Список вопросов

(Ƶ) – смотрите дополнительный материал

1. Библиотека Swing, общие черты и особенности.

Основа:

Swing является частью JFC (Ƶ), входит в Java Core (с Java 1.2), следует модели MVC (Ƶ), является одним из первых решений использующих «Легковесный» стиль компонентов. «Легковесный» стиль подразумевает под собой отрисовку элементов «самими компонентами», а не самой операционной системой, в отличии от AWT, который использует системную отрисовку компонентов. Таким образам данный стиль использует больше ресурсов (медленнее работает), но при этом позволят достичь большей гибкости при проектировании и динамически менять L&F (Ƶ). Это работает примерно так: Swing создаёт системное окно, на котором отрисовываются базовые компоненты (контейнеры) такие как JFrame, и относительно них отрисовываются остальные, т. о. любой компонент отрисовывается относительно своего родителя. Благодаря динамическому L&F можно контролировать отображение элементов, и менять их в зависимости от

системы (используя системный стиль). Мы так же можем комбинировать элементы Swing и AWT, что не рекомендуется.

Доп. Материалы:

JFC (Java Foundation Classes) - набор библиотек на языке Java, предоставляющих программам на Java удобный API для создания GUI.

MVC (Model-View-Controller) – архитектура, при которой: модель (Model) хранит важные данные компонента и обеспечивает программный интерфейс к ним; вид (View) занимается внешним видом; контроллер (Controller) же управляет компонентом в целом, получая сигналы от вида и уведомляя об изменениях модель компонента.

L&F (Look and Feel) – термин, использующийся для определения внешнего вида (Look) и поведения (Feel) элемента.

2. Виды контейнеров в Swing.

Контейнер – базовый элемент GUI. Swing по большому счёту представляет два вида контейнеров: контейнеры на основе AWT и тяжеловесные контейнеры. Так же Swing представляет контейнеры для апплетов (Ƶ) и диалоговых окон, однако они не предоставляют функционал полноценных Swing контейнеров. Тим образом Swing представляет нативные JFrame и JWindow (Ƶ) контейнеры, а также JDialog (диалоговых окон) и JApplet (апплетов) и контейнеры на основе JComponent (Ƶ). Вот некоторые примеры контейнеров на основе JComponent: JPanel – базовый; JScrollPane – контейнер прокрутки; JSplitPane – контейнер, разделённый полоской на две части («окна»), которую можно двигать меня размеры «окон» контейнера.

Доп. Материалы:

Апплеты (Applets) – браузерные приложения написанные на Java.

JFrame vs JWindow:

JWindow – окно без рамки и элементов управления.

JFrame – окно с рамкой.

JComponent – базовый компонент Swing, от него наследуются все компоненты (кроме нативных контейнеров). А он в свою очередь наследуется от AWT-шного класса Container.

3. Элементы пользовательского интерфейса Swing.

Элементы пользовательского интерфейса реализуют паттерн MVC (Ƶ), таким образом элементы работают по принципу: Модель представляет данные компонента -> Представление представляет собой визуальное представление данных компонента -> Контроллер принимает входные данные от пользователя в представлении и отражает изменения в данных компонента. Компонент Swing имеет модель в качестве отдельного элемента, в то время как части View и Controller объединены в элементах пользовательского интерфейса. Из-за этого Swing имеет подключаемую архитектуру внешнего вида. Все элементы Swing наследуются от класса JComponent (Ƶ), существует множество элементов, вот некоторые, самые популярные из них: JLabel; JButton; JCheck Box; JList; JTextField; (Ƶ)

Доп. Материалы:

MVC (Model-View-Controller) – архитектура, при которой: модель (Model) хранит важные данные компонента и обеспечивает программный интерфейс к ним; вид (View) занимается внешним видом; контроллер (Controller) же управляет компонентом в целом, получая сигналы от вида и уведомляя об изменениях модель компонента.

JComponent – базовый компонент Swing, от него наследуются все компоненты (кроме нативных контейнеров). А он в свою очередь наследуется от AWT-шного класса Container.

JLabel – это компонент для размещения текста в контейнере.

JButton - создает кнопку.

JCheckBox – это графический компонент, который может находиться во включенном (true) или выключенном (false) состоянии.

JList - предоставляет список прокручиваемых текстовых элементов.

JTextField – это текстовый компонент, который позволяет редактировать одну строку текста.

4. Модель событий Swing. Интерфейс EventListener.

Swing представляет событийную модель, в которой мы явно можем выделить три объекта: само событие, слушатель и источник события. События в свою очередь можно описать как изменение состояния объекта (источника) и поделить на два типа: пользовательские события (переднего плана) и фоновые события (заднего плана), где первые будут созданы при взаимодействии пользователя с системой (нажатием на кнопку), вторые создаются конечным пользователем (системой), например окончание

таймера, ошибка, отсутствие ресурсов системы (памяти). Система событий реализована на механизме обратного вызова, следовательно, требует наличия источника (того, кто генерирует событие), слушателя (тот, кто обрабатывает событие) и регистрации слушателя на определённое событие. Интерфейс EventListener является базовой реализацией слушателя события, от него наследуются все остальные слушатели событий, например: KeyListener; MouseMotionListener; FocusListener (Ƶ).

Доп. Материалы:

KeyListener - интерфейс используется для получения событий с клавиатуры (нажатие клавиш).

MouseMotionListener - Этот интерфейс используется для получения событий движения мыши.

FocusListener - интерфейс используемы для получения событий фокуса (наведения) мыши.

5. Менеджеры компоновки Swing.

Менеджеры компоновки Swing регулируют как компоненты будут размещены внутри контейнера. Фактически менеджер автоматически размещает компоненты в зависимости от выбранного типа менеджера и заданных параметров компонента. Базовые реализации (LayoutManager и LayoutManager2) представляет AWT, однако Swing расширяет реализации AWT собственными, вот некоторые из них: BorderLayout, FlowLayout, GridLayout. (Ƶ)

Доп. Материалы:
BorderLayout - размещает компоненты в пяти областях: восток, запад, север, юг и центр.
FlowLayout - является макетом по умолчанию. Это расположение компонентов в направленном потоке.

