1. MVC е съкращение от Model-View-Controller, което е архитектурен модел за проектиране на софтуерни приложения. В този модел приложението е разделено на три основни компонента:

Модел (Model) – представлява данните и бизнес логиката на приложението. Това може да бъде например база от данни, която съхранява информация за потребителите и тяхната активност.

Изглед (View) – представлява начина, по който данните са представени на потребителите. Това може да бъде уеб страница или графичен потребителски интерфейс.

Контролер (Controller) – свързва модела и изгледа, като обработва входящи данни от потребителя и решава как да бъдат обработени. Това може да включва валидация на входните данни и извличане на информация от модела за показване в изгледа.

MVC е полезен за проектиране на големи софтуерни приложения, защото го прави по-лесно да се раздели функционалността на различни компоненти, които могат да бъдат разработени и поддържани поотделно. Освен това, той позволява на различни екипи от разработчици да работят по различни части на приложението без да влияят един на друг.

2. Бази данни са организирани съхраняващи устройства за данни, които позволяват на потребителите да съхраняват, организират, извличат и актуализират информация. Съществуват различни видове бази данни, като релационните бази данни са най-често използваните.

ORM (Object-Relational Mapping) е технология за свързване на обектно-ориентиран код с релационни бази данни. Това позволява на програмистите да работят с бази данни, използвайки обектно-ориентирани концепции, вместо да пишат SQL заявки директно. ORM инструментите генерират SQL заявки автоматично, като програмистите просто работят с обекти във своя код.

ORM технологиите осигуряват удобство и улесняват поддръжката на приложенията, като предоставят абстракция между базата данни и приложението. Това позволява на програмистите да се фокусират върху бизнес логиката на приложението, вместо да се занимават с низко ниво детайли на базата данни. Някои от популярните ORM технологии са Hibernate за Java, Entity Framework за .NET и Django ORM за Python.

Базите данни са изключително важен елемент в съвременните приложения и уеб сайтове, защото те съхраняват всички данни, които приложението трябва да използва. Релационните бази данни се състоят от множество таблици, които имат редове и колони. Редовете съдържат информация за конкретни обекти (например потребители, продукти или поръчки), а колоните съхраняват отделните атрибути на тези обекти.

ORM технологиите позволяват на програмистите да създават обекти, които отразяват редовете в базата данни и да използват тези обекти в своя код. ORM инструментите генерират SQL заявки автоматично, като програмистите не трябва да се занимават с директното писане на SQL заявки.

ORM технологиите обикновено предоставят множество полезни функции като механизми за валидация на данни, автоматично генериране на SQL заявки, транзакции и други. Тези функции могат да улеснят работата на програмистите и да намалят вероятността от грешки.

Съществуват много ORM технологии за различните програмни езици и платформи, като някои от най-популярните са:

Hibernate за Java

Entity Framework за .NET

Django ORM за Python

Ruby on Rails Active Record за Ruby

Всички тези ORM технологии имат своите специфични характеристики и функции, но всички те имат за цел да улеснят работата на програмистите при работа с бази данни.

3. CRUD операциите са основните операции, използвани в много приложения за управление на данни. Тези операции са Create (Създаване), Read (Четене), Update (Актуализиране) и Delete (Изтриване) и представляват основните начини за манипулиране на данни в една система.

Ето как може да се реализира CRUD функционалност в едно приложение:

Create (Създаване):

Създаване на форма за въвеждане на данни.

Проверка на входните данни за коректност и валидация.

Запис на данните в базата от данни.

Read (Четене):

Създаване на списък с всички записи в базата данни.

Извличане на информацията от базата данни.

Показване на информацията на екрана чрез HTML и CSS.

Update (Актуализиране):

Създаване на форма за редактиране на записи.

Избор на запис, който да бъде актуализиран.

Валидация на входните данни.

Запис на актуализираната информация в базата данни.

Delete (Изтриване):

Избор на запис, който да бъде изтрит.

Потвърждение за изтриване на записа.

Изтриване на записа от базата данни.

Всяка от тези операции може да бъде реализирана с помощта на различни езици за програмиране и фреймуърки. Например, може да използвате PHP и MySQL за създаване на уеб приложение, което да изпълнява CRUD операции. Има много фреймуърки, като Laravel, Django и Ruby on Rails, които предоставят инструменти и функционалности за лесно реализиране на CRUD операции.

CRUD операциите са от основно значение за много приложения, като уеб приложения, мобилни приложения и десктоп приложения. Те предоставят начин за манипулиране на данни в системата и са основната функционалност на много уеб сайтове, включително онлайн магазини, социални мрежи, блогове и форуми.

Една от най-честите технологии, използвани за създаване на уеб приложения, е моделът на изграждане на уеб-базирани приложения (Model-View-Controller – MVC). MVC е модел за разработка на софтуер, който разделя приложението на три основни части: модел, изглед и контролер. Тази архитектура е много полезна за реализиране на CRUD операции, като:

Моделът (Model) отговаря за работата с базата данни и изпълнението на заявки за вземане, създаване, актуализиране и изтриване на записи.

