReader())
   11. {
   12. **while** (reader.Read())
   13. {
   14. list.Add(**new** Album()
   15. {
   16. Id = Convert.ToInt32(reader["Id"]),
   17. Name = reader["Name"].ToString(),
   18. ArtistName = reader["ArtistName"].ToString(),
   19. Price = Convert.ToInt32(reader["Price"]),
   20. Genre = reader["genre"].ToString()
   21. });
   22. }
   23. }
   24. }
   25. **return** list;
   26. }
2. Now, we need controller to manage our code. Add controller with name AlbumsController.



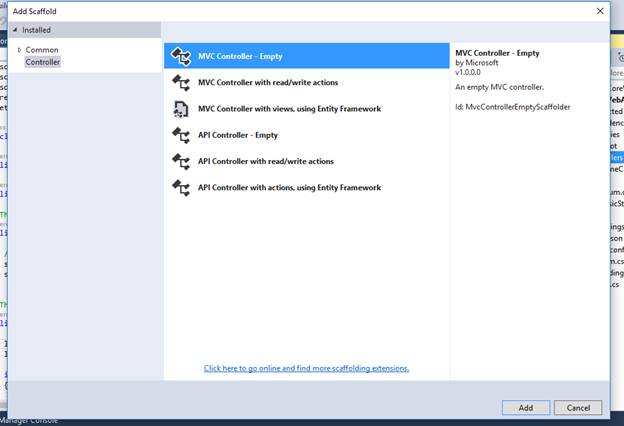
Add the code given below in Album Controller to get the data from DB.

* 1. namespace NetCoreWebApp.Controllers
  2. {
  3. **public** **class** AlbumController : Controller
  4. {
  5. **public** IActionResult Index()
  6. {
  7. MusicStoreContext context = HttpContext.RequestServices.GetService(**typeof**(NetCoreWebApp.Models.MusicStoreContext)) as MusicStoreContext;
  9. **return** View(context.GetAllAlbums());
  10. }

  13. }
  14. }

1. After adding the controller and code, we need a view to display the data to the end-user. Create a folder under Views with name Albums. Right-click on the Albums folder and add new view Albums.

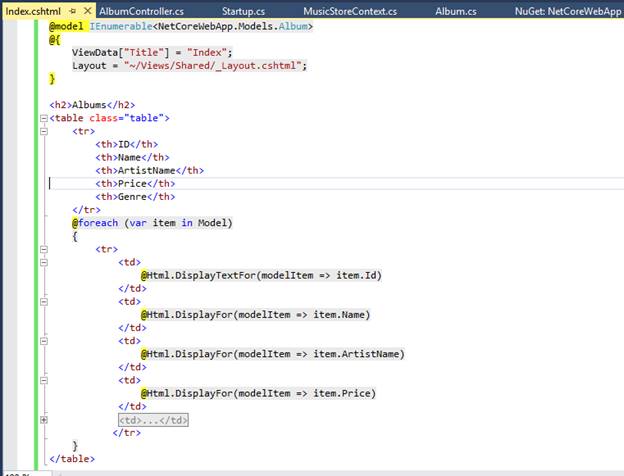
1. Select the Layout page by clicking the button. Now, click add.



1. You can create a view from by selecting the data model in the dropdown or you can create a blank view with the default index name and add the code, as shown below.



1. To route to our Album action, we need to update Startup.cs and add Album controller name, so we after running an app directly. We get the “Album/index” page.
   1. app.UseMvc(routes =>
   2. {
   3. routes.MapRoute(
   4. name: "default",
   5. template: "{controller=Album}/{action=Index}/{id?}");
   6. });
2. Now, just run an Application and we will get the output, as shown below.



Заключение

В тази статия видяхме как да осъществим връзка с базата данни на MYSQL Server. В случай на дистанционно, трябва да актуализираме низа на връзката с подходящо име на сървър и име на порт.

Шаблонни езици от страна на сървъра.

Шаблонни неволи. Преди SPA рамките да излязат на сцената, почти всички приложения бяха разработени с използване на езици на сървърна страна с шаблониращи системи: PHP, Ruby, Python, Java, C # и др. Приложенията, изградени по този начин, рядко имаха сложна клиентска логика или взаимодействие, което изискваше писане на партиди на JavaScript кода.

Скриптове от страна на сървъра: от погледа

Най-популярните скриптови езици и рамки от страна на сървъра включват PHP, ASP.NET, Node.js, Java, Ruby, Perl и Python. Тези скриптове се изпълняват на уеб сървър и отговарят на клиентски заявки чрез HTTP, за да доставят динамично и персонализирано съдържание на потребителя.

Скриптове от страна на сървъра са необходими за достъп или съхраняване на постоянни данни като информация за потребителския профил. Независимо дали това включва изтегляне на данни от файлов сървър, база данни или пощенски сървър, всичко това може да се направи със скриптов език от страна на сървъра като PHP.

Една от ключовите разлики между клиента и сървъра е, че кодът от страна на сървъра винаги е отговорен за предоставянето и съхраняването на всички ресурси, необходими на клиента. Това може да се направи при зареждане на страницата, или интерактивно, докато потребителят прави заявк и промени (използвайки техники като AJAX - повече за това по-долу).

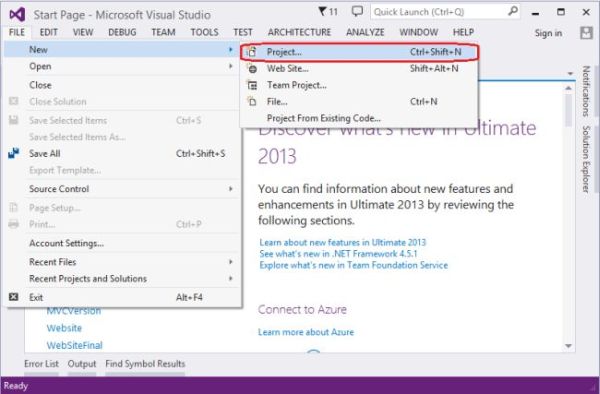
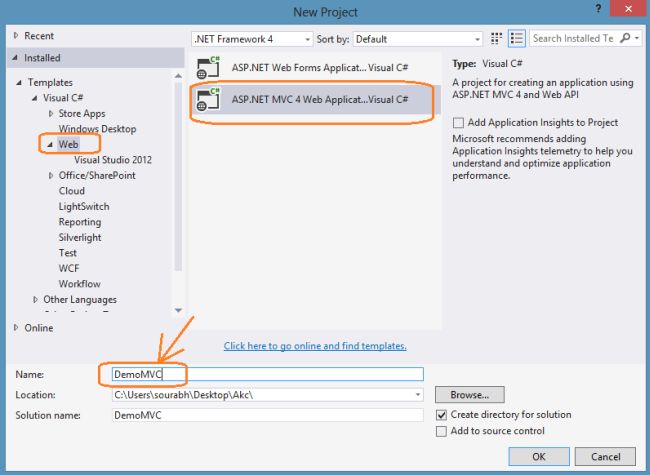
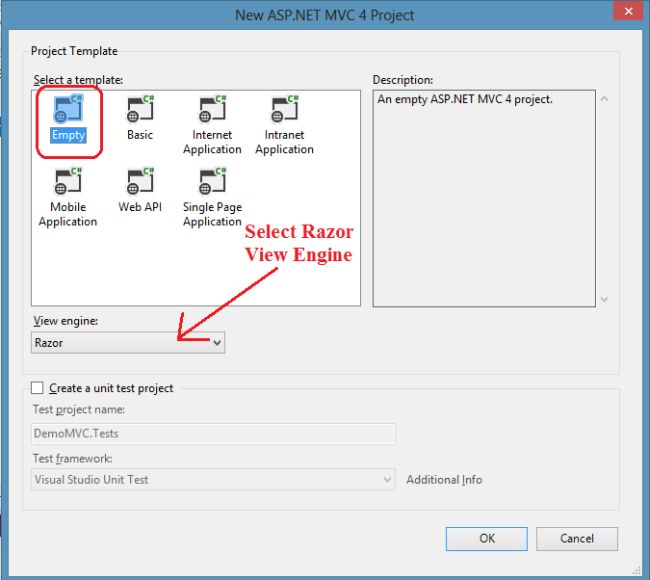
Изходният код от скриптове от страна на сървъра обикновено е напълно скрит от крайния потребител. Например, PHP се изпълнява изключително на сървъра, за да генерира HTML, който от своя страна се визуализира от клиента - вашият браузър всъщност никога не вижда никакъв PHP код. Дори да кликнете върху „Преглед на източника на страницата“, ще видите само HTML, а не PHP скрипта, който го е генерирал.

От решаващо значение е задният сървър да удостоверява всяка заявка от страна на клиента, преди да осъществи достъп или да актуализира каквито и да било ресурси. Фактът, че JavaScript се изпълнява в браузъра, означава, че потребителят може да промени кода, понякога със злонамерено намерение. Затова никога не се доверявайте на данни от страна на клиента, освен ако не могат да бъдат правилно удостоверени. Има много техники за поддържане на удостоверяване, като JSON Web Tokens е едно от най-популярните решения.

Един от недостатъците на скриптове от страна на сървъра е, че просто казано, това означава повече работа за сървъра. Много скриптове, работещи на сървъра, могат да доведат до по-ниска производителност, особено когато заявките на клиента към сървъра се случват често.

Създаване на MVC уеб приложения

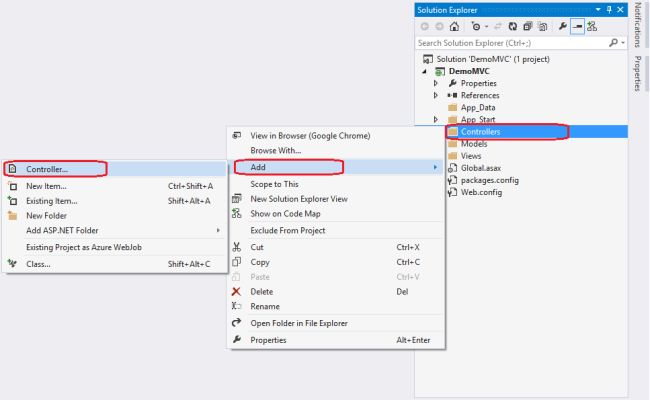
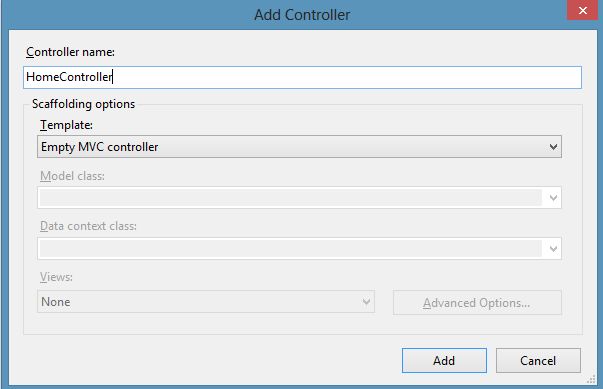
In my previous article ([Get version of your MVC application](http://www.c-sharpcorner.com/UploadFile/75a48f/get-version-of-your-mvc-application/)) we saw how to get your version of your MVC application. Now in this article we will see how to create your own first application using MVC.  
  
The following is the procedure to create your MVC application.

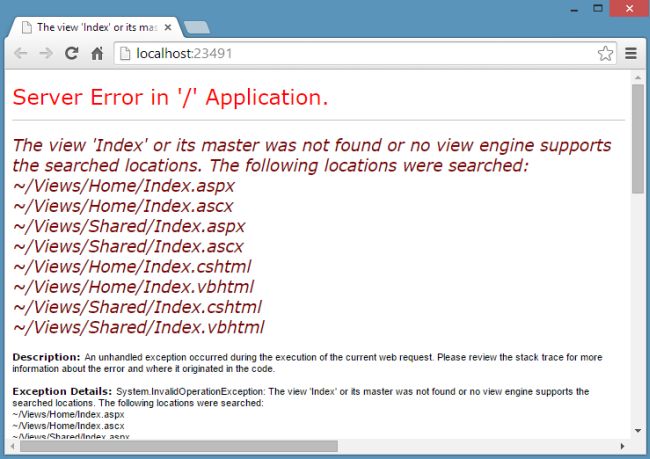
1. Open Visual Studio.
2. Click **File > New > Project**or press **CTRL+SHIFT+N**.  
     
   
3. In the installed templates select Web and select ASP.NET MVC 4 Web Application.
4. Set the name for the application to something such as "**DemoMVC**", then click OK.  
     
   
5. Then select "Empty" Template. Select "Razor" as a view engine and click OK.  
     
   

At this point your MVC application has been created.  
  
Now notice that in the Solution Explorer, you have several folders and files. There will be the three folders Models, Views, Controllers. As the name suggests, the Models folder contains Models, the Views folder contains Views and the Controllers folder contains Controllers. The following are their purposes:

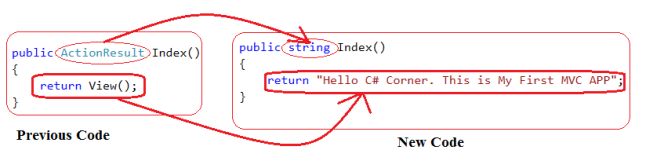
* **Model:** Model is the lowest level of the pattern that maintains the data.
* **View:** View is responsible for displaying all or portion of the data in front of users.
* **Controller:** Controller interacts among Models and Views.

Now let's add a controller to your project.  
  
Use the following procedure to add a Controller to your project:

1. In the Solution Explorer, right-click on the "Controllers" folder.
2. Select the "Add" menu item and click on the "Controllers" menu item.  
     
   
3. Set the controller name as a "**HomeController**"and leave other other options as the default.  
     
   
4. After clicking on the Add button in the controllers folder there will be a "**HomeControllers.cs**" file. It in other words that controller is nothing but a C# code file.

Now if you just press **CTRL+F5**and run your project, you will get an error as shown below:  
  


To fix this error you need to add a view to the Views folder with the Name "Index". Here I am not discussing views and I am fixing this error in another way. I am changing the return type of the following action that is automatically added to the HomeController Class.  
  
I am changing the return type of the Index() action from "ActionResult" to "string" type.

  
  
**Code**

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Web;
5. **using** System.Web.Mvc;
7. **namespace** DemoMVC.Controllers
8. {
9. **public** **class** HomeController : Controller
10. {
11. **public** **string** Index()
12. {
13. **return** "Hello C# Corner. This is My First MVC APP";
14. }
16. }
17. }

Now run the file.  
  
**Output**  


Управление на състоянието в уеб приложенията

Шаблон за наблюдение на наличността на уеб приложения

14.10.2019

14 минути за четене

Шаблонът за наблюдение на наличността на уеб приложения ви позволява да създавате тестове за наблюдение на наличността за един или повече URL адреси на уеб приложения и да стартирате тези тестове за наблюдение от вътрешни местоположения. В допълнение към изгледите за състояние и предупреждение, можете да покажете състоянието на тези тестове в предоставеното табло за управление на картата и в таблото с подробности.

Сценарии

Използвайте шаблона за наблюдение на наличността на уеб приложения в сценарии, при които трябва да наблюдавате уеб-базирани приложения от различни места, за да видите дали работят в съответствие с определени изисквания, които можете да посочите.

Вътрешни местоположения

Възможно е да имате уеб приложения, които трябва да са достъпни по всяко време на вътрешни места. Използвайте шаблона за наблюдение на наличността на уеб приложения, за да видите кои уеб приложения са налични от кои вътрешни местоположения.

Наблюдение, извършвано от шаблона за наблюдение на наличността на уеб приложения

По подразбиране шаблонът за наблюдение на наличността на уеб приложения конфигурира следния мониторинг по подразбиране. Можете да промените монитора в страницата Промяна на конфигурацията на шаблона за наблюдение на наличността на уеб приложения.

МОНИТОРИНГ, ИЗВЪРШВАН ОТ УЕБ-ПРИЛОЖЕНИЕТО НАСТРОЙКА

Описание на монитора Стойности по подразбиране

Монитор на уеб приложения - Мониторът е активиран по подразбиране.

- Честота на теста: 10 минути

- Интервал за събиране на данни за производителността: 1 на всеки 10 минути

- Време за изчакване на теста: 45 секунди

- HTTP код на състоянието: 400 (Ще бъде генерирано предупреждение, ако HTTP кодът на състоянието е 400 или повече.)