GridLayout - управляет компонентами в форме прямоугольной сетки.

6. GUI Designer Swing

7. Текстовые поля в Swing

8. Компонент управления JButton в Swing

9. Платформа JavaFX, особенности, компоненты

JavaFX - платформа на основе Java для создания приложений с насыщенным графическим интерфейсом. Может использоваться как для создания настольных приложений, запускаемых непосредственно из-под операционных систем, так и для интернет-приложений (RIA), работающих в браузерах, и для приложений на мобильных устройствах. JavaFX призвана заменить использовавшуюся ранее библиsотеку Swing. Платформа JavaFX конкурирует с Microsoft Silverlight, Adobe Flash и аналогичными системами.

Особенности JavaFX:

- Декларативный язык разметки: JavaFX использует язык разметки FXML, который позволяет описывать пользовательский интерфейс в виде иерархии компонентов, аналогично HTML или XML. Это упрощает разработку и поддержку интерфейса.
- Мультимедийная поддержка: JavaFX обладает мощными возможностями работы с мультимедиа, включая воспроизведение видео и аудио, создание анимаций и работу с графикой.
- Поддержка стилей и CSS: JavaFX поддерживает применение стилей и каскадных таблиц стилей (CSS) для настройки внешнего вида компонентов интерфейса. Это позволяет легко изменять оформление приложения без изменения кода.
- Встроенная поддержка многопоточности: JavaFX предоставляет механизмы для работы с многопоточностью, что позволяет выполнять длительные операции в фоновом режиме и не блокировать пользовательский интерфейс.
- Графические эффекты и трансформации: JavaFX обладает богатыми возможностями для применения графических эффектов, таких как размытие, тени, переходы и др. Также доступны трансформации объектов, например, масштабирование, поворот и сдвиг.

JavaFX состоит из трёх основных компонентов:

- **Stage** - это вышестоящее окно, в котором мы можем вводить все элементы графического интерфейса. Он содержит все объекты приложения JavaFX и представлен классом Stage пакета javafx.stage. По сути, он и является точкой входа.
- **Scene** - это физическое содержимое приложения JavaFX. В нем находится все содержимое графы Stage. Класс Scene пакета javafx.scene

представляет объект сцены. В одном экземпляре объект сцены добавляется только к одному этапу.

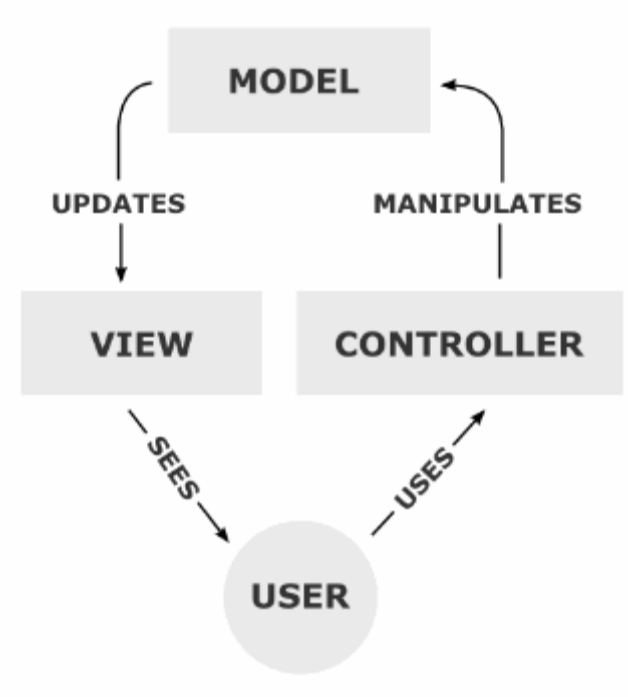
- **Node** - визуальное содержимое Scene. Сюда могут входить:
  - геометрические объекты (как 2D, так и 3D), а также такие фигуры, как прямоугольник, круг, многоугольник;
  - элементы, с которыми взаимодействуют пользователи: кнопки, поля выборов, разнообразные флажки и многое другое;
  - панели - различные панели сетки и границ;
  - объекты мультимедиа - изображения, видео и аудио.

10. Шаблон MVC (Model-View-Controller) в Sping.

MVC (Model-View-Controller) – шаблон программирования, разделяющий данные приложения и управляющей логики на три отдельных компонента:

- Model (Модель) - предоставляет данные и реагирует на команды контроллера, изменяя своё состояние. В Spring MVC модель может быть представлена классами Java, которые представляют сущности приложения, например пользователи, заказы и т. д. Модель может быть связана с базой данных или другими источниками данных.
- View (Представление) - отвечает за отображение данных модели пользователю, реагируя на изменения модели. В Spring MVC представление обычно представлено в виде шаблонов, например, JSP (JavaServer Pages), Thymeleaf или HTML. Шаблоны содержат HTML-разметку с вставками кода для отображения данных из модели.
- Controller (Контроллер) - интерпретирует действия пользователя, оповещая модель о необходимости изменений. В Spring MVC контроллеры обычно представлены в виде классов Java, у которых есть аннотации, такие как @Controller или @RestController. Контроллеры содержат методы, которые обрабатывают запросы, которые описываются в других аннотациях, например в @RequestMapping.

MVC позволяет изменять каждый компонент веб-приложения независимо друг от друга для их простой разработки и поддержки.



11. Классы StringBuffer и StringBuilder.

Строки в Java реализованы в виде объектов класса String. Они финализированы и неизменяемы, вследствие этого при любых манипуляциях с ними всегда создается новая строка, что делает работу со строками весьма ресурсоёмким процессом. Если строки необходимо часто менять, то для этого есть StringBuffer и StringBuilder.