Изгледът (View) е отговорен за показване на информацията на екрана.

Контролерът (Controller) обработва заявките на потребителите и свързва модела и изгледа, за да изпълни необходимата функционалност.

С помощта на MVC и други подобни архитектури, може да се реализира CRUD функционалност на много езици за програмиране, като Java, Python, Ruby, PHP и други. Има много инструменти и библиотеки, които улесняват реализацията на CRUD операции върху база данни, като Hibernate за Java, Django ORM за Python, ActiveRecord за Ruby on Rails и Eloquent за Laravel.

4.Дефинира и създава шаблонен изглед

Шаблонният изглед (template view) е един от начините за организиране на представянето на информацията в уеб приложение. Той позволява да се раздели логиката на приложението от изгледа, който се показва на потребителя.

Шаблонният изглед се дефинира като HTML файл, който съдържа маркировъчен език, позволяващ да се дефинират променливи, условни оператори, цикли и други конструкции за генериране на динамично съдържание.

За да създадем шаблонен изглед в уеб приложение, първо трябва да дефинираме HTML шаблон с необходимите елементи. След това трябва да свържем шаблона с логиката на приложението, като използваме контролери или други механизми за обработка на заявки.

5.Автентикация и авторизация.

Да, автентикацията и авторизацията са две различни концепции, които често се позовават една на друга, но са фундаментално различни.

Автентикацията се отнася до проверка на идентичността на потребителя. Това означава, че потребителят представя свойте идентификационни данни (потребителско име и парола, биометрични данни, сертификати и др.), за да бъде потвърдено, че той е именно този, за който се представя. В случай на успешна автентикация, системата разпознава потребителя като валиден и му позволява да продължи да използва ресурсите или услугите на системата.

Авторизацията, от друга страна, се отнася до определянето на правата на потребителя. Това означава, че след като потребителят е успешно автентикиран, системата използва информацията за потребителя, за да определи какво може да прави и какви ресурси може да достъпва. Например, ако потребителят има администраторски права, той може да има достъп до повече функционалности в системата или да има правото да прави промени върху ресурсите, към които има достъп.

6.Диференцира сесиите от бисквитките.

Сесиите и бисквитките (познати също като HTTP бисквитки) са две различни неща, свързани с уебсайтовете и уеб приложенията.

Сесиите се използват за запазване на информация за потребителите, която трябва да бъде запазена между различни заявки към уебсайта или уеб приложението. Когато потребител посети уебсайта или приложението, сървърът създава уникална сесия за този потребител, която съхранява информацията, свързана с този потребител, като например данните за влизане, състоянието на кошницата за пазаруване и т.н. Сесиите са обикновено криптирани и защитени срещу злоупотреби.

Бисквитките също съхраняват информация, но тази информация е запазена на компютъра на потребителя, а не на сървъра. Когато потребител посети уебсайт, сървърът може да изпрати бисквитка до браузъра на потребителя, която да съдържа информация като например настройките на езика, предпочитанията за показване на уебсайта и т.н. Бисквитките могат да бъдат използвани и за отстраняване на проблеми със сесиите и за съхраняване на информация, която трябва да бъде запазена за дълъг период от време.

Предотвратяване: Приложенията трябва да се валидират и да се филтрират нежелани символи, за да се предотвратят възможни атаки.

SQL инжекции – това е уязвимост, при която злонамерени хора внедрят зловреден SQL код в уеб приложенията, който може да доведе до кражба на данни, изтриване на данни или дори до цялостно презаписване на базата данни.

Пример: Ако уеб приложение не се предпази от SQL инжекции, то злонамерени хора могат да внедрят зловреден SQL код във формата за вход на потребителите, който да им позволи да достъпят и модифицират данните в базата данни.

Предотвратяване: Уеб приложенията трябва да се предпазват от SQL инжекции, като се използват параметризирани заявки и се валидират входните данни.Web Application Security Vulnerabilities

Обяснява и дава пример за различните уязвимости в сигурността на уеб при-

Ложенията. Прави изводи за предотвратяването им.

7. Уязвимости в сигурността на уеб приложенията могат да представляват сериозен риск за личната и финансовата информация на потребителите. Някои от най-честите уязвимости включват:

Извъндомейн атаки (Cross-Site Scripting, XSS) – това е уязвимост, при която злонамерени хора внедряват зловреден код в уеб страници, които след това се изпълняват в браузъра на потребителите, като в този процес се крадат техните лични данни или се нанасят щети на компютърната им система.

Пример: Ако уеб приложение приема входни данни от потребителя без да ги валидира, то това може да доведе до XSS атака. Например, ако потребителят въведе скрипт, който извежда предупреждение, когато натисне бутон, някой друг може да внедри зловреден код в този скрипт, който да извлече личната му информация.