- Брой последователни пъти, когато даден критерий трябва да се провали, преди да се генерира сигнал: 1

- Генериране на предупреждения от всеки тест: активирано

- Разрешаване на пренасочвания: разрешено

- HTTP версия: HTTP / 1.1

- HTTP метод: GET

- HTTP заглавки: приемете "/"

- HTTP заглавки: приемете езика на вашия продукт

- HTTP заглавки: приемете кодиране GZIP

Събиране на данни за ефективността - Времето за реакция на транзакцията: активирано

- Време за реакция: активирано

- Време за TCP връзка: активирано

- Време до първия байт: активирано

- Време до последния байт: активирано

- Време за DNS резолюция: активирано

- Размер на съдържанието: активиран

- Време за съдържание: активирано

- Време за изтегляне: активирано

Преглед на данните от мониторинга

Всички данни, събрани от шаблона за наблюдение на наличността на уеб приложения, се показват в папката за наблюдение на наличността на уеб приложения в папката за мониторинг на приложения в навигационния екран за наблюдение. Папката за наблюдение на наличността на приложения съдържа изгледи по подразбиране и подпапки, които предоставят състояние на теста, състояние на уеб приложението и предупреждения, свързани с наблюдаваните тестове. Използвайки изгледа Test State, можете да видите състоянието на теста на отделните тестове. Състоянието на всеки обект съвпада със състоянието на целевия обект, който има най-лошото здравословно състояние, така че да виждате най-лошото състояние на работещите монитори. Ако един или повече от тестовете са показани с грешка, докато поне един друг тест е здрав, това може да посочи проблем за конкретното място на теста. Ако всички компоненти са нездравословни, това може да означава проблем със самото уеб приложение.

Автентикация и оторизация на WEB API

Удостоверяване и упълномощаване в уеб API

Обратно към: Уроци за ASP.NET Web API за начинаещи и професионалисти

Удостоверяване и упълномощаване в уеб API

В тази статия ще обсъдя удостоверяване и упълномощаване в уеб API. Тук ще ви дам общ преглед на удостоверяването и упълномощаването в уеб API и от следващата статия нататък ще обсъдим практическото изпълнение на удостоверяването и упълномощаването в уеб API на ASP.NET с примери.

След като създадете услуга за уеб API, най-важното нещо, за което трябва да се погрижите, е средствата за сигурност, от които се нуждаете, за да контролирате достъпа до услугите си за уеб API. Така че нека започнем дискусията с определението за удостоверяване и упълномощаване

Удостоверяването е процес на идентифициране на потребителя. Например, един потребител, да речем, че Джеймс влиза с потребителското си име и парола, а сървърът използва неговото потребителско име и парола за удостоверяване на Джеймс.

Упълномощаването е процес на вземане на решение дали на удостоверения потребител е разрешено да извърши действие върху определен ресурс (ресурс на уеб API) или не. Например Джеймс (който е удостоверен потребител) има разрешение да получи ресурс, но няма разрешение да създаде ресурс.

Удостоверяване в уеб API

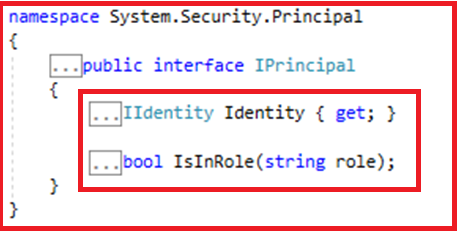
Услугата за уеб API приема, че процесът на удостоверяване трябва да се случи в хост сървъра и обикновено хостваме услугата за уеб API в IIS. IIS сървърът използва HTTP модулите за проверка на удостоверяването на потребител. Можете да конфигурирате проекта си да използва който и да е от вградените модули за удостоверяване, които се предлагат в IIS или ASP.NET, или можете също да създадете свой собствен HTTP модул за извършване на персонализирано удостоверяване.

Когато хостът (IIS сървър) удостоверява потребителя, той обикновено създава основен обект (т.е. IPrincipal обект), под който ще се изпълнява кодът. И така, след като се създаде основният обект (IPrincipal object), тогава хостът (т.е. IIS Server) прикачва този основен обект към текущата нишка, като задава Thread.CurrentPrincipal.

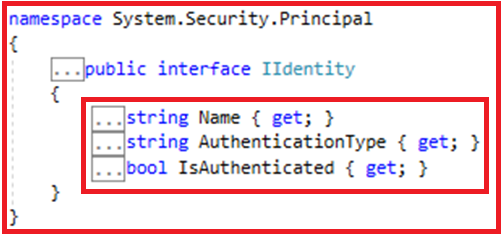
Ако в момента сте объркани за това как е създаден обектът на принципала и как главният обект е прикрепен към текущата нишка, тогава не се притеснявайте, ще обсъдим всички тези неща по-подробно в предстоящите ни статии. В тази статия просто ще ви дам преглед на това как се случват удостоверяването и упълномощаването в услугите на уеб API.

Разбиране на основния обект

Основният обект съдържа две неща, едното е обектът Identity, който всъщност съдържа информацията за потребителя, а другото е свойството IsInRole, което е булево свойство и това свойство е зададено на true, ако потребителят е назначен с други роли false. Следващата диаграма показва дефиницията на IPrincipal интерфейс.



Let’s have a look at the Identity interface definition which contains the user’s information.



Обектът за идентичност, който е свойство на основния обект, съдържа три свойства, т.е. име (тип низ), тип удостоверяване (тип низ) и IsAuthenticated (булев тип). Ако потребителят е удостоверен, тогава свойството Identity.IsAuthenticated ще върне true else false. Свойството Name на обекта Identity ще съхранява името на Identity, като цяло идентификацията не е нищо друго освен вписаното потребителско име. По същия начин свойството AuthenticationType връща типа на удостоверяване, използван за идентифициране на потребителя.

Интерфейсът Identity обикновено се реализира от класовете GenericIdentity и WindowsIdentity. Ще обсъдим как тези класове прилагат интерфейса IIdentity в предстоящата ни статия.

HTTP манипулатори на съобщения за удостоверяване в уеб API

Вместо да използвате хоста (т.е. IIS сървър, където се хоства услугата Web API) за удостоверяване, можете също да напишете логиката за удостоверяване в персонализиран HTTP манипулатор на съобщения. В този случай HTTP манипулаторът на съобщения ще провери входящата HTTP заявка за удостоверяване на потребителя и след това ще зададе основния обект.

Разлики HTTP манипулатор на съобщения спрямо HTTP модул:

HTTP модулът вижда всички входящи заявки, които минават през конвейера ASP.NET, докато манипулаторът на съобщения вижда само входящите заявки, които са насочени към услугата за уеб API.

Също така е възможно да изберете конкретен HTTP манипулатор на съобщения и след това можете да използвате този специфичен манипулатор на HTTP съобщения за удостоверяване за определен маршрут. HTTP модулите са специфични за IIS, докато HTTP манипулаторите на съобщения могат да се използват както с уеб хостинг (в рамките на сървър), така и със самохостинг (в приложение).

HTTP модулите участват в IIS регистриране, одит и т.н. Като цяло, ако не искате да поддържате самохостинг, тогава HTTP модулът е по-добър вариант, но ако искате да поддържате самохостинг, тогава HTTP Message Handler е по-добър вариант.

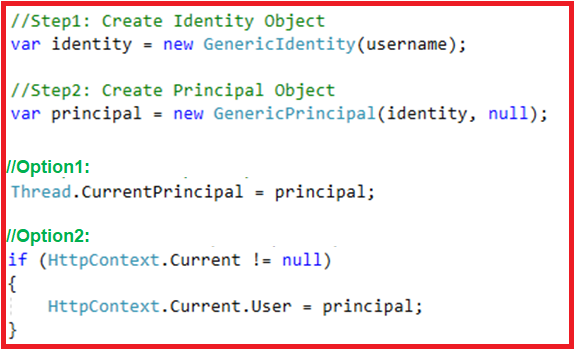
Задаване на основния обект

Ако ще внедрите своя собствена потребителска логика за удостоверяване на потребителя, тогава можете да зададете основния обект на две места, които са както следва:

Тема. Текуща Принцип. Това е стандартният начин за задаване на принципала на нишката в .NET.

HttpContext.Current.User. Това свойство е специфично за ASP.NET.

Следващото изображение показва как да създадете и зададете основния обект с текущата нишка. Тук ви показвам и двете опции за задаване на основния обект.

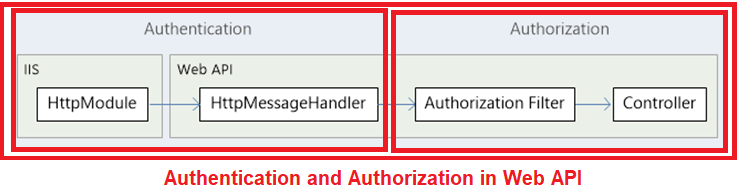


Ако ще хоствате услугата Web API в IIS, тогава трябва да зададете основния обект и на двете места поради опасенията за сигурността, т.е.защитата става несъвместима. В случай на самостоятелно хостване стойността на HttpContext.Current е нула. За да сте сигурни, че кодът ви е агностичен на хоста (т.е. за да поддържа както уеб хостинг, така и самохостинг), трябва да проверите за нула, преди да присвоите основния обект на HttpContext.Current, както е показано на горното изображение.

Упълномощаване в уеб API

Процесът на упълномощаване ще се случи преди изпълнението на метода за действие на контролера, който ви предоставя гъвкавостта да решите дали искате да предоставите достъп до този ресурс или не.

Можем да приложим това в ASP.NET Web API, като използваме филтрите за упълномощаване, които ще бъдат изпълнени преди изпълнението на метода за действие на контролера. Така че, ако заявката не е упълномощена за този конкретен ресурс, тогава филтърът връща отговор на грешка на клиента, без да изпълнява метода на действие на контролера. Следващата диаграма обяснява горното.



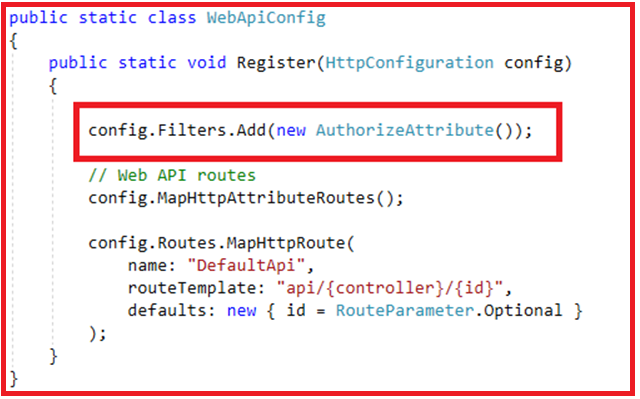
Използване на атрибута [Authorize]

ASP.NET Web API Framework осигурява вграден атрибут за филтър за оторизация, т.е. В противен случай той просто връща кода на състоянието HTTP 401 Unauthorized, без да извиква метода за действие на контролера.

Можете да приложите горния вграден филтър глобално, на ниво контролер или на ниво действие.

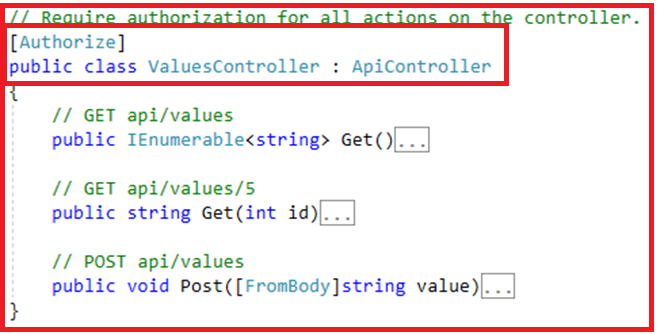
В глобален мащаб:

Ако искате да проверите удостоверяването за всички контролери на Web API, по-добре е да добавите филтъра AuthorizeAttribute към списъка на глобалния филтър в метода Register на класа WebApiConfig, както е показано на изображението по-долу:



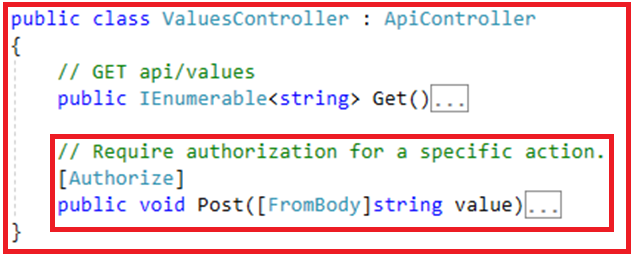
На ниво контролер:

Ако искате да осигурите удостоверяване за всички методи за действие на конкретен контролер, тогава е по-добре и препоръчително да добавите филтъра за разрешаване на ниво контролер, както е показано на изображението по-долу.

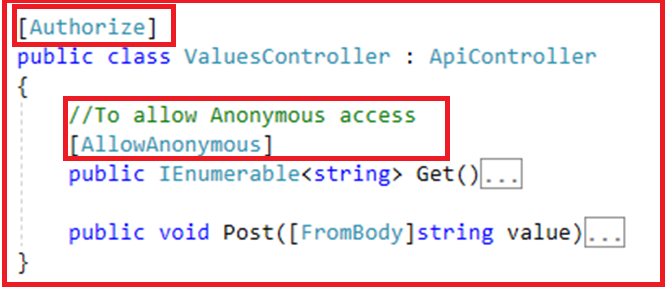


На ниво действие:

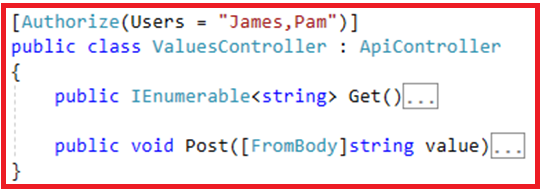
Ако искате да осигурите удостоверяване за конкретни методи за действие на контролер, тогава е по-добре да добавите атрибута Authorize filter към метода на действие, който изисква удостоверяване, както е показано на изображението по-долу.



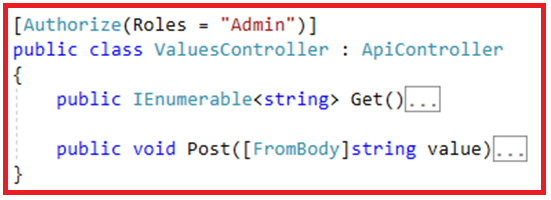
Друг начин да направите това е да ограничите контролера, като декорирате контролера с атрибут Authorize filter и след това да разрешите анонимен достъп до методите за действие, който не изисква удостоверяване, като използвате атрибута AllowAnonymous. В примера по-долу методът Post е ограничен, но методът Get позволява анонимен достъп.



Към момента обсъдихме две неща. Ако искаме да проверим удостоверяването, преди да извикаме метода на действие, трябва да използваме вградения атрибут за разрешаване на филтъра. Ако искаме някой метод за действие да бъде достъпен от анонимните потребители, тогава трябва да украсим този метод за действие с атрибута AllowAnonymous. По пътя можем също да ограничим достъпа до конкретни потребители или до потребители с конкретни роли.

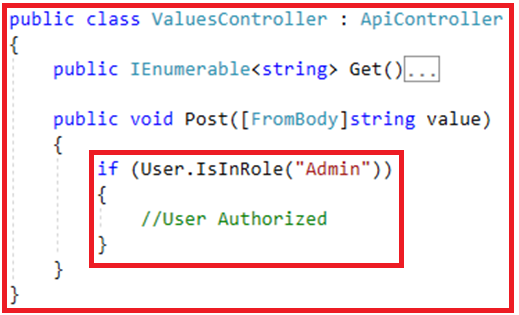
Ограничаване от потребители:

###### **Restrict by Roles:**



**Забележка: Въпросът, който трябва да запомните тук, е, че филтърът AuthorizeAttribute за уеб API се намира в пространството от имена System.Web.Http. В MVC има и филтър AuthorizeAttribute, който се намира в пространството от имена System.Web.Mvc, което не е съвместимо с контролерите на Web API.**

**Упълномощаване вътре в действие на контролера**

**В някои сценарии може да разрешите дадена заявка да продължи, но трябва да промените поведението въз основа на принципала. Например информацията, която ще върнете от действието, зависи от ролята на потребителя. В рамките на метода за действие на контролера можете да получите текущия главен обект от свойството ApiController.User е показано на изображението по-долу.**

Сигурността на уеб приложенията е централен компонент на всеки уеб-базиран бизнес. Глобалният характер на Интернет излага уеб свойствата да атакуват от различни места и от различни нива на мащаб и сложност. Защитата на уеб приложенията се занимава специално със защитата около уебсайтове, уеб приложения и уеб услуги като API.