Разница между ними заключается в следующем:

- String финализирован и неизменяемы (final), тогда как StringBuffer и StringBuilder являются изменяемыми классами.
- StringBuffer является потокобезопасным и синхронизированным, тогда как StringBuilder - нет. Вот почему StringBuilder быстрее, чем StringBuffer.

Для манипуляций со строками в среде без многопоточности лучше использовать StringBuilder, иначе использовать класс StringBuffer.

Основными операциями в StringBuilder и StringBuffer являются методы append и insert, которые принимают данные любого типа. Каждый из них эффективно преобразует заданные данные в строку, а затем добавляет или вставляет символы этой строки в конструктор строк.

Метод append всегда добавляет эти символы в конце конструктора, метод insert добавляет символы в указанной точке.

12. Архитектура JDBC (Java DataBase Connectivity). Двух и трехуровневые модели доступа к базе данных. Преимущества и недостатки JDBC.

JDBC (Java Database Connectivity) - это API в языке Java, предоставляющее возможность взаимодействия с базами данных. Оно обеспечивает доступ к базам данных через стандартные SQL-запросы и позволяет Java-приложениям работать с различными СУБД, такими как Oracle, MySQL, PostgreSQL и другими.

Основные компоненты архитектуры JDBC:

- Драйвер JDBC (JDBC Driver): представляет собой набор классов, реализующих спецификацию JDBC. Каждая база данных требует своего драйвера, который обеспечивает соединение с конкретной СУБД и выполнение SQL-запросов.
- Менеджер драйверов JDBC (JDBC Driver Manager): обеспечивает загрузку и управление драйверами JDBC. Он отвечает за поиск и выбор подходящего драйвера для конкретной базы данных.

Основные интерфейсы JDBC:

- Connection: предоставляет методы для установления соединения с базой данных, выполнения транзакций, получения объектов Statement и PreparedStatement для выполнения SQL-запросов.
- Statement и PreparedStatement: Эти интерфейсы предоставляют методы для выполнения SQL-запросов и обработки результатов. Statement используется для выполнения статических SQL-запросов, а PreparedStatement - для выполнения динамических запросов с параметрами.
- ResultSet: представляет результаты выполнения запроса и предоставляет методы для извлечения данных из результирующего набора.

В двухуровневой модели доступа к базе данных приложение напрямую взаимодействует с базой данных через драйвер JDBC. Оно самостоятельно выполняет SQL-запросы, получает результаты и обрабатывает их. В этом случае бизнес-логика и код доступа к данным находятся в том же слое.

В трёхуровневой модели пользовательские команды или запросы отправляются службам среднего уровня, из которых команды снова отправляются в источник данных. Результаты отправляются обратно на средний уровень и оттуда пользователю. Этот тип модели очень полезен руководителям информационных систем управления, поскольку он упрощает управление доступом и обновление корпоративных данных. Развертывание

приложений также становится простым и обеспечивает выигрыш в производительности.

13. Массивы Java: объявление, инициализация. Основные методы класса Arrays. Доступ к элементам массивов, итерация массивов. Двумерные массивы.

Массив представляет набор однотипных значений. Объявление массива похоже на объявление обычной переменной, которая хранит одиночное значение, причем есть два способа объявления массива:

тип_данных название_массива[]; //или

тип_данных[] название_массива;

Также можно сразу при объявлении массива инициализировать его:

int nums[] = new int[4];   // массив из 4 чисел

int[] nums = new int[5]; //массив из 5 чисел

При подобной инициализации все элементы массива имеют значение по умолчанию. Однако также можно задать конкретные значения для элементов массива при его создании:

int[] nums = { 1, 2, 3, 5 };

Класс java.util.Arrays предоставляет ряд полезных методов для работы с массивами, таких как сортировка, копирование, заполнение и т. д.

Основные методы класса Arrays:

- sort(array): сортирует элементы массива в порядке возрастания.
- binarySearch(array, key): выполняет бинарный поиск элемента в отсортированном массиве
- copyOf(array, length): создаёт копию массива указанной длины
- fill(array, value): заполняет массив указанным значением
- toString(array): возвращает строковое представление массива

Доступ к элементам массива осуществляется по индексу, начиная с 0.

Массивы бывают многомерными. Наиболее известный многомерный массив - двухмерный, представляющий собой таблицу.

Инициализация двухмерного массива выглядит следующим образом:

int[][] nums = { { 0, 1, 2 }, { 3, 4, 5 } }; //или

int[][] nums = new int[2][3];

14. Иерархия наследования Java. Преобразование типов при наследовании. Ключевое слово instanceof.

Иерархия наследования в Java позволяет создавать отношения между классами, где один класс наследует свойства и методы от другого класса. В этой иерархии возможно преобразование типов объектов при наследовании, и оно осуществляется с помощью upcasting и downcasting.

Upcasting (восходящее преобразование): Upcasting происходит, когда объект подкласса приводится к типу суперкласса. Восходящее преобразование является неявным и выполняется автоматически компилятором.

Downcasting (нисходящее преобразование): Downcasting происходит, когда объект суперкласса приводится к типу подкласса. Нисходящее преобразование является явным и требует явного приведения типа с использованием оператора (тип).

Ключевое слово instanceof используется для проверки принадлежности объекта к определенному типу. Оно возвращает значение true, если объект относится к указанному типу или его подклассу, и false в противном случае.

15. Интерфейсы Java: определение интерфейса, реализация интерфейса. Преимущества применения интерфейсов. Переменные интерфейсов. Наследование интерфейсов. Методы по умолчанию. Статические методы интерфейсов.

В языке Java мы можем наследовать только от одного класса. Подобную проблему частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.

Чтобы определить интерфейс, используется ключевое слово interface.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Чтобы класс применил интерфейс, надо использовать ключевое слово implements.

Наследование интерфейсов аналогично наследованию классов (см. пункт 14).

В JDK 8 были добавлены методы по умолчанию - обычные методы без модификаторов, которые помечаются ключевым словом default. Теперь интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод.