Предотвратяване: Приложенията трябва да се валидират и да се филтрират нежелани символи, за да се предотвратят възможни атаки.

SQL инжекции – това е уязвимост, при която злонамерени хора внедрят зловреден SQL код в уеб приложенията, който може да доведе до кражба на данни, изтриване на данни или дори до цялостно презаписване на базата данни.

Пример: Ако уеб приложение не се предпази от SQL инжекции, то злонамерени хора могат да внедрят зловреден SQL код във формата за вход на потребителите, който да им позволи да достъпят и модифицират данните в базата данни.

Предотвратяване: Уеб приложенията трябва да се предпазват от SQL инжекции, като се използват параметризирани заявки и се валидират входните данни.

Използването на уеб приложенията е станало част от нашия ежедневен живот и затова е много важно да се предпазим от потенциалните опасности, които могат да предизвикат уязвимости в сигурността на уеб приложенията.

CSRF атаки – това е уязвимост, при която злонамерени хора могат да изпратят заявки от името на потребителя, като използват неговата автентикационна сесия, без да го уведомят за това. Това може да доведе до нежелани действия, като например промяна на паролата на потребителя или изтриване на неговата информация.

Пример: Ако уеб приложение не използва CSRF защита, злонамерени хора могат да прехвърлят автентикационната сесия на потребителя и да изпратят заявки, които да имат вреден ефект върху неговата информация.

Предотвратяване: Уеб приложенията трябва да използват CSRF защита, като например генериране на уникален токен за всяка сесия.

Недостатъчна защита на паролите – това е уязвимост, която може да доведе до компрометиране на потребителските акаунти. Това може да се случи, ако паролите на потребителите не са достатъчно сложни и не се криптират на сървъра.

Пример: Ако уеб приложение съхранява паролите на потребителите в незащитен вид, злонамерени хора могат да ги използват, за да се впишат в системата и да имат достъп до личната информация на потребителите.

Предотвратяване: Уеб приложенията трябва да използват силни пароли, криптиране на паролите на сървъра и двуфакторна аутентикация.

Недостатъчна защита на уеб сървърите – това е уязвимост, при която злонамерени хора могат да използват слабости в уеб сървърите, за да имат достъп до личната информация на потребителите или да нанесат щети на системата.

8.

REST API е архитектурен стил за създаване на уеб услуги, които използват HTTP протокола за изпращане на данни между клиенти и сървъри. Тук са основните принципи на REST API:

Клиент-сървър архитектура: REST API се базира на разделянето на клиента и сървъра, което позволява на всеки един от тях да бъде развит и мащабиран независимо.

Без състояние (stateless): Всяка заявка към сървъра трябва да съдържа всички необходими данни, които са необходими за да бъде изпълнена заявката. Сървърът не запазва състоянието на клиента между заявките, което позволява по-голяма гъвкавост и мащабируемост.

Кеш: Сървърите трябва да изпращат подходящи заглавки (HTTP headers), които позволяват на клиентите да кешират (запазват) отговорите. Това намалява времето за заявката към сървъра и намалява натовареността върху мрежата.

Еднороден интерфейс: REST API трябва да има еднороден интерфейс, който позволява на клиентите да извличат информация от сървъра. Това включва използването на стандартни HTTP методи като GET, POST, PUT и DELETE, като също така включва и предоставянето на достатъчно информация за ресурсите и начина, по който могат да бъдат достъпени.

Слоеста архитектура (layered system): REST API може да използва слоеста архитектура, която позволява на различни компоненти да комуникират през различни слоеве на приложението, без да знаят за другите компоненти.

Код по заявка (code on demand): Този принцип е опционален и изисква REST API да позволява изпращане на изпълним код като JavaScript. Това може да се използва за да се разшири функционалността на клиента, като зареди допълнителни скриптове или приложения.

9. Ето един примерен код за работа с REST API в JSON формат:

import requests

# Използвайте метода get за да заявите заявката към REST API URL

response = requests.get('https://jsonplaceholder.typicode.com/users')

# Проверете дали заявката е била успешна

if response.status\_code == 200:

# Конвертирайте JSON отговора в Python обект

data = response.json()

# Изведете информацията за всички потребители

for user in data:

print('Name:', user['name'])

print('Email:', user['email'])

print('Company:', user['company']['name'])

print('')

else:

# Ако заявката не е била успешна, изведете грешката

print('Error:', response.status\_code)

Този код използва библиотеката requests, която позволява да се правят заявки към REST API. Заявката към REST API се прави с метода get, който приема URL-а на REST API. Отговорът на заявката се получава чрез response обекта.

Когато отговорът е в JSON формат, той може да бъде конвертиран в Python обект чрез метода json(). Така може да се обработва информацията отговорена от REST API.

В този пример, данните за потребителите се обработват чрез for цикъл. За всеки потребител, се извеждат неговото име, електронна поща и име на компанията, за която работи.

Ако заявката не е била успешна, грешката се извежда на екрана.