Сигурност на уеб приложенията и добре известни атаки

Какви са често срещаните уязвимости в сигурността на уеб приложенията?

Атаките срещу уеб приложения варират от целенасочена манипулация на база данни до мащабно прекъсване на работата на мрежата. Нека разгледаме някои от често използваните методи за атака или „вектори“.

Скриптиране на различни сайтове (XSS) - XSS е уязвимост, която позволява на нападателя да инжектира скриптове от страна на клиента в уеб страница, за да получи директен достъп до важна информация, да се представя за потребителя или да подведе потребителя да разкрие важна информация.

SQL инжекция (SQi) - SQi е метод, чрез който атакуващият използва уязвимости в начина, по който базата данни изпълнява заявки за търсене. Атакуващите използват SQi, за да получат достъп до неоторизирана информация, да модифицират или създадат нови потребителски разрешения или по друг начин да манипулират или унищожат чувствителни данни.

Атаки за отказ на услуга (DoS) и разпределени атаки за отказ на услуга (DDoS) - Чрез различни вектори атакуващите могат да претоварят целевия сървър или околната инфраструктура с различни видове трафик на атаки. Когато сървърът вече не е в състояние ефективно да обработва входящите заявки, той започва да се държи вяло и в крайна сметка да отказва услуга на входящите заявки от легитимни потребители.

Повреда на паметта - Повреда на паметта възниква, когато дадено местоположение в паметта е неволно модифицирано, което води до потенциално неочаквано поведение в софтуера. Лошите актьори ще се опитат да подушат и да използват корупцията на паметта чрез експлойти като инжекции на код или атаки за преливане на буфер.

Препълване на буфера - Препълването на буфера е аномалия, която възниква, когато софтуерът записва данни в определено пространство в паметта, известно като буфер. Препълването на капацитета на буфера води до презаписване на съседни места в паметта с данни. Това поведение може да се използва за инжектиране на злонамерен код в паметта, потенциално създавайки уязвимост в целевата машина.

Подправяне на заявки между сайтове (CSRF) - Подправянето на заявки между сайтове включва подвеждане на жертва да направи заявка, която използва тяхното удостоверяване или упълномощаване. Използвайки привилегиите на акаунта на даден потребител, нападателят може да изпрати заявка, маскирана като потребител. След като акаунтът на потребителя бъде компрометиран, нападателят може да проникне, да унищожи или модифицира важна информация. Силно привилегированите акаунти като администратори или ръководители обикновено са насочени.

Нарушаване на данни - различно от конкретните вектори на атака, нарушаването на данни е общ термин, отнасящ се до освобождаването на чувствителна или поверителна информация и може да възникне чрез злонамерени действия или по грешка. Обхватът на това, което се счита за нарушаване на данни, е доста широк и може да се състои от няколко изключително ценни записи до милиони открити потребителски акаунти.

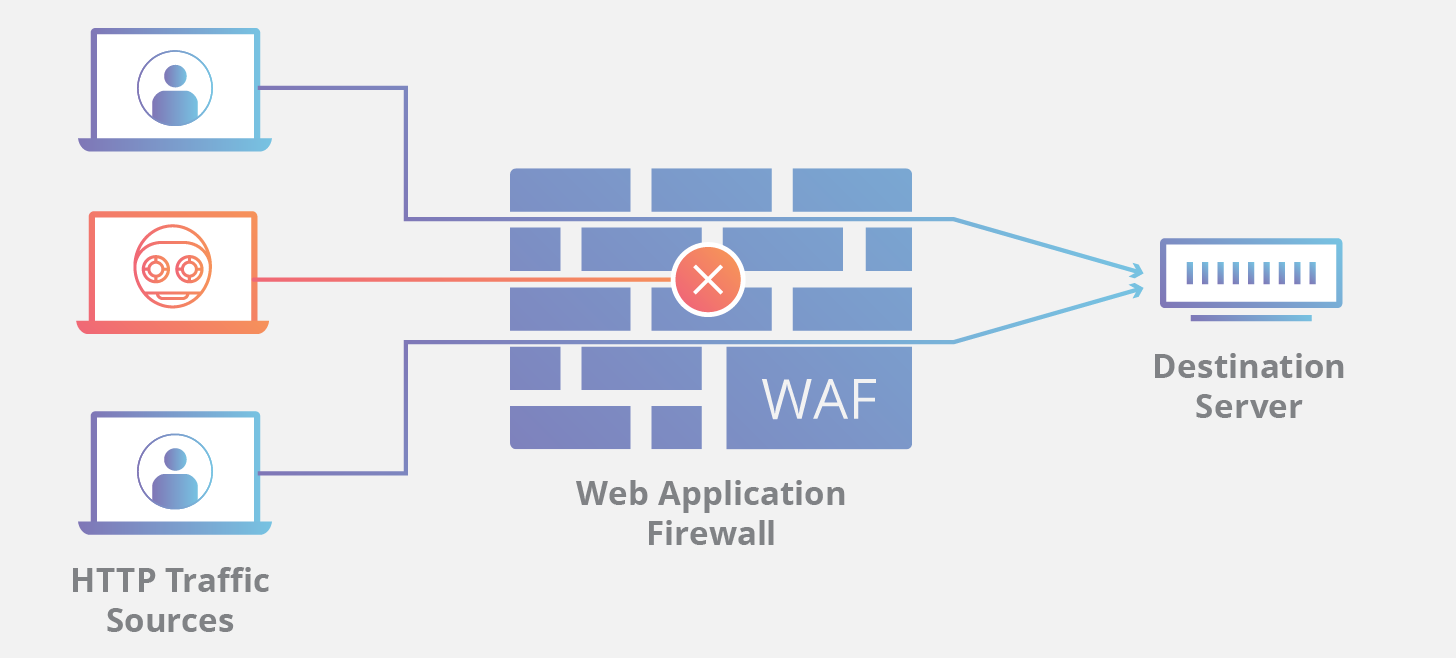
Кои са най-добрите практики за намаляване на уязвимостите?

Важните стъпки за защита на уеб приложенията от експлоатация включват използване на актуално криптиране, изискване за правилно удостоверяване, непрекъснато закърпване на открити уязвимости и добра хигиена за разработване на софтуер. Реалността е, че умните нападатели могат да открият уязвимости дори в доста стабилна среда за сигурност и се препоръчва цялостна стратегия за сигурност.

Защитата на уеб приложенията може да бъде подобрена чрез защита срещу DDoS, слой от приложения и DNS атаки:

WAF - Защитена срещу атаки на слой приложения

Защитна стена за уеб приложения или WAF помага да се защити уеб приложение от злонамерен HTTP трафик. Чрез поставяне на филтрационна бариера между целевия сървър и нападателя, WAF е в състояние да защити срещу атаки като фалшифициране на различни сайтове, скриптове на различни сайтове и SQL инжектиране. Научете повече за WAF на Cloudflare.



Смекчаване на DDoS

Често използван метод за разрушаване на уеб приложение е използването на разпределени атаки за отказ на услуга или DDoS. Cloudflare смекчава DDoS атаките чрез различни стратегии, включително отпадане на обемни атаки на нашия ръб и използване на нашата мрежа Anycast за правилно насочване на легитимни заявки без загуба на услуга. Научете как Cloudflare може да ви помогне да защитите уеб собственост от DDoS атака.

DNS security - DNSSEC protection

Системата с имена на домейни или DNS е телефонният указател на Интернет и представлява начина, по който интернет инструмент, като уеб браузър, търси правилния сървър. Лошите актьори ще се опитат да отвлекат този DNS процес на заявка чрез отравяне на DNS кеш памет, атаки по пътя и други методи за намеса в жизнения цикъл на DNS търсене. Ако DNS е телефонният указател на Интернет, тогава DNSSEC е идентификатор на повикващия. Разгледайте как можете да защитите DNS търсене с помощта на Cloudflare.

Изпитна тема № 17:

Интернет програмиране План-тезис:

Създаване на REST API. Извикване на REST заявки с JavaScript и AJAX. Внедряване на проект (deployment). Примерна приложна задача: По време на теоретичния изпит се предоставя непълен/неработещ/некоректен програмен фрагмент на приложната задача. Предоставеният фрагмент да се приведе в работещ вид. Условие: Да се идентифицират проблеми в приложния програмен интерфейс и да се отстранят.

REST API

# **REST**

REST (на английски: Representational State Transfer) е стил софтуерна архитектура за реализация на уеб услуги. Основната идея е да се определи системен ресурс, който се променя в резултат на взаимодействието между доставчика на услуги и потребителя. Архитектурният модел REST включва взаимодействията между сървър и клиент, осъществени по време на трансфера на данни. Концепцията беше въведена за пръв път от Рой Филдинг през 2000 г. като част от неговата докторска дисертация. Филдинг е един от основните автори на HTTP протокола, под който се изпълняват REST имплементациите в повечето случаи.

## **Условия**

Архитектурният стил на „REST“ прилага шест ограничителни условия, като същевременно дава свобода за дизайна и имплементацията на индивидуалните компоненти:

**Клиент-сървър**

Единният интерфейс разделя клиента и сървъра. Това означава, например, че клиента не се грижи за складирането на данни. Тази задача остава изцяло за сървъра, като по този начин се подобрява портативността на клиентския код (може да се използва в различни среди). Сървърът няма връзка с потребителския интерфейс и по този начин е по-семпъл и лесен за премащабиране. Клиентът и сървърът могат да бъдат заменяни или развивани независимо един от друг, стига това да не налага промяна на единния интерфейс помежду им.

**Без статус на сесията *(на английски: Stateless)***

Следващото условие е на сървъра да не се запазват статуси на сесиите. Всяка заявка от клиента, съдържа в себе си нужната информация за нейната обработка, статуси на сесии се запазват единствено при клиента.

**Кеширане**

Клиентът има право да кешира (запазва) информация, получена в отговор от сървъра, за да я преизползва при последващи заявки. За тази цел сървърът трябва имплицитно или експлицитно да е посочил дали информацията в отговора може да се кешира, за да се избегнат случаи, в които клиентът получава грешна информация при бъдещи заявки. При правилно управление и използване на кеширането могат частично или напълно да се елиминират ненужни взаимодействия между клиента и сървъра, като по този начин се подобрява бързината и производителността.

**Многослойна система**

Обикновено клиентът не знае дали е свързан с крайния сървър или със сървър-посредник. Сървърите-посредници подобряват ефективността, като увеличават капацитета за обработване на заявки и предоставят споделени кешове. Също така те допринасят да подобряването на сигурността.

**Код при поискване (незадължително)**

Сървърът може временно да разшири функционалността, изпращайки код, който се изпълнява директно при клиента. Например клиентски скриптове, написани на JavaScript или компилирани компоненти като Java applets.

**Единен интерфейс**

Единният интерфейс между клиента и сървъра разделя и опростява архитектурата. По този начин всеки компонент може да се развива самостоятелно.

Единственото условие на REST архитектурата, което не е задължително е „Код по поискване“. Всяко приложение (услуга), изпълняващо на гореописаните условия, може да се нарече „RESTful“. Ако нарушава дори едно от условията, то не може да бъде считано за „RESTful“.

Всяка разпространена хипермедийна система, съответстваща на архитектурния стил на „REST“ притежава нужната производителност, мащабируемост, опростеност, гъвкавост, видимост, портативност и надеждност.

## **RESTful API**

RESTful уеб API (също наричано RESTful уеб service) е уеб приложение, което използва принципите на HTTP и REST. Представлява колекция от ресурси с четири дефинирани аспекта:

* Основният „URL“ за уеб приложенията като http://example.com/resources/
* • Internet media типът на данните поддържани от уеб приложенията. Това най-често е JSON, но може да бъде всеки друг валиден Интернет медиен тип, като се има предвид, че е валиден хипертекст стандарт.
* Операции поддържани от уеб приложението използвайки HTTP методи (примерно: GET, PUT, POST, или DELETE).
* Приложенията трябва да се задвижват от хипертекст.

Таблицата показва как HTTP методите обикновено се използват, за да се създаде уеб приложение.

Методите PUT и DELETE са Idempotent\_methods\_and\_web\_applications idempotent methods. Методът GET е безопасен метод, което означава, че извикването му не причинява странични ефекти.

Интерфейс

Единният интерфейс на REST се счита за основа на дизайна на всяка REST услуга.

**Идентификация на ресурсите**

Отделните ресурси се разпознават по заявките (например използвайки [URIs](https://bg.wikipedia.org/w/index.php?title=URIs&action=edit&redlink=1) в уеб-базирани REST системи). Самите ресурсите са отделни от изображението, което се изпраща на клиента. Например сървърът вместо да изпраща цялата база данни, изпраща HTML, XML или JSON, които представляват някакви записи в нея.

**Управление на ресурс чрез изображение**

Имайки изображение на ресурса, клиента има достатъчно информация, с която може да променя или трие ресурсите от сървъра, в случай че има разрешението да го направи.

**Самоописващи съобщения**

Всяко съобщение включва информация, която описва как да се обработи съобщението.

**HATEOAS (Hypermedia as the Engine of Application State)**

[HATEOAS](https://bg.wikipedia.org/w/index.php?title=HATEOAS&action=edit&redlink=1) е ограничението, което отличава архитектурата на REST приложението от повечето архитектури на мрежови приложения. При нея клиента „общува“ с приложението изцяло чрез [hypermedia](https://bg.wikipedia.org/w/index.php?title=Hypermedia&action=edit&redlink=1), получена динамично от сървъра.

Примери за използване на REST

През последните години REST е широко използван в контекста на Web 2.0 приложения и внедряването на уеб услуги и SOA. Следват някои примери за неговата употреба:

* [Atom Publishing Protocol](https://bg.wikipedia.org/w/index.php?title=Atom_Publishing_Protocol&action=edit&redlink=1) – Протокол, използван за абониране за проследяване на промени в уебсайт, подобен на RSS.
* [Azure](https://bg.wikipedia.org/w/index.php?title=Azure&action=edit&redlink=1) – Имплементира изчислителни облаци на Microsoft въз основа на REST, XML и HTTP.
* Sun Microsystems е добър пример за документация, тип медия.
* Инициативата Open Services for Lifecycle Collaboration (OSLC)  създава RESTful подход към обединяване на софтуерни продукти.
* [CouchDB](https://bg.wikipedia.org/w/index.php?title=CouchDB&action=edit&redlink=1) е документно ориентирана база данни написана в Erlang, която предоставя RESTful JSON API, който може да достъпен от всяка среда, която поддържа HTTP молби.
* [MySQL Cluster](https://bg.wikipedia.org/w/index.php?title=MySQL_Cluster&action=edit&redlink=1) база данни достижима през прост REST/JSON []](https://bg.wikipedia.org/wiki/REST#cite_note-4) интерфейс като Apache модул.
* Canonical REST Entity Service на Microsoft.
* [Nuxeo](https://bg.wikipedia.org/w/index.php?title=Nuxeo&action=edit&redlink=1), open sourse документ мениджър, създава Content Automation интерфейс чрез REST API.
* [Restful Objects](https://bg.wikipedia.org/w/index.php?title=Restful_Objects&action=edit&redlink=1), публична спецификация за общ RESTful API за всеки домейн обект модел.
* [Sones GraphDB](https://bg.wikipedia.org/w/index.php?title=Sones_GraphDB&action=edit&redlink=1) е графа – ориентирана база данни написана на C#, която предоставя RESTful интерфейс.
* Поддръжката на Google Fusion Tables включва RESTful API.

Създаване на REST API

Как да проектирам REST API

Изучаването на REST на парчета е едно, докато прилагането на всички тези научени концепции в реалния дизайн на приложения е напълно друго предизвикателство. В този урок ще се научим да проектираме REST API за мрежово приложение. Моля, обърнете внимание, че извеждането от цялото това упражнение е научаването как да се прилагат принципите REST в процеса на проектиране.

**Steps in designing REST Services**

Identify Object Model

Create Model URIs

Determine Representations

Assign HTTP Methods

More Actions

**Стъпки при проектирането на REST услуги**

Идентифицирайте обектния модел

Създайте URI на модела

Определете представянията

Присвояване на HTTP методи

Повече действия

**Идентифицирайте обектния модел**

**(Identify Object Model)**

* Първата стъпка при проектирането на приложение, базирано на REST API, е - идентифициране на обектите, които ще бъдат представени като ресурси.
* За мрежово приложение обектното моделиране е доста по-лесно. Може да има много неща като устройства, управлявани обекти, рутери, модеми и др. За по-голяма простота ще разгледаме само два ресурса, т.е.Devices

• Конфигурации

Тук конфигурацията е подресурс на устройство. Устройството може да има много опции за конфигуриране.