Начиная с JDK 8 в интерфейсах доступны статические методы (static) - они аналогичны методам класса. Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод.

16. Байтовые потоки InputStream и OutputStream. Консольный ввод и вывод Java. Символьные потоки данных. Абстрактные классы Writer, Reader.

Поток ввода - объект, из которого можно считать данные, поток вывода - объект, в который можно записывать данные. Например, если надо считать содержание файла, то применяется поток ввода, а если надо записать в файл - то поток вывода.

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: InputStream (представляющий потоки ввода) и OutputStream (представляющий потоки вывода). Но поскольку работать с байтами не очень удобно, то для работы с потоками символов были добавлены абстрактные классы Reader (для чтения потоков символов) и Writer (для записи потоков символов). Все остальные классы, работающие с потоками, являются наследниками этих абстрактных классов.

Основные методы класса InputStream:

- available(): возвращает количество байтов, доступных для чтения в потоке
- read(): возвращает целочисленное представление следующего байта в потоке. Когда в потоке не останется доступных для чтения байтов, данный метод возвратит число -1
- skip(long number): пропускает в потоке при чтении некоторое количество байт, которое равно number

Основные методы класса OutputStream:

- write(int b): записывает в выходной поток один байт, который представлен целочисленным параметром b
- flush(): очищает буфер вывода, записывая все его содержимое

Основные методы класса Reader:

- read(): возвращает целочисленное представление следующего символа в потоке. Если таких символов нет, и достигнут конец файла, то возвращается число -1
- skip(long count): пропускает количество символов, равное count. Возвращает число успешно пропущенных символов

Основные методы класса Writer:

- write(int c): записывает в поток один символ, который имеет целочисленное представление
- append(char c): добавляет в конец выходного потока символ c. Возвращает объект Writer
- flush(): очищает буферы потока

Также есть общий метод close(), который закрывает поток

17. Основные фреймворки и задачи, решаемые Spring.

Spring - это один из самых популярных фреймворков для разработки приложений на языке Java. Он предоставляет множество функций и инструментов для упрощения разработки приложений и решения различных задач.

Основные фреймворки и задачи, решаемые с помощью Spring:

- Spring Core: предоставляет основные функции и возможности фреймворка, такие как управление жизненным циклом объектов (IoC - Inversion of Control) и внедрение зависимостей (DI - Dependency Injection). Он позволяет создавать модульные и масштабируемые приложения, уменьшает связность между компонентами и облегчает тестирование.
- Spring MVC (Model-View-Controller): является веб-фреймворком Spring, предназначенным для разработки веб-приложений. Он обеспечивает разделение бизнес-логики, представления и обработки запросов. Spring MVC обеспечивает удобные средства для обработки HTTP-запросов, маршрутизации, обработки форм, валидации и других функций, связанных с веб-разработкой.
- Spring Data: предоставляет удобные абстракции и инструменты для работы с различными хранилищами данных, включая реляционные базы данных, NoSQL-хранилища, поисковые системы и другие. Он позволяет сократить количество повторяющегося кода для доступа к

данным и обеспечивает удобные функции, такие как автоматическая генерация SQL-запросов и поддержка транзакций.

- Spring Security: фреймворк для обеспечения безопасности в приложениях. Он предоставляет механизмы аутентификации, авторизации, управления доступом и другие функции, связанные с обеспечением безопасности. Spring Security позволяет защитить веб-ресурсы, REST API и другие компоненты приложения от несанкционированного доступа и злоумышленников.
- Spring Boot: фреймворк, который упрощает создание автономных приложений на основе Spring. Он предоставляет конфигурацию по умолчанию, автоматическое управление зависимостями, встраиваемые контейнеры и другие функции для ускорения разработки. Spring Boot позволяет создавать приложения с минимальным объемом настроек и максимальной производительностью.

18. Spring Inversion of Control (IoC) контейнер Spring.

Inversion of Control (инверсия управления) — это некий абстрактный принцип, набор рекомендаций для написания слабо связанного кода. Суть которого в том, что каждый компонент системы должен быть как можно более изолированным от других, не полагаясь в своей работе на детали конкретной реализации других компонентов.

IoC-контейнер - основной компонент фреймворка Spring. Он предоставляет механизмы для управления жизненным циклом объектов и внедрения зависимостей (Dependency Injection, DI) в приложениях.

Основные концепции и возможности контейнера IoC в Spring:

- Управление жизненным циклом объектов: Контейнер IoC Spring создает объекты, управляет их инициализацией и уничтожением, а также обеспечивает доступ к ним при необходимости. Это освобождает разработчика от необходимости явно создавать и управлять объектами, что упрощает кодирование и улучшает поддержку приложения.
- Внедрение зависимостей (Dependency Injection, DI): Контейнер IoC автоматически связывает зависимости между классами. Вместо того чтобы создавать и управлять зависимыми объектами самостоятельно, вы определяете зависимости в классе и позволяете контейнеру Spring автоматически внедрять эти зависимости при создании экземпляров объектов. Это делает код более гибким, расширяемым и тестируемым.
- Конфигурация через XML, аннотации или Java-код: Контейнер IoC Spring позволяет конфигурировать приложение с использованием различных подходов. Вы можете использовать XML-файлы

конфигурации, где определяются бины и их зависимости. Вы также можете использовать аннотации, чтобы указать компоненты и их связи. Или вы можете использовать Java-конфигурацию, где конфигурация выполняется с помощью Java-кода. Это позволяет выбрать подход, который лучше всего соответствует вашим потребностям и предпочтениям.

- Автоматическое разрешение зависимостей: Контейнер IoC Spring обеспечивает автоматическое разрешение зависимостей между классами. Он анализирует зависимости, определенные в классах, и автоматически находит и связывает соответствующие бины. Это позволяет уменьшить объем необходимого конфигурационного кода.

19. Dependency Injection (DI) в Spring.

Внедрение зависимостей - это стиль настройки объекта, при котором поля объекта задаются внешней сущностью. Другими словами, объекты настраиваются внешними объектами. DI - это альтернатива самонастройке объектов.

Создание объектов непосредственно внутри класса является негибким, поскольку оно привязывает и делает невозможным последующее изменение экземпляра отдельно от класса. Это лишает класс возможности класс к определённым объектам повторного использования, если требуются другие объекты, и затрудняет тестирование класса, поскольку реальные объекты нельзя заменить макетными.

Класс больше не отвечает за создание объектов, которые ему требуются, и ему не нужно делегировать создание экземпляра объекту factory, как в шаблоне проектирования Abstract Factory.

20. Жизненный цикл объекта Bean Spring.

Жизненный цикл объекта Bean состоит из нескольких фаз:

- Конфигурация: в этой фазе бин создается на основе его определения в конфигурационном файле или аннотациях. Spring контейнер создает экземпляр бина и настраивает его свойства и зависимости.
- Инициализация: в этой фазе происходит инициализация бина. Если у бина определен метод инициализации (через аннотацию @PostConstruct или интерфейс InitializingBean), то он будет вызван после создания и настройки бина. (Не является обязательным)
- Использование: в этой фазе бин готов к использованию. Вы можете получить ссылку на бин из Spring контейнера и вызывать его методы.

- Уничтожение: когда контекст Spring закрывается или бин больше не нужен, наступает фаза уничтожения. Если у бина определен метод уничтожения (через аннотацию @PreDestroy или интерфейс DisposableBean), то он будет вызван перед уничтожением бина. (Не является обязательным)

Стоит сказать, что spring автоматически управляет жизненным циклом bean, что упрощает разработку.

21. Конфигурация ApplicationContext с помощью xml в Spring.

Конфигурация ApplicationContext с использованием XML в Spring выполняется путем создания специального XML-файла, который содержит определения бинов и другие конфигурационные настройки.

Сначала создается файл конфигурации, в котором определяется корневой элемент <beans>, в нем будут содержаться остальные элементы конфигурации. Внутри корневого элемента <beans> определяем бины используя <bean>, каждый элемент представляет собой определение одного бина и содержит его атрибуты. Внутри <bean> определяем атрибуты и свойства бина, типа class, id и тд. После создания xml файла, создаем ApplicationContext и получаем бины из контекста используя их идентификаторы или имена.

22. Область видимости Bean в Spring.

Существует несколько областей видимости бинов, они определяют, как долго существует экземпляр бина и как он доступен в приложении.

- Singleton – при использовании Singleton создается только один экземпляр бина для всего контекста приложения. Этот экземпляр возвращается каждый раз, когда запрашивается бин с тем же идентификатором (id). Singleton является областью видимости по умолчанию, если не указан другой скоуп.
- Prototype - При использовании Prototype для бина каждый запрос возвращает новый экземпляр. Это означает, что каждый раз, когда бин запрашивается из контекста, создается и возвращается новый экземпляр.
- Request - применяется только в веб-приложениях. При использовании Request каждый HTTP-запрос создает новый экземпляр бина, который доступен только для этого запроса. После завершения запроса экземпляр уничтожается.

23. Фабричные или factory-методы в Spring.

Фабричные методы используются для создания экземпляров бинов вместо использования стандартного конструктора. Этот метод может быть определен в любом классе, не обязательно в классе самого бина. В XML-конфигурации или необходимо указать имя фабричного метода при помощи атрибута “factory - method”.

Фабричные методы можно использовать двумя способами:

1. Статический фабричный метод – определяется как статический метод в классе фабрике. Используется для создания бина, без необходимо создания экземпляра фабричного класса.
2. Фабричный метод экземпляра - определяется как нестатический метод в классе-фабрике, который создает и возвращает экземпляр бина. Этот метод вызывается на уже созданном экземпляре фабричного класса, он должен быть заранее определен как бин.

В обоих случаях фабричные методы предоставляют способ гибкого создания экземпляров бинов и позволяют выполнять дополнительную логику перед их созданием. Выбор между статическим фабричным методом и фабричным методом экземпляра зависит от контекста использования и логики создания объектов в вашем приложении.

24. Конфигурация ApplicationContext с помощью аннотаций в Spring.

Аннотации представляют собой удобный способ настройки бинов.

Список аннотаций, используемый для конфигурации ApplicationContext:

- @Configuration: применяется к классу, который служит конфигурационным классом. Внутри этого класса вы можете определить бины с помощью аннотаций, таких как @Bean, @ComponentScan и других.
- @ComponentScan: используется для автоматического сканирования и обнаружения компонентов (бинов) в вашем приложении. Она указывает контексту Spring, в каком пакете или пакетах искать компоненты для регистрации как бины.
- @Bean: используется для определения метода, который создает и возвращает экземпляр бина. Этот метод должен быть определен в классе, помеченном аннотацией @Configuration. Результатом выполнения этого метода будет зарегистрированный в контексте Spring бин.

- @Autowired: используется для внедрения зависимостей. Она может быть применена к полю, конструктору или методу с аргументами, и Spring автоматически разрешит соответствующую зависимость и внедрит ее в бин.
- @Value: используется для внедрения значений из внешних источников (например, из файла конфигурации) в поля бина.
- @Qualifier: используется для уточнения зависимости, когда в контексте есть несколько бинов одного типа. С помощью этой аннотации вы можете указать конкретный бин, который должен быть внедрен в зависимость.
- @Scope: используется для определения области видимости бина. С помощью нее вы можете указать, должен ли бин быть создан в единственном экземпляре (singleton), для каждого запроса (prototype) или в другой области видимости.

25. Связывание в Spring, аннотация @Autowired.

Аннотация @Autowired позволяет автоматически разрешить и связать зависимости без явного создания экземпляров и установки связей вручную.

Когда мы используем аннотацию @Autowired, Spring будет искать бин, соответствующий указанному типу зависимости, и автоматически внедрять его в ваш бин.