Обърнете внимание, че и двата обекта / ресурси в нашия по-горе модел ще имат уникален идентификатор, който е свойството за цяло число id.

**Създайте URI на модела**

Сега, когато обектният модел е готов, е време да решим ресурсите URI. На тази стъпка, докато проектирате URI на ресурсите - фокусирайте се върху връзката между ресурсите и техните подресурси. Тези ресурсни URI са крайни точки за RESTful услуги.

В нашето приложение устройството е ресурс от най-високо ниво. А конфигурацията е подресурс под устройството. Нека запишем URI адресите.

|  |
| --- |
| /devices  /devices/{id}    /configurations  /configurations/{id}    /devices/{id}/configurations  /devices/{id}/configurations/{id} |

Notice that these **URIs do not use any verb or operation**. It’s crucial not to include any verb in URIs. URIs should all be nouns only.

**Определете представянията**

Сега, когато URI на ресурсите са решени, нека да работим върху техните представяния. Представянията са дефинирани предимно във формат XML или JSON. Ще видим XML примери като по-изразителен за начина на съставяне на данните.

Събиране на ресурс на устройството

Когато връщате колекционен ресурс, включвайте само най-важната информация за ресурсите. Това ще поддържа размера на полезния товар малък и така ще подобри производителността на REST API.

|  |
| --- |
| <devices size="2">        <link rel="self" href="/devices"/>        <device id="12345">          <link rel="self" href="/devices/12345"/>          <deviceFamily>apple-es</deviceFamily>          <OSVersion>10.3R2.11</OSVersion>          <platform>SRX100B</platform>          <serialNumber>32423457</serialNumber>          <connectionStatus>up</connectionStatus>          <ipAddr>192.168.21.9</ipAddr>          <name>apple-srx\_200</name>          <status>active</status>      </device>        <device id="556677">          <link rel="self" href="/devices/556677"/>          <deviceFamily>apple-es</deviceFamily>          <OSVersion>10.3R2.11</OSVersion>          <platform>SRX100B</platform>          <serialNumber>6453534</serialNumber>          <connectionStatus>up</connectionStatus>          <ipAddr>192.168.20.23</ipAddr>          <name>apple-srx\_200</name>          <status>active</status>      </device>    </devices> |

**Ресурс за едно устройство**

За разлика от URI на колекцията, тук включва пълна информация за устройство в този URI. Тук включете и списък с връзки за подресурси и други поддържани операции. Това ще направи вашия REST API HATEOAS управляван.

|  |
| --- |
| <device id="12345">      <link rel="self" href="/devices/12345"/>        <id>12345</id>      <deviceFamily>apple-es</deviceFamily>      <OSVersion>10.0R2.10</OSVersion>      <platform>SRX100-LM</platform>      <serialNumber>32423457</serialNumber>      <name>apple-srx\_100\_lehar</name>      <hostName>apple-srx\_100\_lehar</hostName>      <ipAddr>192.168.21.9</ipAddr>      <status>active</status>        <configurations size="2">          <link rel="self" href="/configurations" />            <configuration id="42342">              <link rel="self" href="/configurations/42342" />          </configuration>            <configuration id="675675">              <link rel="self" href="/configurations/675675" />          </configuration>      </configurations>        <method href="/devices/12345/exec-rpc" rel="rpc"/>      <method href="/devices/12345/synch-config"rel="synch device configuration"/>  </device> |

**Колекция ресурси за конфигуриране**

Подобно на представяне на колекция от устройства, създайте представяне на колекция от конфигурации само с минимална информация.

|  |
| --- |
| <configurations size="20">      <link rel="self" href="/configurations" />        <configuration id="42342">          <link rel="self" href="/configurations/42342" />      </configuration>        <configuration id="675675">          <link rel="self" href="/configurations/675675" />      </configuration>      ...      ...  </configurations> |

Моля, имайте предвид, че configurations collection representation inside device е подобна на top-level configurations URI. Единствената разлика е че configurations-ите за устройство са само два, така че само два елемента от конфигурацията са изброени като подресурс под устройство.

Ресурс за единична конфигурация

Сега едно представяне на ресурс за конфигурация трябва да съдържа цялата възможна информация за този ресурс - включително съответните връзки.

|  |
| --- |
| <configuration id="42342">      <link rel="self" href="/configurations/42342" />      <content><![CDATA[...]]></content>      <status>active</status>      <link  rel="raw configuration content" href="/configurations/42342/raw" />  </configuration> |

**Събиране на ресурси за конфигуриране под едно устройство**

Тази колекция от ресурси на конфигурации ще бъде подмножество на основната колекция от конфигурации и ще бъде специфична само за устройство. Тъй като това е подмножеството на първичната колекция, НЕ създавайте различни полета за данни за представяне от първичната колекция. Използвайте същите презентационни полета като основната колекция.

|  |
| --- |
| <configurations size="2">      <link rel="self" href="/devices/12345/configurations" />        <configuration id="53324">          <link rel="self" href="/devices/12345/configurations/53324" />          <link rel="detail" href="/configurations/53324" />      </configuration>        <configuration id="333443">          <link rel="self" href="/devices/12345/configurations/333443" />          <link rel="detail" href="/configurations/333443" />      </configuration>  </configurations> |

Забележете, че тази подресурсна колекция има две връзки. Един за директното му представяне в подколекцията i.e. /devices/12345/configurations/333443 и други, сочещи мястото му в първичната колекция, i.е.. /configurations/333443.

Наличието на две връзки е от съществено значение, тъй като можете да осигурите достъп до специфична за устройството конфигурация по по-уникален начин и ще имате възможност да маскирате някои полета (ако дизайнът го изисква), които няма да се виждат във вторична колекция.

**Ресурс за единична конфигурация под Единично устройство**

Това представяне трябва да има или точно подобно представяне като представяне на конфигурацията от първичната колекция, ИЛИ може да маскирате няколко полета.

Това представяне на подресурс също ще има допълнителна връзка към основното му представяне.

|  |
| --- |
| <configuration id="11223344">      <link rel="self" href="/devices/12345/configurations/11223344" />      <link rel="detail" href="/configurations/11223344" />      <content><![CDATA[...]]></content>      <status>active</status>      <link rel="raw configuration content" href="/configurations/11223344/raw" />  </configuration> |

Сега, преди да преминем към следващия раздел, нека отбележим няколко наблюдения, за да не ги пропуснете.Resource URIs are all nouns.

• URI обикновено са в две форми - събиране на ресурси и единичен ресурс.

• Събирането може да бъде в две форми първично и вторично събиране. Вторичната колекция е под-колекция само от първична колекция.

• Всеки ресурс / колекция съдържа поне една връзка, т.е. към себе си.

• Колекциите съдържат само най-важната информация за ресурсите.

• За да получите пълна информация за ресурс, трябва да осъществите достъп само чрез неговия специфичен URI на ресурса.

• Представителствата могат да имат допълнителни връзки (т.е. методи в едно устройство). Тук методът представлява метод POST. Можете да имате повече атрибути или да формирате връзки по съвсем нов начин.

• Все още не сме говорили за операции с тези ресурси.

**Присвояване на HTTP методи**

Така че нашите ресурсни URI и тяхното представяне са фиксирани сега. Нека да решим възможните операции в приложението и да картографираме тези операции върху URI на ресурсите. Потребител на нашето мрежово приложение може да извършва операции за сърфиране, създаване, актуализиране или изтриване. Така че нека ги назначим.

**Преглед на всички устройства или конфигурации** **[Primary Collection] [Основна колекция]**

|  |
| --- |
| HTTP GET /devices  HTTP GET /configurations |

If the collection size is large, you can apply paging and filtering as well. e.g., Below requests will fetch the first 20 records from collection.

|  |
| --- |
| HTTP GET /devices?startIndex=0&size=20  HTTP GET /configurations?startIndex=0&size=20 |

**Преглед на всички устройства или конфигурации [Secondary Collection]**

|  |
| --- |
| HTTP GET /devices/{id}/configurations |

It will be mostly a small size collection – so no need to enable filtering or soring here.

**Преглед на едно устройство или конфигурация [Основна колекция]**

To get the complete detail of a device or configuration, use GET operation on singular resource URIs.

|  |
| --- |
| HTTP GET /devices/{id}  HTTP GET /configurations/{id} |

**Преглед на едно устройство или конфигурация [Secondary Collection]**

|  |
| --- |
| HTTP GET /devices/{id}/configurations/{id} |

Представянето на подресурса ще бъде същото като или подмножество на първичното представяне.

**Създайте устройство или конфигурация**

Създаването не е идемпотентна операция, а в HTTP протокола - POST също не е идемпотентна. Така че използвайте POST.

|  |
| --- |
| HTTP POST /devices  HTTP POST /configurations |

Моля, обърнете внимание, че полезният товар на заявката няма да съдържа никакъв атрибут id, тъй като сървърът е отговорен за решението му. Отговорът за създаване на заявка ще изглежда така:

|  |
| --- |
| HTTP/1.1 201 Created  Content-Type: application/xml  Location: <http://example.com/network-app/configurations/678678>    <configuration id="678678">      <link rel="self" href="/configurations/678678" />      <content><![CDATA[...]]></content>      <status>active</status>      <link  rel="raw configuration content" href="/configurations/678678/raw" />  </configuration> |

**Актуализирайте устройство или конфигурация**

Операцията за актуализация е идемпотентна операция и HTTP PUT също е идемпотентен метод. Така че можем да използваме метода PUT за операции по актуализиране.

|  |
| --- |
| HTTP PUT /devices/{id}  HTTP PUT /configurations/{id} |

PUT response may look like this.

|  |
| --- |
| HTTP/1.1 200 OK  Content-Type: application/xml    <configuration id="678678">      <link rel="self" href="/configurations/678678" />      <content><![CDATA[. updated content here .]]></content>      <status>active</status>      <link  rel="raw configuration content" href="/configurations/678678/raw" />  </configuration> |

**Премахнете устройство или конфигурация**

**Премахването винаги е с операция Delete.**

|  |
| --- |
| HTTP DELETE /devices/{id}  HTTP DELETE /configurations/{id} |

A successful response SHOULD be 202 (Accepted) if resource has been queues for deletion (async operation), or 200 (OK) / 204 (No Content) if resource has been deleted permanently (sync operation).

В случай на асинхронна операция, приложението трябва да върне идентификатор на задача, който може да бъде проследен за състояние на успех / неуспех.

Моля, обърнете внимание, че трябва да направите достатъчно анализ при определяне на поведението, когато подресурсът бъде изтрит от системата. Обикновено може да искате да ИЗТРИВАТЕ ресурс в тези заявки - с други думи, задайте състоянието им НЕАКТИВНО. Следвайки този подход, няма да е необходимо да намирате и премахвате референциите и от други места.

**Прилагане или премахване на конфигурация от устройство**

В реално приложение ще трябва да приложите конфигурацията на устройството - ИЛИ може да искате да премахнете конфигурацията от устройството (не от основната колекция). В този случай ще използвате методите PUT и DELETE поради неговия идемпотентен характер.

|  |
| --- |
| //Apply Configuration on a device  HTTP PUT /devices/{id}/configurations    //Remove Configuration on a device  HTTP DELETE /devices/{id}/configurations/{id} |

**Повече действия**

Досега сме проектирали само обектни модели, URI и след това сме решили HTTP методи или операции върху тях. Трябва да работите и върху други аспекти на приложението:

1) Регистрация

2) Сигурност

3) Откриване и т.н.

В следващата статия ще създадем това приложение в Java, за да получим по-подробно разбиране.

Извикване на REST заявки с JavaScript

Всички възможни начини за извършване на API извикване в JavaScript.

В JavaScript беше наистина важно да се знае как да се правят HTTP заявки и да се извличат динамичните данни от сървъра / базата данни.

JavaScript предоставя някои вградени обекти на браузъра и някои външни библиотеки с отворен код за взаимодействие с API.

Ето възможните начини за извършване на API повикване:

1. XMLHttpRequest
2. донеси
3. Аксиос
4. jQuery

XMLHttpRequest

Преди да излезе ES 6, единственият начин да се направи HTTP заявка в JavaScript беше XMLHttpRequest. Това е вграден обект на браузъра, който ни позволява да правим HTTP заявки в JavaScript.

JSONPlaceholder е безплатен онлайн REST API, който можете да използвате, когато имате нужда от фалшиви данни.



По подразбиране получаваме отговора във формата на низове, трябва да анализираме в JSON.

XMLHttpRequest беше остарял в ES 6 чрез въвеждането на fetch. Но все пак използваме XMLHttpRequest, когато трябва да работим със стари браузъри и не искаме полифили.

Fetch

Fetch ви позволява да направите HTTP заявка по подобен начин като XMLHttpRequest, но с ясен интерфейс, като използвате обещания. Не се поддържа от стари браузъри (може да бъде многократно попълнен), но много добре се поддържа сред съвременните. Можем да осъществим API извикване, като използваме извличане по два начина.

Метод 1:



Като използвате Async и Await



API за извличане е много мощен. Можем лесно да изпращаме AJAX заявки, използвайки API за извличане на браузъра. Основният недостатък на API за извличане е обработката на грешки.

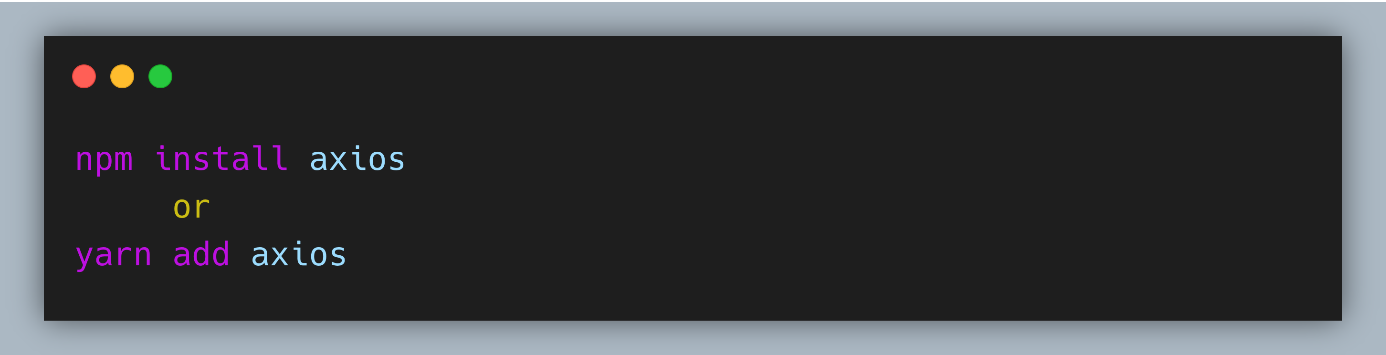
Аксиос - Axios

Axios е библиотека с отворен код за извършване на HTTP заявки и предлага много страхотни функции и работи както в браузъри, така и в Node.js. Това е HTTP клиент, базиран на обещания, който може да се използва в обикновен JavaScript и усъвършенствани рамки като React, Vue.js и Angular.

Той поддържа всички съвременни браузъри, включително поддръжка за IE 8 и по-нова версия.

Инсталация:

Ако използвате някой от мениджърите на пакети като npm или yarn.