Используя аннотацию @Autowired, мы так же можем ее комбинировать с другими аннотациями, вроде @Value для внедрения значений из внешних источников. Так же важно сказать, что для использования @Autowired необходимо настроить компонентное сканирование или явно определить бины в конфигурационном классе.

26. Архитектурный стиль REST.

Архитектурный стиль REST (Representational State Transfer) является широко применяемым и популярным подходом к проектированию распределенных систем, особенно веб-сервисов и API. REST определяет набор принципов и ограничений, которые позволяют создавать гибкие, масштабируемые и легко управляемые системы.

Основные принципы архитектуры REST

- Клиент-серверная архитектура: REST предполагает разделение клиентской и серверной частей системы. Клиенты и серверы

взаимодействуют посредством стандартизированных интерфейсов, обычно предоставляемых через HTTP протокол.

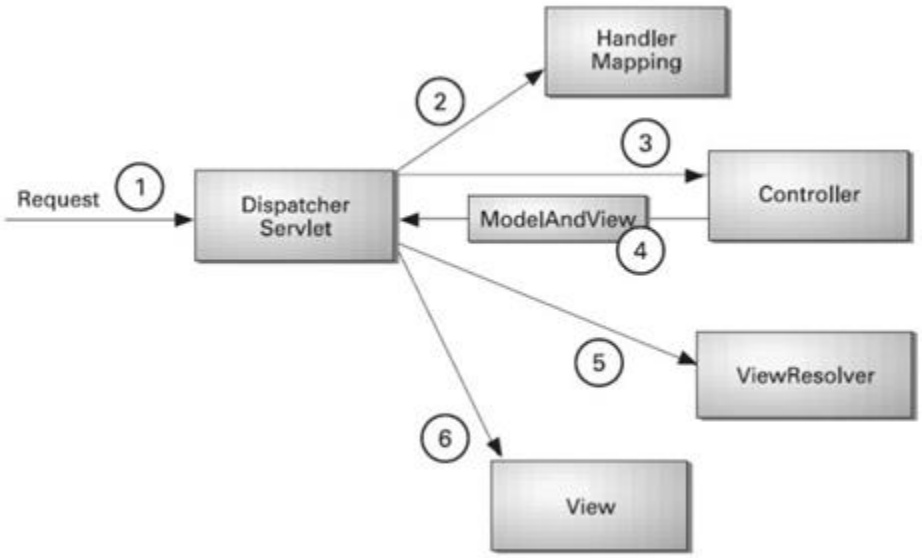
- Без состояния (Stateless): Сервер не хранит информацию о состоянии клиента между запросами. Каждый запрос от клиента должен содержать все необходимые данные для выполнения операции. Это повышает масштабируемость и упрощает управление состоянием системы.
- Кэширование: REST поддерживает использование кэшей на стороне клиента или сервера для повышения производительности. Клиенты могут кэшировать ответы сервера, чтобы избежать повторных запросов к серверу.
- Единообразный интерфейс: REST определяет единообразный интерфейс взаимодействия между клиентом и сервером. Он включает использование уникального идентификатора ресурса (URI) для идентификации ресурсов, стандартных методов HTTP (например, GET, POST, PUT, DELETE) для работы с ресурсами, а также использование гипермедиа (например, ссылок) для связывания ресурсов.
- Слоистая система (Layered System): REST поддерживает слоистую архитектуру, где клиенты могут взаимодействовать с промежуточными серверами или прокси-серверами. Это позволяет упростить систему и добавлять масштабируемость и безопасность.

27. Spring Web-MVC, основная схема и логика работы.

Spring MVC – веб-фреймворк, призванный упростить разработку веб-приложений. Опираясь на шаблон модель–представление–контроллер (Model-View-Controller, MVC), фреймворк Spring MVC помогает строить веб-приложения, столь же гибкие и слабо связанные, как сам фреймворк Spring.

Схема работы фреймворка Spring MVC

Схема работы фреймворка выглядит следующим образом:



Краткое описание схемы работы Spring MVC звучит следующим образом:

- вначале **DispatcherServlet** (диспетчер сервлетов) получает запрос, далее он смотрит свои настройки, чтобы понять какой контроллер использовать (на рисунке **Handler Mapping**);
- после получения имени контроллера запрос передается на обработку в этот контроллер (на рисунке **Controller**). В контроллере происходит обработка запроса и обратно посылается **ModelAndView** (модель — сами данные; view (представление) — как эти данные отображать);
- **DispatcherServlet** на основании полученного **ModelAndView**, должен определить, какое представление будет выводить данные. Для этого используется **арбитр представлений** (**View Resolver**), который на основании полученного логического имени представления возвращает ссылку на файл **View**;
- в представление передаются данные (Model) и обратно, если необходимо, посылается ответ от представления.

Логика работы:

1. Клиент отправляет HTTP-запрос на сервер.
2. Диспетчер сервера, который является центральным компонентом фреймворка, принимает запрос и определяет, какой контроллер должен обработать этот запрос, основываясь на маппинге URL-адресов.
3. Диспетчер передает запрос соответствующему контроллеру для дальнейшей обработки.
4. Контроллер получает запрос и обрабатывает его. Это может включать выполнение бизнес-логики, вызов сервисных методов, извлечение данных из базы данных и т. д.

5. Контроллер формирует модель данных, которая будет передана в представление для отображения.
6. Контроллер выбирает подходящее представление, которое будет использоваться для отображения данных. Это может быть JSP, Freemarker, Thymeleaf, AngularJS или другие представления.
7. Контроллер передает модель данных в представление.
8. Представление получает модель данных и использует их для формирования вывода - HTML-страницы, JSON-ответа или любого другого формата.
9. Сформированный ответ отправляется обратно на сервер.
10. Диспетчер сервера принимает ответ и отправляет его обратно клиенту

28. Класс DispatcherServlet, его функции.

Класс DispatcherServlet выполняет роль фронтенд контроллера в веб приложении.