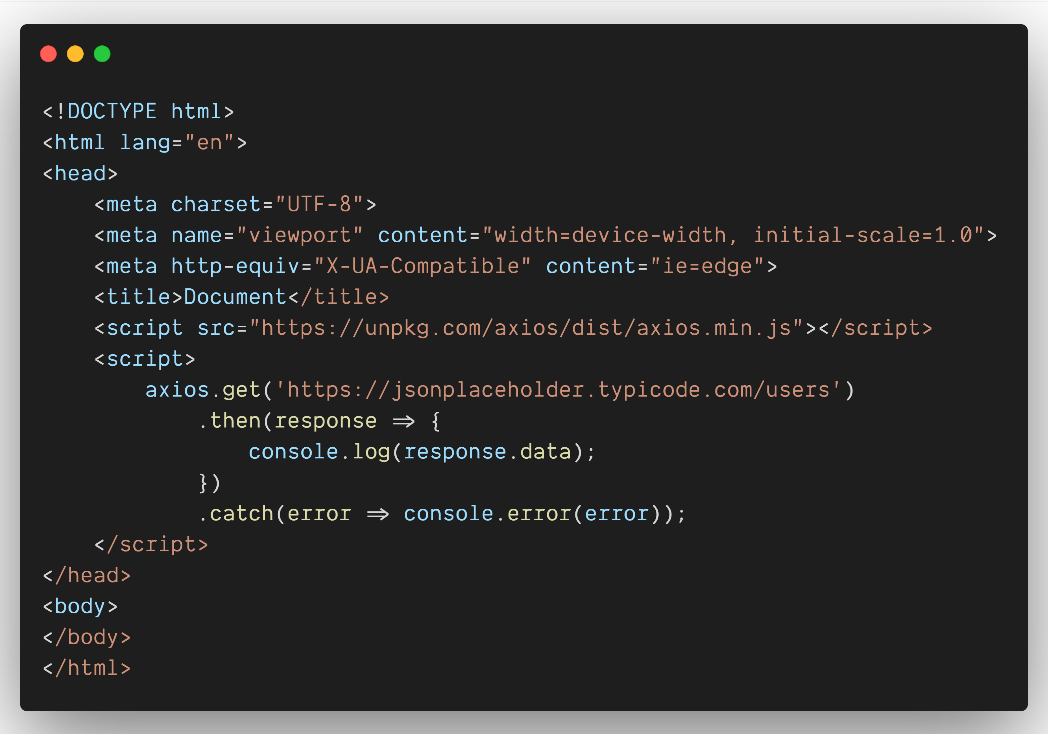
И го включете в HTML файл като този



Най-лесният начин да включите Axios е чрез използване на външен CDN:



Сега можете да започнете да изпращате HTTP заявка, като включите следния скрипт във вашия HTML файл.



Следват предимствата на **Axios**

1. Axios извършва автоматични трансформации и връща данните в JSON формат.
2. По-добро обработване на грешки
3. Axios има широка гама от поддържани браузъри.

jQuery

jQuery има много методи за обработка на асинхронни HTTP заявки. За да използваме jQuery, трябва да включим изходния файл на jQuery и методът $ .ajax () се използва за извършване на HTTP заявката.

$ .ajax ()



Методът $ .ajax отнема много параметри, някои от които са задължителни, а други по избор. Той съдържа две функции за обратно извикване, успех и грешка за обработка на получения отговор.

Заключение

Повечето приложения в реално време използват Axios за отправяне на HTTP заявки. Axios е много лесна и библиотека с отворен код за извършване на HTTP заявки.

Разгледах най-популярните начини за извършване на HTTP заявки в JavaScript.

За ваше улеснение добавям изходния код тук.

<https://github.com/jayanthbabu123/all-possible-ways-making-api-call-javascript>

Никой не е перфектен. Моля, коментирайте всички предложения и подобрения.

Извикване на REST заявки

с JavaScript и AJAX.

Направете първите си API повиквания с JQuery AJAX

Информационната ера свърши: разполагаме с цялата информация сега. Всичко това. Намираме се на уникално място в човешката история, където по някакъв начин сме успели да извлечем повече данни, отколкото знаем какво да правим ... и много от тези данни са лесно достъпни чрез API.

Днес ще си намокрим краката с REST API, като по този начин ще ни позволи да взаимодействаме със значима информация. Извършването на повиквания Ajax GET с JQuery е може би толкова основно, колкото се получава: има голяма вероятност вече да знаем всички тези неща, но дори и аз постоянно забравям точния синтаксис на ежедневните функции. Шансовете са, че ще се върна към този пост по някое време, само за да копирам и поставя фрагментите по-долу.

**Въведение в REST API**

Ако сте нов в изпълнението на заявки AJAX, има вероятност да сте нов в REST API като цяло. Този курс на катастрофа ще бъде бърз и груб по краищата, така че каишка.

В най-простия смисъл крайната точка на REST API е URL. Случва се така, че този URL адрес вероятно очаква повече от вас, отколкото просто да го посетите и в резултат на това трябва да изведе нещо полезно за вас. Крайните точки на API почти винаги извеждат или JSON, или XML; тези отговори ще ви дадат информация, варираща от кодовете за грешки до действителните данни, които търсите.

API на REST очакват заявките да идват в една от следните форми:

* GET: Заявка, търсеща данни само за четене. Някои GET заявки просто трябва да бъдат копирани и поставени в прозореца на браузъра, за да се получат резултати, но обикновено трябва или да се удостоверим, или да посочим какво търсим.
* POST: Заявка за запис към целевия ресурс. Очаква, че в резултат на това искане ще дойде нова информация.
* PUT: Актуализира някъде съществуващи данни, вероятно в някои бази данни.
* PATCH: Донякъде подобен на PUT и според моя опит изобщо рядко се използва.
* DELETE: Очаква информацията да бъде изтрита в резултат на заявката

Ако всичко това изглежда като нова информация, силно препоръчвам да изтеглите Postman, за да се запознаете с това как са структурирани извикванията на API.

Засега ще се съсредоточим върху работата с проста GET крайна точка.

## Learning By Doing

Ако напоследък проверявате публикациите на Snkia, може би сте забелязали генерирани изящни визуализации на връзки. За да демонстрираме как да правим API повиквания чрез интерфейсен клиент с JQuery, ще разгледаме как да създаваме визуализации на връзки с помощта на API на LinkPreview. Тази услуга служи като добър урок, защото:

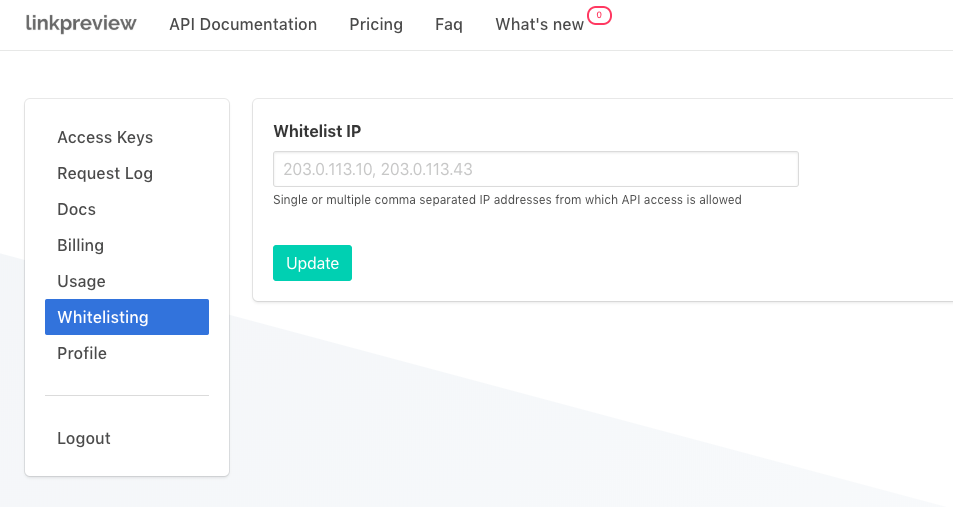
* Това е пример за проста GET крайна точка
* Има бърз и веднага полезен краен резултат
* Безплатно е

**Кажи ми, че това не е несигурно**

Тук искам да подчертая, че правим това в името на ученето; макар че това е може би най-бързият начин да започнете да работите с API, той определено не е сигурен.

Извършването на обаждания с частни ключове, съхранени и предадени от страна на клиента, излага вашия ключ публично. В производствена среда това е като да набуташ паролата си в лицата на хората. Хората определено ще искат да откраднат и експлоатират вашия частен ключ: ако това, което правите, няма никаква стойност, на първо място няма да изисква ключ.

Дано това ви изплаши достатъчно, за да помислите за преминаване на удостоверения в бъдеще. Въпреки това има още една солидна причина да изберем LinkPreview като днешен пример. LinkPreview предлага списък с разрешени домейни за заявки, така че дори някой да е откраднал ключа ви, той ще може да го използва само от вашия домейн;).



## Fetch Me Daddy

Отидете да започнете с API ключ в LinkPreview, ако следите. Предполагам, че вече имате JQuery лесно достъпен оттук нататък.

За да започнем, ще изчакаме зареждането на нашия документ и ще зададем две критични променливи: URL адресът на API и нашият API ключ.

**$**( document ).**ready**(**function**() {

**let** endpoint = 'https://api.linkpreview.net'

**let** apiKey = '5b578yg9yvi8sogirbvegoiufg9v9g579gviuiub8'

});

Set API endpoint and key

Ако следвате това, което сме направили с Lynx Roundups, следващата ни стъпка е да вземем всички съответни <a> тагове на дадена страница, да ги прегледаме и да ги заменим със съответните им визуализации на връзки:

**$**( document ).**ready**(**function**() {

**let** endpoint = 'https://api.linkpreview.net'

**let** apiKey = '5b578yg9yvi8sogirbvegoiufg9v9g579gviuiub8'

**$**( ".content a" ).**each**(**function**( index, element ) {

console.**log**(**$**( **this** ).**text**());

}

});

Loop over <a> tags

**Методът .each на JQuery създава цикъл, който итерира над всеки елемент, съответстващ на предоставения селектор. В нашия пример искаме само <a> тагове в съдържанието на нашата страница; в противен случай бихме получили всички връзки, като навигационни връзки и т.н.**

**Сега е време да внесем онова нещо за $ .ajax (), за което сме тръгнали.**

**$**( document ).**ready**(**function**() {

**let** endpoint = 'https://api.linkpreview.net'

**let** apiKey = '5b578yg9yvi8sogirbvegoiufg9v9g579gviuiub8'

**$**( ".content a" ).**each**(**function**( index, element ) {

$.**ajax**({

url: endpoint + "?key=" + apiKey + " &q=" + **$**( **this** ).**text**(),

contentType: "application/json",

dataType: 'json',

**success**: **function**(result){

console.**log**(result);

}

})

});

});

HTTP request with JQuery AJAX

Ето как е структурирана заявката на Ajax: съдържанието на $ .ajax () по същество е обект, който приема стойности, които ще използва за изграждане на заявката. Горният пример е толкова лесен, колкото е лесен за извършване на GET повикване с необработени кости. Преглеждаме всеки <a> таг и предаваме съдържанието му (URL адреса) на API и получаваме обект в отговор.

Заявките на Ajax могат да поемат повече параметри от тези, които току-що посочихме. Препоръчвам да прочетете внимателно документацията за JQuery Ajax; не само заради тези заявки, но разбирането на потенциалните елементи, които можем да посочим, ще затвърди разбирането за REST API като цяло.

Редът contentType: "application / json" указва, че съдържанието, което се връща при нас, ще бъде във формат JSON - това е много често срещана заглавка при работа с REST API.

С късмет отговорът ви трябва да се върне, изглеждащ така:

{

**"title"**:"Google",

**"description"**:"Search webpages, images, videos and more.",

**"image"**:"https//:www.google.com/images/logo.png",

**"url"**:"https://www.google.com/"

}

JSON response from our AJAX request

Ако искате да използвате това по смислен начин, не се колебайте да направите нещо като хаоса, който съм събрал по-долу:

**$**( document ).**ready**(**function**() {

**let** endpoint = 'https://api.linkpreview.net'

**let** apiKey = '5b578yg9yvi8sogirbvegoiufg9v9g579gviuiub8'

**$**( ".content a" ).**each**(**function**( index, element ) {

$.**ajax**({

url: endpoint + "?key=" + apiKey + " &q=" + **$**( **this** ).**text**(),

contentType: "application/json",

dataType: 'json',

**success**: **function**(result){

**$**( element ).**after**(

'<a href="' + result.url + '"> \n ' +

'<div class="link-preview"> \n ' +

'<div class="preview-image" style="background-image:url(' + result.image + ');"></div> \n ' +

'<div style="width:70%;" class="link-info"> \n ' +

'<h4>' + result.title +'</h4> \n ' +

'<p>' + result.description +'</p> \n ' +

'</div><br> \n ' +

'<a href="' + result.url + '" class="url-info"><i class="far fa-link"></i>' + result.url + '</a> \n ' +

'</div></a>');

**$**( element ).**remove**();

}

})

});

});

Generate HTML embed from JSON response

Този шаблон трябва да ви служи добре за повечето GET API обаждания, които ще правите чрез JQuery. Развихрете се и вижте какво можете да направите, за да използвате API-та и да разкриете личните данни на някои хора или каквото и да било.

Внедряване (или имплементиране) на проект (deployment).

**Имплементиране, създаване и Deploy-ване на FUSE приложение базирано на REST API**

Можете да използвате Red Hat Fuse API Designer, за да генерирате проект на Camel Fuse въз основа на дефиниция на REST API. Във вашата среда за разработка на Fuse можете да завършите маршрутите на Camel и Rest DSL API. И накрая, можете да изградите проекта и да разположите полученото приложение във Fuse на OpenShift.

**Предпоставки**

Имате съществуваща дефиниция на API, която отговаря на спецификацията OpenAPI 2.0. Например файл openapi-spec.json, който сте създали с API Designer.

API Designer е инсталиран и работи на вашия локален клъстер OpenShift.

Имате съществуващ проект на OpenShift с API Designer, добавен като услуга.

Инсталирали сте Maven и Red Hat Fuse.

Следните теми описват как да внедрите, изградите и внедрите приложение Fuse на базата на REST API:

* Раздел 3.1, „Качване на дефиниция на API в API Designer“
* Раздел 3.2, „Генериране на проект за Fuse Camel от API Designer“
* Раздел 3.3, „Завършване на проекта на Camel, генериран от API Designer“
* Раздел 3.4, „Изграждане и внедряване на REST услуга“

**3.1. Качване на дефиниция на API в API Designer**

Можете да качите съществуваща дефиниция на API в API Designer.

**Предпоставки**

Имате съществуваща дефиниция на API, която отговаря на спецификацията OpenAPI 2.0. Например файл openapi.json, който сте създали с API Designer.

API Designer е инсталиран и работи на вашия локален клъстер OpenShift.

Имате съществуващ проект на OpenShift с API Designer, добавен като приложение.

**Процедура**

* Във вашата уеб конзола OpenShift отворете проекта, който съдържа API Designer.
* Отворете конзолата на API Designer. В списъка с приложения за проекта щракнете върху URL адреса под apidesigner. Например: https://apidesigner-myproject.192.168.64.38.nip.io
* Конзолата на API Designer се отваря в отделен раздел или прозорец на уеб браузъра.
* Щракнете върху Open API.
* Отваря се прозорец на файловия мениджър.
* В прозореца на файловия мениджър:
* Придвижете се до папката, която съдържа съществуващия файл с дефиниция на OpenAPI, например openapi.json.
* Изберете файла за дефиниция на OpenAPI и след това щракнете върху Отваряне.
* Определението OpenAPI се отваря в конзолата на API Designer.

**3.2. Генериране на проект Fuse Camel от API Designer**

Можете да използвате API Designer, за да генерирате проект Fuse Camel въз основа на дефиниция на API.

Предпоставки

API Designer е инсталиран и работи на вашия локален клъстер OpenShift.

Имате съществуващ проект на OpenShift с API Designer, добавен като приложение.

Създали сте или отворили файл с дефиниция на API в конзолата на API Designer.

Процедура

В конзолата на API Designer:

Щракнете върху Генериране.

Изберете Fuse Camel Project от падащия списък.

API Designer генерира файл camel-project.zip и го изтегля във вашата локална папка за изтегляне по подразбиране.

Zip файлът съдържа проект Fuse Camel, който осигурява скелетно изпълнение по подразбиране на дефиницията на API, използвайки Camel’s Rest DSL и включва всички операции с ресурси. Проектът включва и оригиналния файл с дефиниция OpenAPI, който сте използвали за генериране на проекта.

**3.3. Завършване на проекта на Camel, генериран от API Designer**

API Designer генерира проект Fuse, който осигурява скелетна реализация по подразбиране на дефиницията на API, използвайки Camel’s Rest DSL и обхващаща всички операции с ресурси. Във вашата среда за разработка на Fuse вие завършвате проекта.

Предпоставки

Имате файл camel-project.zip, генериран от API Designer.

(По избор) Инсталирали сте Red Hat Developer Studio с Fuse Tooling.

Процедура

Разархивирайте генерирания от camel-project.zip файл на API Designer във временна папка.

Отворете Red Hat Developer Studio.

В Developer Studio изберете Файл → Импортиране.

В диалоговия прозорец Импортиране изберете Maven → Съществуващи проекти на Maven.

Отворете файла camel-context.xml на проекта в изгледа на редактора.

Щракнете върху раздела REST, за да редактирате компонентите Rest DSL.

За информация относно дефинирането на REST услуги, вижте раздела "Определяне на REST услуги" от Ръководството за развитие на Apache Camel.

За информация относно разширяването на крайни точки на JAX-RS с поддръжка на Swagger, вижте Ръководството за разработка на Apache CXF.