Функции DispatcherServlet:

- Прием HTTP-запросов: DispatcherServlet принимает все входящие HTTP-запросы от клиентов, которые направляются на веб-приложение.
- Координация обработки запросов: Он координирует обработку запросов, определяя, какой контроллер должен обработать каждый запрос. DispatcherServlet основывается на конфигурации URL-маппинга, которую можно настроить в приложении.
- Управление жизненным циклом контроллеров: DispatcherServlet отвечает за создание экземпляров контроллеров и управление их жизненным циклом. Он создает контроллеры на основе их конфигурации и требуемых зависимостей.
- Выполнение бизнес-логики: когда DispatcherServlet определяет контроллер, который должен обработать запрос, он передает управление соответствующему контроллеру. Контроллер выполняет необходимую бизнес-логику, взаимодействует с сервисами и базой данных для получения и обработки данных.
- Передача данных в представление: после выполнения бизнес-логики контроллер формирует модель данных, которую он передает обратно в DispatcherServlet. DispatcherServlet сохраняет эту модель данных в объекте ModelAndView, который используется для передачи данных в представление.
- Выбор представления: DispatcherServlet выбирает соответствующее представление на основе конфигурации и информации о запросе. Представление может быть JSP, Thymeleaf, Freemarker или другим типом представления.

- Генерация ответа: DispatcherServlet передает модель данных в выбранное представление для генерации конечного ответа. Представление использует данные из модели для формирования HTML-страницы, JSON-ответа или других форматов ответа, которые отправляются обратно клиенту в виде HTTP-ответа.

29. Маппинг в Spring.

Маппинг в Spring относится к процессу связывания (mapping) HTTP-запросов с методами обработки в вашем приложении. Это позволяет определить, какой метод должен быть вызван при получении определенного запроса.

При определении маппинга в Spring можно использовать различные критерии, такие как URL-шаблоны, HTTP-методы, параметры запроса, заголовки и другие атрибуты запроса. Каждый метод контроллера может быть аннотирован соответствующей аннотацией, которая определяет его маппинг.

Например, с помощью аннотации @RequestMapping или ее производных (например, @GetMapping, @Portmapping) можно указать URL-шаблон, который будет сопоставлен с запросом. Метод контроллера, помеченный этой аннотацией, будет вызван для обработки запроса, соответствующего указанному URL-шаблону.

Также можно использовать дополнительные аннотации и параметры для определения дополнительных условий маппинга. Например, аннотация @RequestParam позволяет извлекать параметры запроса, а @RequestHeader - заголовки запроса.

Маппинг в Spring является гибким и мощным механизмом, который позволяет точно определить, какие методы обработки должны быть вызваны для различных типов запросов.

30. Интерфейсы HttpServletRequest и HttpServletResponse.

Класс Servlet существует для стандартизации работы сервлета и контейнера. Непосредственно с этим классом программисты не работают. Ну или работают очень редко. Чаще всего используется класс HttpServlet, унаследованный от Servlet’a.

Самые популярные методы этого класса:

	Метод	Описание
1	`init()`	Вызывается один раз при загрузке сервлета
2	`destroy()`	Вызывается один раз при выгрузке сервлета
3	`service(HttpRequest, HttpResponse)`	Вызывается для каждого нового запроса к сервлету
4	`doGet(HttpRequest, HttpResponse)`	Вызывается для каждого нового GET-запроса к сервлету
5	`doPost(HttpRequest, HttpResponse)`	Вызывается для каждого нового POST-запроса к сервлету
6	`doHead(HttpRequest, HttpResponse)`	Вызывается для каждого нового HEAD-запроса к сервлету
7	`doDelete(HttpRequest, HttpResponse)`	Вызывается для каждого нового DELETE-запроса к сервлету
8	`doPut(HttpRequest, HttpResponse)`	Вызывается для каждого нового PUT-запроса к сервлету

Методы init() и destroy() унаследованы от класса Servlet. Поэтому если ты решишь переопределить их в своем сервлете, тебе нужно будет так же вызвать их реализацию из базового класса. Для этого используется команда super.имяМетода().

Метод service(HttpServletRequest, HttpServletResponse)
Если смотреть на обработку клиентского запроса с точки зрения сервлета, то дела обстоят примерно так.
Для каждого клиентского запроса контейнер (веб-сервер) создает объекты HttpServletRequest и HttpServletResponse, а затем вызывает метод service(HttpServletRequest request, HttpServletResponse response) у соответствующего сервлета. В него передаются эти объекты, чтобы метод мог взять нужные данные из request'а и положить результат работы в response.
У метода service() есть реализация по умолчанию. Если ее не переопределить, то выполняться будет именно она. Вот что он делает.
Метод service() определяет из request'а тип HTTP-метода (GET, POST, …) и вызывает метод соответствующий запросу.

31. Архитектурный стиль CRUD, его соответствие REST и HTTP.

Архитектурный стиль CRUD (Create, Read, Update, Delete) описывает базовые операции, которые можно выполнять с данными: создание (Create), чтение (Read), обновление (Update) и удаление (Delete).

REST (Representational State Transfer) - это набор принципов, которые описывают, как должны быть организованы системы для работы с распределенными данными в сети. REST основан на использовании стандартных HTTP-методов (GET, POST, PUT, DELETE) для работы с ресурсами.

Соответствие между архитектурным стилем CRUD, REST и HTTP основано на следующих принципах:

- Создание (Create): Операция создания нового ресурса в архитектурном стиле CRUD соответствует HTTP методу POST. В REST, для создания нового ресурса также используется метод POST. В запросе POST передается информация о новом ресурсе, который должен быть создан на сервере.
- Чтение (Read): Операция чтения данных из архитектурного стиля CRUD соответствует HTTP методу GET. В REST, для получения данных ресурса также используется метод GET. Запрос GET выполняется к определенному URL-адресу, который представляет собой путь к ресурсу на сервере.
- Обновление (Update): Операция обновления данных в архитектурном стиле CRUD соответствует HTTP методу PUT или PATCH. В REST, для обновления существующего ресурса используются методы PUT или PATCH. Запрос PUT или PATCH содержит новые данные, которыми необходимо обновить ресурс на сервере.
- Удаление (Delete): Операция удаления ресурса в архитектурном стиле CRUD соответствует HTTP методу DELETE. В REST, для удаления ресурса также используется метод DELETE. Запрос DELETE отправляется на URL-адрес ресурса, который должен быть удален.