За информация относно използването на редактора REST на Fuse Tooling вижте раздела "Преглед и редактиране на компоненти Rest DSL" в Ръководството за потребителя на Tooling.

В раздела Дизайн редактирайте маршрутите на Camel.

За информация относно редактирането на Camel маршрути, вижте раздела "Редактиране на контекст на маршрутизиране в редактора на маршрути" в Ръководството за потребителя на Tooling.

**3.4. Изграждане и внедряване на REST услуга**

След като завършите проекта Fuse, можете да изградите и внедрите проекта в OpenShift.

Предпоставки

Имате пълен проект без грешки Fuse, който дефинира REST услуга.

Инсталирали сте Java 8 JDK (или по-нова версия) и Maven 3.3.x (или по-нова версия).

Процедура

Ако имате инсталиран и работещ клъстер OpenShift с един възел, като Minishift или Red Hat Container Development Kit, можете да разположите проекта си там.

За да разположите този проект в работещ клъстер OpenShift с един възел:

1. Влезте във вашия клъстер OpenShift:

$ oc login -u developer -p developer

1. Създайте нов проект OpenShift за проекта. Например следната команда създава нов проект с име test-deploy.

$ oc new-project test-deploy

1. Променете директорията в папката, която съдържа вашия проект Fuse Camel (например, myworkspace/camel-project) :

$ cd myworkspace/camel-project

1. Изградете и внедрете проекта в клъстера OpenShift:

$ mvn clean fabric8:deploy -Popenshift

1. 5. Във вашия браузър отворете конзолата OpenShift и отидете до проекта (например тестване-разгръщане). Изчакайте, докато видите, че шушулката за приложението на камила е стартирала.
2. 6. На страницата „Преглед на проекта“ намерете URL адреса на приложението за камила. URL адресът използва този формуляр: <http://camel-project-MY_PROJECT_NAME.OPENSHIFT_IP_ADDR.nip.io>.
3. Щракнете върху URL адреса за достъп до услугата.

Изпитна тема № 18:

Софтуерно инженерство План-тезис:

Етапи в софтуерната разработка. Методологии за разработка на софтуер. Инструменти. Сорс-контрол системи. Софтуерни изисквания и прототипи. Софтуерно тестване. Софтуерна документация. Примерна приложна задача:

Етапи в софтуерната разработка

Етапи от разработката на софтуер

**Анализ на изискванията** в системното инженерство и софтуерното инженерство, обхваща онези задачи, които определят потребностите или условията отговарящи за нови или изменени продукти, като се вземат предвид възможните противоречащи си изисквания на различни заинтересовани страни, анализиране, документиране, валидиране и управление на софтуера или системните изисквания. Анализа на изискванията е от решаващо значение за успеха на системите или софтуерните проекти. Изискванията трябва да бъдат *документирани, обжалвани, измерими, проверими, проследими, свързани с определени бизнес потребности или възможности, и да определя ниво на детайлност, достатъчно за проектирането на системата.*

**Спецификация на изискванията (SRS)** представлява пълно описание за поведението на една система, която ще бъде развивана, и да включва набор от използвани случаи описващи взаимодействията, които потребителите ще имат със софтуера. Спецификация на изискванията съдържа ѝ нефункционални изисквания, които налагат ограничения по отношение на дизайна или имплементацията (като например изисквания за ефективност, стандарти за качество или ограничения по проекта).

Документът отнасящ се за спецификация на софтуерните изисквания събира всички необходими изисквания, които са важни за развитието на проекта. За получаване на изискванията е нужно ясно и задълбочено познание за продуктите, които ще се развиват. Това се получава след подробна комуникация между екипа на проекта и клиента.

**Софтуерната архитектура** определя високите структури в нивото на една софтуерна система и предоставя един по-обобщен поглед върху даден проект. Тази архитектура може да се определи и като множество от структури, които обхващат софтуерните елементи, отношенията между тях и техните свойства.

Софтуерната архитектура означава и система от практики за подбор, определяне и проектиране на една софтуерна структура.

Също така термина софтуерна архитектура може да се разглежда като документация на софтуерната система. Документирането улеснява комуникацията между заинтересованите страни в проекта, очертава основните насоки на последващия дизайн и позволява повторна употреба на компонентите в други проекти.

**Софтуерният дизайн** е процес, чрез който се създава софтуерен продукт с определена цел с помощта на набор от основни компоненти и подлежащи ограничения. Към този дизайн се отнасят и всички дейности, включващи концепцията, изготвянето, прилагането, въвеждането в експлоатация и последващата промяна на цялата система.

Софтуерният дизайн обикновено включва разрешаването на проблеми и планиране на софтуерното решение. Това обхваща както алгоритмичния дизайн на ниско равнище така и софтуерната архитектура на високо ниво.

**Писането на програмен код** започва, когато дизайна приключи и важните решения относно софтуерната система са направени. Целта на фазата на писането на код е да се преведе дизайна на системата използвайки кода на някой от езиците за програмиране. Винаги се търси най-добрия начин за това представяне, защото се влияе директно на фазите за тестване и поддръжка. Добре написаният код намалява усилията при тестването и поддръжката.

За ефективно оползотворяване на времето работата се разделя на модули/части, които се задават като под-задачи на отделните софтуерни разработчици според техните умения.

Тази фаза отнема най-много време спрямо останалите.

**Софтуерното тестване** е процес на изследване и проучване на софтуера, с цел получаване на информация за качеството на продукта и услугата, която се изпитва. То може също да предостави обективна и независима гледна точка върху софтуера и да позволи на компаниите да оценят риска от вграждането на този софтуер.

Техниките за тестване включват, но не се изчерпват с, процеса по изпълнение на програмите и приложенията, за да се намерят евентуално грешки или други дефекти в тях. Важно е да се установи дали написания софтуер реално удовлетворява зададените при дизайна параметри. Тестовете на качеството може да се извършат или ръчно, или чрез автоматизирани тестващи инструменти, докато не е сигурно, че всички компоненти на софтуера работят коректно.

Софтуерното тестване е компромис между бюджета, времето и качеството.

**Дебъгване** (на английски: debugging) е процесът на проследяване на изпълнението на дадена компютърна програма с цел намиране и отстраняване на грешки („бъгове“) в нея. Дебъгването има тенденция да бъде по-трудно когато различни подсистеми са тясно свързани, тъй като промените по едната може да доведе до грешки в другата. Процесът на отстраняване на грешки се извършва с помощта на специализирани програмни инструменти наречени дебъгери.

**Внедряването на софтуер** обхваща всички процеси, които се извършват, за да може една софтуерна система да бъде напълно готова за употреба.

Най-общо това са няколко свързани дейности с възможност за преход между тях. Тези дейности могат да се проявят както при изпълнителят на проекта, така и при клиента. Поради уникалността на всяка една софтуерна система е трудно да се дефинират точните процеси и процедури, които се извършват. Следователно внедряването на софтуер следва да се тълкува като общ процес, който трябва да се адаптира в зависимост от специфичните изисквания на всеки отделен проект.

**Поддръжка на софтуера** е процеса на модифициране на софтуерните продукти след пускането им в експлоатация и цели подобряване на характеристиките и качествата им.

Всяка работа по промяна на софтуера след това се счита за поддръжка. Целта е да се запази стойността на софтуера за по-дълго време. Това се постига чрез добавяне на нови функции, улесняване на използването, постигането на по-голяма ефикасност и прилагането на нови технологии. Дейностите, които се извършват, включват поправка на грешки, усъвършенстване на възможностите, премахване на остарели функционалности и оптимизации.

Тъй като промените са неизбежни, в софтуерните системи е необходимо да бъде заложена възможността да се правят модификации още при тяхното проектиране.

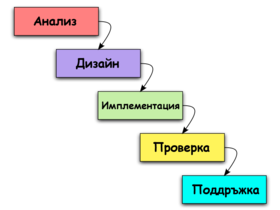
Методологии за разработка на софтуер

Процеси на разработка на софтуер

**Методологии**

**Методологията** е свързана от една страна с анализ на принципите и методите, правилата и постулатите, прилагани в една дисциплина, а от друга – със систематичното изследване на методите, които са или могат да бъдат приложени в тази дисциплина.[[7]](https://bg.wikipedia.org/wiki/%D0%A1%D0%BE%D1%84%D1%82%D1%83%D0%B5%D1%80%D0%BD%D0%BE_%D0%B8%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D1%81%D1%82%D0%B2%D0%BE#cite_note-7)

**Водопаден модел**

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:%D0%92%D0%BE%D0%B4%D0%BE%D0%BF%D0%B0%D0%B4%D0%B5%D0%BD_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB.png)

Немодифицирания „водопаден модел“. Процеса тече от горе надолу като каскаден водопад.

**Водопадният модел** (среща се и като **каскаден модел**) е една от най-ранните методологии разработена за изграждане на софтуерни продукти. Този модел разделя софтуерните процеси на различни фази, всяка от които следва точно определен ред при разработката на софтуер. Тези фази са:[[8]](https://bg.wikipedia.org/wiki/%D0%A1%D0%BE%D1%84%D1%82%D1%83%D0%B5%D1%80%D0%BD%D0%BE_%D0%B8%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D1%81%D1%82%D0%B2%D0%BE#cite_note-Sommerville_2008_89-8)

1. Спецификация на изискванията
2. Софтуерен дизайн
3. Имплементация и интеграция
4. Тестване (или Валидация)
5. Внедряване (или Инсталация)
6. Поддръжка

Процесите при водопадния модел протичат линейно и последователно. Всеки от етапите в процеса на разработка започва, само когато предишната фаза е напълно завършена. При стриктно спазване на методологията, връщане към предишна фаза за преправяне на продукта поради промяна на изискванията, не се допуска.

**Спирален модел**

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Spiral_model_(Boehm,_1988).svg)

Спирален модел (Boehm, 1988).

Главната характеристика на Спиралния модел е рисковото управление във всяка стъпка от процеса на разработка. През 1988, [Barry Boehm](https://bg.wikipedia.org/w/index.php?title=Barry_Boehm&action=edit&redlink=1) публикува официална система за разработка на софтуер наречена „Спирален модел“, която комбинира някои основни аспекти на методологиите на [водопадния модел](https://bg.wikipedia.org/w/index.php?title=%D0%92%D0%BE%D0%B4%D0%BE%D0%BF%D0%B0%D0%B4%D0%B5%D0%BD_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB&action=edit&redlink=1) и модела за [бързо създаване на прототипи](https://bg.wikipedia.org/w/index.php?title=Rapid_application_development&action=edit&redlink=1). В Спиралния модел се набляга на някои ключови области, в които при другите два модела са пренебрегнати като се разисква: необходимостта от по-задълбочен анализ на риска крайно необходим за големи по мащаб и сложност системи.

Спиралният модел се визуализира като повтарящ се процес преминаващ през няколко сектора, които чрез диаграма с четири квадранта могат да се визуализират следните дейности:

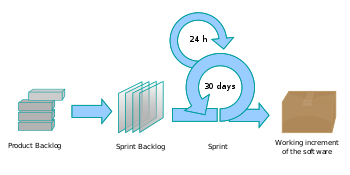
1. Изграждане на план за: набелязване на целите, разработване на програма, уточняване на задачите и дейностите по проекта.
2. Анализ на риска включва: оценка доколко са успешни набелязаните програми, идентифициране на рисковите моменти по време на разработването и предотвратяването им.
3. Имплементирането на съответния проект: разработването на програмния код и преминаването през съответните тестове за годност и пускане на пазара.

Риск-устойчивият спирален модел, набляга на условностите на различни опции и ограничения с цел да осигури четимост на кода – възможност за надграждането и редактирането му в процеса на разработка. Въпреки това, спиралния модел има някои ограничителни условия които са:

1. Спиралният модел набляга на анализа на риска, поради тази причина се изисква клиента да приеме тази особеност и да сътрудничи активно в процеса на разработване. Това изисква доверие в софтуерния разработчик, както и желание да се инвестира време в изчистването на малките детайли. Това е и причината този модел да е по-подходящ за мащабни вътрешно-търговски приложения.
2. Ако анализа на риска ще повлияе в значителна степен на себестойността на проекта, то тогава спиралния модел е неподходящ за прилагане.
3. За да сработи спиралния модел, софтуерния разработчик активно взима участие в анализа на рисковете преди, по време и след разработването на проекта.

Първата стъпка е да се изработи план за постигане на целите при вече изяснените ограничения, след това да се открият и отстранят всички рискове чрез внимателен анализ, а при необходимост да се изработи и прототип. Ако някои усложнения не могат да бъдат предотвратени, клиентът сам трябва да реши дали да прекрати разработката, или просто да ги пренебрегне. Накрая, след като резултатите се оценят, започва разработването на следващата фаза от планирането.

**Scrum методология**

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Scrum_process.svg)

Процесът на Scrum

**Scrum** e итеративна, инкрементална рамка за управление на проекти, често при гъвкавите модели на разработване.

Макар че, подходът на Scrum е първоначално предложен за управление на проектите за разработване на продукти, той се фокусира с времето върху управлението на софтуерни проекти и може да бъде използван за да задвижва екипи за софтуерна поддръжка или като общ подход за проектов / програмен мениджмънт. Тази методология е променила възприятията за типичното управление на проекти, като ясно показва предимствата на гъвкавите пред „водопадните“ или неитеративните, негъвкави методологии.

Scrum процесът се състои от отделни итерации, наречени *спринтове*. Спринтовете могат да имат продължителност от една седмица до четири седмици. В края на всеки спринт, екипът разполага с работеща версия на продукта, която включва всички готови задачи от backlog-а.

SCRUM е разработен за тези компании, чиято верига на стойността (изобразяваща процеса от началото до края на продукта и всяка стъпка, на която се добавя стойност) прави дългосрочното планиране на продукта доста трудно. За разлика от типичния мениджмънт чрез контрол и командване, тук се набляга на обратната връзка и се дава повече власт в ръцете на хората, които извършват операциите по процесите.

**Гъвкава методология**

**Гъвкави методологии** (английски: *agile*) за разработка на софтуер е неформален сбор от методологии и техники за управление на проекти за разработка на софтуер. Както подсказва и името, във фокуса на гъвкавите методологии е идеята, че разработката на софтуер е динамичен процес, в който дългосрочното планиране има ограничена ефективност.

Гъвкавите методологии намират особено широко приложение в разработката на продукти, където чрез учестеното създаване на прототипи, производителите имат възможност да получат обратна връзка от клиентите и да адаптират разработката според новопостъпилите от това изисквания.

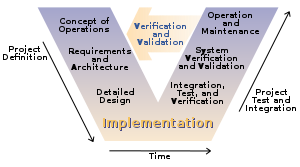
Част от практиките на гъвкавите методологии намират приложение и в други сфери на бизнеса, главно в ИКТ.

Списък с някои от най-популярните гъвкави методологии:

* Scrum
* Екстремно програмиране
* Kanban или Lean
* Crystal
* Dynamic Systems Development Method (DSDM)
* Feature-Driven Development (FDD)

Докато всяка от гъвкавите методологии е уникална със своя специфичен подход, всички те споделят общи концепции и основни ценности на Agile Manifesto.

**V-образен модел**

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Systems_Engineering_Process_II.svg)

V-образният модел в системното инженерство.[[11]](https://bg.wikipedia.org/wiki/%D0%A1%D0%BE%D1%84%D1%82%D1%83%D0%B5%D1%80%D0%BD%D0%BE_%D0%B8%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D1%81%D1%82%D0%B2%D0%BE#cite_note-FHWA_05-11)

**V-образният модел** представлява методология за разработка на софтуер (също така приложима в разработването на хардуер), която може да бъде разглеждана като разширение на водопадния модел.

Вместо процеса на разработка на софтуер да се движи линейно надолу, след етапа на писане на изходен код процесите се „извиват“ нагоре, като по този начин графиката придобива характерната за модела V-образна форма. V-моделът показва връзките между всяка от фазите на жизнения цикъл на софтуерната разработка и как те се отнасят към фазата на тестване. По хоризонталната ос е представено времето или степента на пълнотата на проекта (от ляво надясно), а по вертикалната – нивата на абстракция.

Поддръжниците на V-образния модел твърдят, че той се развива във времето и подкрепя гъвкавостта и подвижността през цялото развитие на процеса. Те също така твърдят, че предлага много дисциплиниран подход и спомага за щателното проектиране, разработване и документиране, необходими за изграждането на стабилни софтуерни продукти. Напоследък V-моделът се приема и в медицинско-устройствената индустрия.

**Бърза разработка на приложения**

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:RADModel_bg.jpg)

Фази на бързата разработка на приложения (RAD).

**Бърза разработка на приложения** (английски: *RAD*) е методология, която използва минимално планиране за сметка на бързото създаване на софтуерни прототипи. Планирането на софтуера, който се разработва, се припокрива с писането на самия код. Липсата на по-задълбочено предварително планиране позволява кода да се пише много по-бързо, а също така и да се правят лесно промени в първоначалните изисквания. RAD включва итеративни методи за софтуерна разработка и софтуерно прототипиране.

При бързата разработка на приложения, структурни техники и създаване на протипи се използват основно, за да се определят изисквания на потребителите и да се изработи крайната система. Процеса на разработка започва със създаването на предварителен модел на данните и процесите чрез структурни техники. При следващия етап изискванията се проверяват като се използват прототипи, за да може да се стигне до усъвършенстване на създадените модели. Тези етапи се повтарят итеративно.

Етапите при RAD са четири:

* *Планиране на изискванията* – комбинира елементи от фазите на системното планиране и анализиране. Потребители, мениджъри и техническия персонал дискутират заедно и взимат решения относно обхвата на проекта, целите и системните изисквания.
* *Фаза на потребителския дизайн* – през тази фаза потребителите подпомагат системните анализатори и се изработват модели и прототипи, които представят всички процеси, входове и изходи.
* *Конструкция* – фокусът е насочен към програмата и нейната имплементация. При RAD потребителите продължават своето участие и могат да предлагат промени и подобрявания.
* *Експлоатация* – представлява крайната фаза, където се извършва преработка на данните, тестване и обучение на потребителите. Сравнен с традиционните методи тук процеса е редуциран. Резултата е пускане на системата в експлоатация за по-кратък период.

**Итеративно и постепенно развитие**

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Iterative_development_model_V2_bg.jpg)

Схема на итеративното и постепенно развитие

**Итеративното и постепенно развитие** е всякаква комбинация от итеративен дизайн и методи и модела на постепенно изграждане за разработка на софтуер. Тази комбинация е позната отдавна и е широко използвана за големи проекти. Връзката между итерациите и надграждането се определя от цялостната методология и процеса по разработката на софтуера. Точния брой и природата на конкретните стъпки и кое ще се повтаря е специфично за отделните проекти.

Итеративното и постепенно развитие са основни съставни части на Модифицирания Водопаден Модел, Рационалния Унифициран Процес, Екстремното програмиране и са основна част на някои Аджайл методологии.

Основната идея зад този метод е системата да се разработва чрез повторяемост на цикли (итерации) и това да става за малки периоди от време (постепенно), давайки възможност на разработчиците да се възползват от наученото през по-ранните етапи на системата. При всяка итерация се правят промени по дизайна и се добавят нови функционалности.

Фази:

* *Проучване* – определят се изискванията (функционални и не-функционални) и риска на общо ниво, но с достатъчно детайли, за да може да се оцени обема на работата.
* *Разработка* – създава работеща архитектура, която намалява основните рискове и допълва не-функционалните изисквания.
* *Конструкция* – постепенно допълва архитектурата с работещ код получен след анализ, дизайн, изпълнение и тестване на функционалните изисквания.
* *Преход* – превръща системата в готова за пускане в експлоатация среда.

Всяка от тези фази може да бъде разделена на една или няколко итерации, които са по-често разпределени по време от колкото по функционалност. Архитектите и анализаторите работят с една итерация в аванс, за да може навреме да снабдяват с работа разработчиците и тестерите.

**Писане на код и фиксиране**

**Писането на код и фиксиране** е може би най-често използваната методология при разработката на софтуер. При нея се започва с много малко или дори никакво планиране. Писането на код е основната начална дейност и когато се появят проблеми те се отстраняват докато проекта бъде завършен.

Това е примамлив избор когато графика за разработката на продукта е силно намален, защото писането на кода започва веднага и следователно бързо се постигат някакви резултати.

Главния недостатък на тази методология се проявява ако бъде открит сериозен архитектурен проблем късно в процеса на разработка, защото тогава се налага пренаписването на големи части от кода. Съществуват алтернативни модели, които могат да помогнат да се уловят такива проблеми в по-ранен етап, когато промените са по-лесни и евтини.

Писането на код и фиксирането е подходящо за малки проекти и не се предполага да бъде използвано за основа на бъдещата разработка на софтуер.

Разработка на софтуер

Разработка на софтуер

[](https://bg.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Coding_Shots_Annual_Plan_high_res-5.jpg)

[Софтуерен разработчик](https://bg.wikipedia.org/wiki/%D0%A1%D0%BE%D1%84%D1%82%D1%83%D0%B5%D1%80%D0%B5%D0%BD_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%87%D0%B8%D0%BA) на работа

**Разработката на софтуер** (среща се и като **разработка на приложения**, **софтуерен дизайн**, **проектиране на софтуер**, **разработване на приложен софтуер**) е разработването на софтуерен продукт съобразен с нуждите на дадена целева група или маркетинга на един софтуерен продукт. Терминът „разработка на софтуер“ може да се използва, за да опише компютърното програмиране, което е процес на писане и поддържане на сорс код, но в по-широк смисъл на понятието се включва всичко – от концепцията на желания софтуер до крайната проява на софтуера, което в идеалния случай е планиран и структуриран процес. Следователно, разработката на софтуер може да включва изследвания, нови разработки, прототипиране, модификация, повторно използване, ре-инженеринг, поддръжка, или всякакви други дейности, чийто краен резултат е софтуерният продукт.

**Основни задачи**

**Планиране**

Планирането е част от всеки проект. В процеса на планиране се откриват конкретни задачи свързани със самия проект.

Важна част от създаването на софтуерна програма е извличане на изискванията или техния анализ. Клиентите най-често имат абстрактно виждане какво искат като краен резултат и нямат идея какво софтуерния продукт трябва да прави. Опитните софтуерни инженери разпознават непълните, прекалено амбициозни или често противоречиви изисквания още във фазата на планиране. Честото тестване и изпробване на продуктите в процеса на разработка намалява риска от внедряване на ненужни изисквания.

След като основните изисквания са събрани от клиента, започва техния по-задълбочен анализ. Определя се обхвата на разработения продукт като се поставят конкретни задачи на проекта и се изработва съответната документация.

Някои функционалности могат да останат извън обхвата на проекта като впоследствие могат да го оскъпят. Най-честа причина е неяснота по отношение на изискванията и приложението в самото начало на разработване. Ако друга компания извършва разработването и планирането, този документ може да се счита за правен документ и е част от договорните отношения. В случай на възникване на спорове и двусмислено тълкуване – какво е обещано на клиента и какво е получено като продукт, документацията по разработване на проекта може да се приложи за изясняване и разрешаване на спорни моменти.

**Архитектура, имплементация, тестване и документация**

Софтуерна архитектура представлява процеса за изработка на архитектурата на софтуера, при което се избират технологиите които ще се използват, стандарт за писане на код, инструменти и платформа. Разделят се сложните задачи на по-прости и лесни за изпълнение. Разделяне на компоненти и описание на тяхната функционалност.

Имплементацията е част от процеса, при който софтуерните инженери пишат програмния код за проекта на база вече създадената архитектура.

Софтуерното тестване е процес на изследване и проучване на софтуера, с цел получаване на информация за качеството на продукта и услугата, която се изпитва. Процесите на софтуерното тестване са неразделна част от софтуерното разработване и осигуряване на качеството на софтуера.

Документацията описва начина на работа на програмата, така че тя да може да бъде използвана от крайните потребители. Без ясна документация софтуерът трудно може да бъде използван, особено когато става дума за силно специализирани и сравнително сложни продукти, като Photoshop или AutoCAD.

Често софтуерът включва и техническа документация, предназначена за използване от разработчиците. Тя може да бъде във вид на коментари в самия изходен код или обособена в отделни файлове. Предназначението на тази документация е да улесни бъдещата поддръжка и промяна на софтуера.

**Внедряване и поддръжка**

Внедряването започва веднага след като кода мине през подходящо тестване, одобрен е за пускане на пазара, продаден или е пласиран по друг начин. Това може да включва инсталиране на съответен брой машини, настройка на продукта към конкретните изисквания на клиента, тестване и при необходимост удължаване на периода на оценяване.

Обучението за работа със софтуера и техническата поддръжка е изключително важно, тъй като ефективността му зависи от познанията на обикновения потребител.

Отстраняване на дефектите и подобряването на софтуера при възникнали проблеми може да отнеме значително време и усилия, а пропуснатите изисквания могат да принудят цялостна пренастройка на софтуера.

Инструменти за разработка на софтуер

Инструменти за разработка

Инструментите за разработка на софтуер са важни за вас, в противен случай не бихте попаднали в тази статия. Ефективният инструмент може значително да увеличи производителността на екипа. Не става въпрос само за инструменти за програмиране, но и за комуникация и управление на ресурси.

Нов инструмент за разработване на софтуер отнема време за внедряване, свикване и разбиране. Има много възможности, когато става въпрос за избор на платформа за разработка. Тази статия предоставя списък с често използвани инструменти, използвани в отделите за разработка.

1. Sublime Text

Sublime Text е инструмент за разработка на софтуер за редактиране на изходен код. Има много клавишни комбинации, които ви позволяват лесно да отваряте файлове и да превключвате между редовете. Можете да редактирате много редове наведнъж поради опцията за множествен избор. Можете също така лесно да превключвате между проекти. API на приставката също е силен.

2. Инструмент за разработване на софтуер Intellij

Intellij е интегрирана среда за разработка. IDE интегрира всички необходими инструменти. Така че има редактор на код, компилатор, дебъгер и др. Intellij завършва автоматично и е наистина удобен за потребителя и лесен за използване. Също така помага за бързо навигиране през вашия код и анализ на грешки. Това значително увеличава производителността.

3. Docker (Докър)

Docker предлага платформа за контейнериране на софтуер, която ви позволява да включите вашето приложение или софтуер в контейнер за файлове. Този контейнер може да бъде поставен и изнесен навсякъде. Ще намерите всичко необходимо за работа: код, системни библиотеки и др. Това означава, че софтуерът ще работи навсякъде по един и същ начин и е независим от средата.

4. Git

Git е система за контрол на версиите с отворен код за софтуерни проекти. Разработчикът трябва да прави чести промени в кода, преди да е готова последната версия.

Система за контрол на версиите съхранява всяка направена промяна, за да може екипът да работи заедно. Съхранява се копие на кода на всеки разработчик. С Git работата в екип може да се синхронизира, кодът се актуализира по всяко време.

5. GitHub

GitHub е хранилище на Git в Интернет. Това е като dropbox за софтуерни проекти, където можете да намерите код. Когато качвате проект, имате избор да го направите публичен или частен. Това е чудесно място за мрежи, срещи със съмишленици и споделяне на проекти. Общността е огромна и броят на проектите е още по-голям.

6. Jira

Jira е предназначена за Agile екипи за планиране, проследяване и пускане на софтуер. Очевидно е, че докато Agile екипите използват Jira. Той поддържа управление на проекти. Jira е изключително адаптивна и има мощни функции за всеки етап от развитието. Можете да управлявате напредъка, да визуализирате текущата работа и да генерирате отчети в един инструмент.

7. Jenkins (Дженкинс)

Jenkins е сървър за автоматизация с отворен код, по-точно сървър за непрекъсната интеграция. Можете да си представите този инструмент за разработка на софтуер като посредник между вашия код и вашия сървър за изграждане. Дженкинс периодично проверява за промени на вашия сървър и ги изпраща на сървъра за изграждане. Инструментът има чудесни приставки, които улесняват живота.

8. Slack

Slack е приложение, което може да се използва за комуникация в екип. Страхотен е, защото може да се използва навсякъде. Използването значително намалява вътрешното използване на електронна поща. Slack е много практичен: имате „канали“, които създавате за проекти, екипи, теми и др. Можете лесно да навигирате между каналите. Освен това имате възможност да изпращате директни лични съобщения до членовете на вашия екип. Можете да споделяте и коментирате всякакви файлове или да маркирате хора. Можете също да търсите в цялата си история.

9. Препълване на стека

Stack Overflow всъщност не е инструмент за разработка на софтуер. Това е сайт с въпроси и отговори с най-голямата общност от програмисти. В тази библиотека ще намерите всички отговори на вашите въпроси. От промяна на цвета на текста до промяна на ядрото на Linux. Това е чудесно място за учене и споделяне на знания.

10. YouTube

С YouTube можете да преглеждате уроци и съвети от други ИТ специалисти. Но също така и семинари, които сме пропуснали.

Сорс-контрол системи

**Какво представлява системата за контрол на версиите**

**(Version Control System) ?**

Система за контрол на версиите позволява на потребителите да следят промените в проектите за разработване на софтуер и да им позволяват да си сътрудничат по тези проекти. Използвайки го, разработчиците могат да работят заедно по код и да разделят задачите си чрез клонове.

В системата за контрол на версиите може да има няколко клона според броя на сътрудниците. Клоновете поддържат индивидуалност, тъй като промените в кода остават в определен клон (и).

Разработчиците могат да комбинират промените в кода, когато е необходимо. Освен това те могат да преглеждат историята на промените, да се връщат към предишната версия (и) и да използват / управляват кода по желания начин.

**Предимства от използването на система за контрол на версиите**

Основните предимства на използването на система за контрол на версиите включват рационализиране на процеса на разработка, управление на код за множество проекти и водене на история на всички промени в рамките на код.

Софтуерът за контрол на версиите запазва всички промени в хранилището. Следователно, ако разработчиците допуснат грешка, те могат да я отменят. В същото време те могат да сравняват новия код с предишна версия (и), за да разрешат оплакването си. Това може до голяма степен да намали човешките грешки и нежеланите последици. Страхотно за всяка компания за уеб разработки по целия свят.

Освен това може да се интегрира с няколко инструмента за разработване на софтуер като доставчици на PaaS, интегрирани среди за разработка (IDE) и инструменти за автоматизация на изграждането.

Най-добрите системи за

контрол на версиите

На пазара има много опции. Затова създадохме списък с 10 най-добри софтуера за контрол на версиите, за да стесним опциите и да улесним нещата.

**1. GitHub**

GitHub помага на софтуерните екипи да си сътрудничат и поддържат цялата история на промените в кода. Можете да проследявате промените в кода, да върнете часовника назад, за да отмените грешките и да споделите усилията си с други членове на екипа.

Това е хранилище за домакинство на Git проекти. За тези, които се чудят какво е Git? Това е система за контрол на версиите с отворен код, която включва локално разклоняване, множество работни потоци и удобни области за подреждане. Контролът на версиите на Git е лесна за научаване опция и предлага по-бърза скорост на работа.

**2. GitLab**

GitLab се предлага с много удобни функции като интегриран проект, уебсайт за проекти и др. Използвайки възможностите за непрекъсната интеграция (CI) на GitLab, можете автоматично да тествате и доставяте кода.

Можете да осъществите достъп до всички аспекти на даден проект, да видите кода, да изтеглите заявки и да комбинирате разрешаването на конфликти.

**3. Beanstalk**

Beanstalk е идеален вариант за тези, които трябва да работят от отдалечени места. Този софтуер се основава на браузър и облак, позволявайки на потребителите да кодират, ангажират, преглеждат и разполагат с помощта на браузър.

Той може да бъде интегриран с платформи за съобщения и имейл за ефективно сътрудничество, свързано с кодове и актуализации. Той поддържа както Git, така и SVN и се предлага с вградени аналитични функции.

За сигурност използва криптиране, двуфакторно удостоверяване и функции за защита с парола.

**4. PerForce**

Perforce предоставя възможностите за контрол на версиите чрез своя HelixCore. HelixCore се предлага с единна платформа за безпроблемно екипно сътрудничество и поддръжка както за централизирани, така и за разпределени работни процеси.

Това е решение за сигурност, което защитава най-ценните активи. HelixCore ви позволява да проследявате точно промените в кода и улеснява пълната екосистема на Git.

**5. Apache Subversion**

Apache Subversion е друга система за контрол на версиите с отворен код, която е основана от CollabNet преди няколко десетилетия. Както арената с отворен код, така и предприятията я смятат за надежден вариант за ценни данни.

Основните характеристики на Subversion включват управление на запасите, управление на сигурността, проследяване на историята, контрол на достъпа на потребителите, евтино локално разклоняване и управление на работния процес.

**6. AWS CodeCommit**

AWS CodeCommit е управлявана система за контрол на версиите, която хоства защитени и мащабируеми частни Git хранилища. Той безпроблемно се свързва с други продукти от Amazon Web Services (AWS) и хоства кода в защитени AWS среди. Следователно той е подходящ за съществуващите потребители на AWS.

Интеграцията на AWS също осигурява достъп до няколко полезни приставки от партньори на AWS, което помага при разработването на софтуер.

**7. Сървър на Microsoft Team Foundation**

Разработен от Microsoft, Team Foundation Server е инструмент за корпоративно ниво за управление на изходния код и други услуги, които се нуждаят от версии. Той може да следи работните елементи, за да открива дефекти, изисквания и сценарии в даден проект.

Той се предлага с няколко уникални функции като Team Build, събиране на данни и отчитане, Team Project Portal, Team Foundation Shared Services и др.

**8. Mercurial**

Mercurial е известен със своята ефективност при работа с проекти от всякакъв размер. Това е безплатна и разпределена услуга за управление на управление, която осигурява опростен и интуитивен потребителски интерфейс.

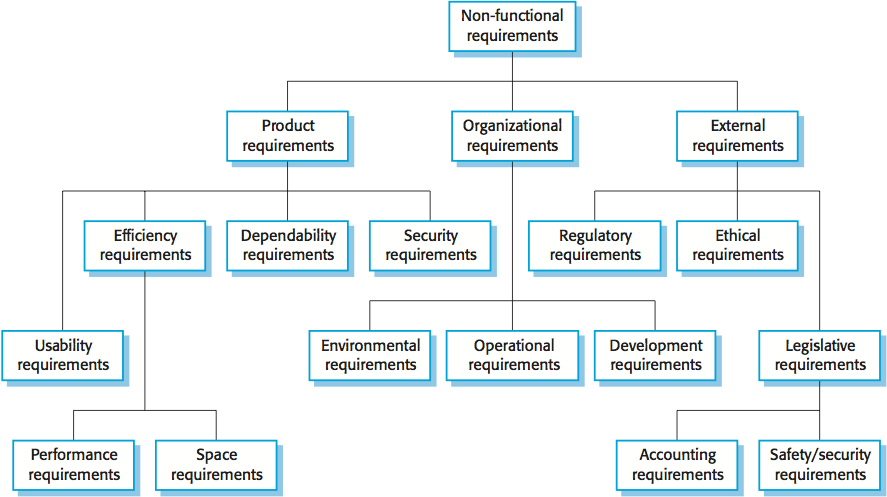
Разработчиците и предприятията обожават Mercurial за неговата система за архивиране, функционалност за търсене, проследяване и управление на проекти, импортиране и експортиране на данни и инструмент за мигриране на данни. Той също така включва управление на работния процес, история.

Софтуерни изисквания и прототипи

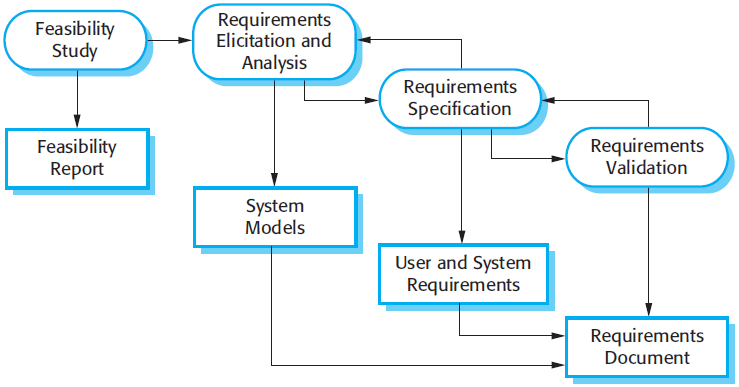
Софтуерни изисквания: -

Изискванията за като система се дефинират като това, което системата трябва да прави, услугата, която предоставя и ограниченията за нейната работа. Тези изисквания се отнасят до нуждите на клиента в дадена система. Видове изисквания: -

1. Функционални изисквания: - Това са отчети за услугата, които системата трябва да предоставя, как системата трябва да реагира на определени входове и как системата трябва да се държи в определени ситуации. Понякога може също така изрично да посочва какво системата не трябва да прави.
2. Нефункционални изисквания: - Това са ограничения за услугите или функциите, предлагани от системата. Те включват времеви ограничения, ограничения върху процеса на развитие и ограничения, наложени от стандартите. Нефункционалните изисквания често са по-критични от индивидуалните функционални изисквания. При липса на това води цялата система е неизползваема.



Инженерни процеси на изискванията: - Инженерният процес на изискванията може да включва дейности на четири нива. Те се фокусират върху оценка дали системата е полезна за бизнеса (проучване на осъществимостта), откриване на изискване (извличане и анализ), преобразуване на тези изисквания в някаква стандартна форма (спецификация) и проверка дали изискването действително определя системата, която клиентът иска (валидиране ).



Предпроектно проучване: - От изискването за софтуер, оценка дали системата може да се конкурира или не с текущата софтуерна и хардуерна технология. Проучването определя дали системата може да допринесе за общите цели на организацията, може ли системата да бъде внедрена в рамките на графика и бюджета, като се използва настоящата технология и може ли системата да бъде интегрирана с друга използвана система.

Извличане и анализ на изисквания: - На този етап софтуерните инженери работят с клиенти и крайни потребители, за да изведат системното изискване с дискусия. Той включва разнообразие от различни видове хора в организацията. Това може да доведе до заключение на един или много модели и прототип на системата. Процесната дейност е: -

i. Откриване на изисквания

ii. Класификация и организация на изискванията

iii. Приоритизиране на изискванията и договаряне

iv. Спецификация на изискванията

3. Управление на изискванията: - Изискванията за големи софтуерни системи винаги се променят. Защото проблемът не е напълно дефиниран, което води до непълна система. По време на софтуерния процес се разбират изискванията, които след това трябва да се променят в изискването. И дори след като системата е завършена, потребителят ще открие необходимостта от функция. Следователно управлението на изискванията е процесът на разбиране и контрол на промяната в системните изисквания.

4. Валидиране на изискванията: - Това е процесът, който изискването всъщност определя системата, която клиентът наистина иска. По време на този процес нашата цел е да открием грешките и да ги решим и да проверим пълнотата, последователността, реализма, проверките на валидността на софтуера.

Софтуерно тестване

**Дефиниция на „** **Софтуерно тестване“**

Определение: Процесът или методът за намиране на грешки / грешки в софтуерно приложение или програма, така че приложението да функционира според изискванията на крайния потребител се нарича софтуерно тестване.

Описание: Софтуерното тестване е процес на проверка на системата с цел идентифициране на грешки, пропуски или липсващи изисквания спрямо действителното изискване. Тестването на софтуера е широко разделено на два вида - функционално тестване и нефункционално тестване.

Кога да започнем тестовите дейности: Тестването трябва да започне възможно най-рано, за да се намалят разходите и времето за преработка и производство на софтуер, който не съдържа грешки, така че да може да бъде доставен на клиента. Въпреки това, в жизнения цикъл на разработката на софтуер (SDLC), тестването може да започне от фазата на събиране на изисквания и да продължи, докато софтуерът не бъде пуснат в производство. Това също зависи от модела на развитие, който се използва. Например, в модела Waterfall, тестването започва от фазата на тестване, която е доста отдолу в дървото; но във V-модела тестването се извършва успоредно с фазата на разработка.

Кога да спрете тестовите дейности: Приложението никога не може да бъде 100% без грешки. Така че е трудно да се установи кога може да се спре тестването. При спиране на тестовите дейности обаче се имат предвид следните неща:

• Срокове за тестване

• Завършване на цикъла на изпълнение на тестовия случай

• Завършване на функционално тестване и покритие на кода до уговорена граница

• Не се открива грешка с висок приоритет и скоростта на намиране на грешка намалява след определено ниво

• Управленско решение Процесът на тестване на софтуер се състои от изброените по-долу дейности:

• Планиране и контрол

• Анализ и дизайн

• Внедряване и изпълнение

• Спазване на критериите за излизане

• Дейности по затваряне на тестове.

Софтуерна документация

**Как да създадете софтуерна документация |2021 2021|**

Добрата документация за софтуера, независимо дали става въпрос за спецификационна документация за програмисти и тестери, технически документи за вътрешни потребители или ръководства и помощни файлове

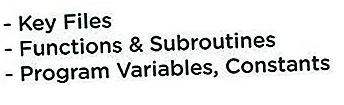


**Съдържание:**

Добрата документация за софтуера, независимо дали става въпрос за спецификационна документация за програмисти и тестери, технически документи за вътрешни потребители или ръководства и помощни файлове за крайни потребители, ще помогне на потребителите да разберат функциите и функциите на софтуера. Добрата документация е документация, която е специфична, ясна и уместна, с цялата информация, от която се нуждаят потребителите. Тази статия ще ви насочи при писането на софтуерна документация за технически потребители и крайни потребители.

**стъпка**

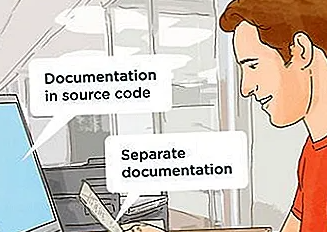
**Метод 1 от 2: Писане на документация за софтуер за технически потребители**



1. **Знайте каква информация да включите.**

 Документът със спецификация се използва като справочно ръководство за дизайнери на интерфейси, програмисти за писане на код и тестери, проверяващи работата на софтуера. Информацията, която трябва да бъде включена, ще зависи от програмата, която се създава, но може да включва следното:

* + Важни файлове в приложения, като например файлове, създадени от екипа за разработка, бази данни, достъп до които работи програмата, и приложения на трети страни.
  + Функции и подпрограми, включително описание на това, за което се използва функцията / подпрограмата, входни стойности и изходи.
  + Променливи и константи на програмата и как се използват всички тези променливи / константи.
  + Цялостна структура на програмата. За програми, базирани на диск, може да се наложи да опишете отделните модули и библиотеки. Или, ако пишете наръчник за уеб-базирана програма, може да се наложи да обясните кои файлове използва всяка страница.



1. **Решете какво ниво на документация трябва да има и може ли да бъде отделено от програмния код.**

Колкото повече техническа документация е включена в програмния код, толкова по-лесно ще бъде да се актуализира и поддържа и да се описват различни версии на програма. Като минимум документацията във вашия програмен код трябва да включва полезни функции, подпрограми, променливи и константи.

* + Ако вашият изходен код е дълъг, можете да напишете документация в помощен файл, който след това може да бъде индексиран или търсен по конкретни ключови думи. Отделни файлове с документация са полезни, ако логиката на програмата е разделена на няколко страници и включва файлове за поддръжка, като например уеб приложение.
  + Някои езици за програмиране (като Java, Visual Basic.NET или C #) имат свои собствени стандарти за кодова документация. В такива случаи следвайте стандартната документация, която трябва да бъде включена в изходния код.

1. **Изберете подходящия инструмент за документация.**

 В някои случаи инструментът за документация се определя от използвания език за програмиране. Езиците C ++, C #, Visual Basic, Java, PHP и други имат свои инструменти за документация. Ако обаче не, използваните инструменти ще зависят от необходимата документация.

* + Текстови процесори като Microsoft Word са подходящи за създаване на текстови файлове с документация, стига документацията да е кратка и проста. За да създадат дълга документация със сложен текст, повечето технически писатели предпочитат специализирани инструменти за документация, като Adobe FrameMaker.
  + Помощните файлове за документиране на изходния код могат да бъдат създадени с програма за създаване на поддръжка на файлове като RoboHelp, Помощ и наръчник, Doc-To-Help, MadCap Flare или HelpLogix.

**Метод 2 от 2: Писане на софтуерна документация за крайни потребители**

1. **Знайте бизнес причините, които стоят зад ръчното създаване**

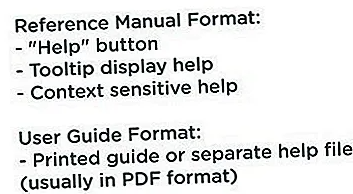
Въпреки че основната причина за софтуерната документация е да се помогне на потребителите да разберат как да използват приложението, има няколко други причини, които могат да са в основата на документацията, като например подпомагане на маркетинговата продажба на приложения, подобряване на корпоративния имидж и намаляване на разходите за техническа поддръжка. В някои случаи е необходима документация, за да се съобразят с разпоредбите или други законови изисквания.

* + Документацията обаче не замества добрия интерфейс. Ако дадено приложение изисква много документация за работа, то трябва да бъде проектирано така, че да бъде по-интуитивно.

**Познайте целевата аудитория за документацията.**

 По принцип потребителите на софтуер имат ограничени компютърни познания извън приложенията, използвани от тях. Има няколко начина за удовлетворяване на техните нужди от документация:

* + Обърнете внимание на заглавието на работата на софтуера. Например системните администратори обикновено разбират различни компютърни приложения, докато секретарите контролират само приложенията, които използват за въвеждане на данни.
  + Обърнете внимание на потребителите на софтуер. Въпреки че постовете им обикновено са в съответствие със задълженията, които изпълняват, тези длъжности могат да имат различно натоварване в зависимост от мястото на работа. Разпитвайки потенциалните потребители, можете да разберете дали вашата оценка на заглавието на тяхната работа е правилна.
  + Обърнете внимание на съществуващата документация. Спецификацията на документацията и функциите на софтуера може да покаже какво трябва да знае потребителят, за да го използва. Имайте предвид обаче, че потребителите може да не се интересуват от познаването на „вътрешностите“ на програмата.
  + Знайте кои неща са необходими за изпълнение на задачата и какви неща са необходими, преди да можете да я изпълните.



1. **Определете подходящ формат за документация.**

Софтуерната документация може да бъде подредена в 1 или 2 формата, а именно справочници и ръководства. Понякога комбинирането на двата формата е добро решение.

* + Референтните формати се използват за описание на всички функции на софтуера, като бутони, раздели, полета и диалогови прозорци и как работят. Някои помощни файлове са написани в този формат, особено тези, които са чувствителни към контекста. Когато потребителят кликне *Помогне* на определен екран, потребителят ще получи съответна тема.
  + Ръчният формат се използва за описание на това как да направите нещо със софтуера. Ръководствата обикновено се произвеждат в печат или PDF формат, въпреки че някои страници за помощ също включват инструкции за извършване на определени неща. (Обикновено ръчните формати не са чувствителни към контекста, но могат да бъдат свързани от контекстно-чувствителна тема.) Ръководствата обикновено са под формата на ръководства, с обобщение на задачите, които трябва да се изпълняват в описание и стъпка форматирано ръководство.

1. **Решете вида на документацията.**

Софтуерната документация за потребителите може да бъде опакована в един или повече от следните формати: отпечатани наръчници, PDF файлове, помощни файлове или онлайн помощ. Всеки тип документация е създаден да ви покаже как да използвате функциите на софтуера, независимо дали е ръководство или урок. Онлайн документацията и страниците за помощ могат също да включват демонстрационни видеоклипове, текст и статични изображения.

* + Онлайн файловете за помощ и поддръжка трябва да бъдат индексируеми и могат да се търсят с помощта на ключови думи, така че потребителите да могат бързо да намерят необходимата информация. Въпреки че приложението за генериране на помощни файлове може да генерира индекси автоматично, все пак се препоръчва ръчно да създавате индекси с често търсени ключови думи.



1. **Изберете подходящия инструмент за документация.** Печат наръчник или PDF може да бъде създаден с помощта на програма за текстообработка, като Word или сложен текстов редактор като FrameMaker, в зависимост от дължината и сложността на файла. Помощните файлове могат да бъдат написани в програма за създаване на помощни файлове, като RoboHelp, Помощ и наръчник, Doc-To-Help, Flare, HelpLogix или HelpServer.

**Съвети**

* Текстът на програмната документация трябва да бъде структуриран по лесен за четене начин. Позиционирайте изображението възможно най-близо до подходящия текст. Логично разбийте документацията по раздел и тема. Всеки раздел или тема трябва да описва конкретен проблем, както задание, така и функция на програмата. Свързаните проблеми могат да бъдат обяснени с връзка или списък с препратки.
* Всеки от инструментите за документация, описани в тази статия, може да бъде оборудван с програма за създаване на екрани, например SnagIt, ако вашата документация изисква много скрийншоти. Както и в друга документация, вие също трябва да включите екранни снимки, за да обясните как работи приложението, а не да „ангажирате“ потребителите.
* Обръщането на внимание на езиковия стил е много важно, особено ако пишете софтуерна документация за крайни потребители. Определете потребителите с местоимението „ти“ вместо „потребител“.