RESTful API, основанный на архитектурном стиле CRUD, стремится к предоставлению простого и единообразного интерфейса для взаимодействия с ресурсами через HTTP протокол. Он использует различные HTTP методы для выполнения операций CRUD над ресурсами, а также статусы HTTP и форматы данных (например, JSON) для передачи информации между клиентом и сервером.

В заключение, архитектурный стиль CRUD соответствует REST и HTTP в том смысле, что CRUD операции могут быть выполнены с использованием соответствующих HTTP методов (POST, GET, PUT/PATCH, DELETE) и передачи данных через HTTP протокол. RESTful API, основанный на архитектурном стиле CRUD, позволяет разработчикам создавать гибкие и масштабируемые системы, используя стандартные проток

32. Шаблон Data Access Object (DAO).

Шаблон Data Access Object (DAO) используется для организации доступа к данным и абстрагирования слоя работы с базой данных.

DAO предоставляет абстрактный интерфейс для выполнения операций с данными, таких как создание, чтение, обновление и удаление (CRUD) объектов. Он скрывает детали работы с базой данных и предоставляет более высокоуровневый интерфейс для взаимодействия с данными.

Основные компоненты шаблона DAO:

- Интерфейс DAO: определяет операции, которые можно выполнять с данными, например, сохранение, поиск, обновление и удаление. Этот интерфейс служит в качестве контракта между клиентским кодом и реализацией DAO.
- Реализация DAO: представляет конкретную реализацию интерфейса DAO. В этой реализации выполняются фактические операции с базой данных, используя соответствующий ORM или низкоуровневые API доступа к данным.
- Модель данных: представляет объекты данных, с которыми работает DAO. Модель данных может быть сущностями, отображаемыми на таблицы базы данных, или другими объектами, представляющими данные.
- Фабрика DAO: иногда DAO может быть создан и возвращен через фабрику, которая обеспечивает инстанцирование правильного типа DAO в соответствии с определенными правилами или конфигурацией.

Преимущества использования шаблона DAO:

- Разделение ответственности: DAO отделяет бизнес-логику от деталей работы с базой данных. Это позволяет разделить обязанности между слоем доступа к данным и остальными компонентами приложения.
- Абстракция базы данных: DAO предоставляет абстракцию для работы с базой данных. Это позволяет легко заменять или изменять используемую базу данных без внесения изменений в другие части приложения.
- Тестируемость: DAO можно легко тестировать, поскольку его реализацию можно заменить фиктивным или инмемориным хранилищем данных для модульных тестов.
- Улучшение производительности: DAO может оптимизировать запросы к базе данных, кэшировать результаты или использовать другие техники для улучшения производительности доступа к данным.

Шаблон DAO является частью слоя доступа к данным в архитектуре приложения и помогает создать более гибкое, управляемое и тестируемое взаимодействие с базой данных.

33. Основные понятия Объектно-реляционного отображения (ORM - Object-Relational Mapping).

Основные понятия:

- Сущности (Entities): Сущности представляют объекты в приложении, которые хранятся в базе данных. Они соответствуют таблицам в базе данных и содержат поля, которые отображаются на столбцы таблицы. Сущности могут иметь отношения друг с другом, такие как один к одному, один ко многим или многие ко многим.
- Маппинг объектов на таблицы: ORM позволяет определить отображение между объектами и таблицами базы данных. Это включает сопоставление полей объектов с столбцами таблицы, определение первичных и внешних ключей, а также управление отношениями между сущностями.
- Персистентность: Персистентность означает способность сохранять объекты в базе данных и восстанавливать их из базы данных. ORM обеспечивает механизмы для сохранения, обновления и удаления сущностей, а также для выполнения запросов для поиска и фильтрации данных.
- Ленивая загрузка (Lazy Loading): Ленивая загрузка - это механизм, который позволяет отложить загрузку связанных сущностей до тех пор, пока они действительно не потребуются. Это позволяет улучшить производительность, избегая избыточной загрузки данных.
- Каскадные операции (Cascade Operations): Каскадные операции позволяют автоматически распространять операции сохранения, обновления или удаления на связанные сущности. Например, если вы сохраняете родительскую сущность, связанные дочерние сущности также будут сохранены.
- Язык запросов: ORM-фреймворки предоставляют свои языки запросов, которые позволяют выполнять сложные запросы к базе данных, используя объектную модель данных. Например, Hibernate предоставляет HQL (Hibernate Query Language), а EclipseLink - JPQL (Java Persistence Query Language).

34. Спецификация Java Persistence API (JPA).

Java Persistence API (JPA) - это спецификация Java для стандартизации и упрощения работы с объектно-реляционным отображением (ORM) в Java-приложениях. Спецификация JPA была разработана как часть Java EE (Enterprise Edition). Спецификация JPA определяет стандартный набор интерфейсов, аннотаций и поведения для работы с ORM в Java. Это позволяет разработчикам писать портативный код, который может быть использован с различными реализациями JPA, такими как Hibernate, EclipseLink и другими.

Вот некоторые ключевые особенности спецификации JPA:

- Аннотации и XML-конфигурация: JPA предоставляет возможность определения маппинга между объектами Java и таблицами базы данных с использованием аннотаций или XML-конфигурации. Это позволяет разработчикам выбрать наиболее удобный подход к определению метаданных сущностей.
- ORM-модель: JPA определяет объектную модель данных для представления сущностей и их отношений в базе данных. Она позволяет разработчикам работать с данными в терминах объектов Java, скрывая сложности работы с SQL и реляционной моделью данных.
- EntityManager: JPA предоставляет интерфейс EntityManager для выполнения операций с базой данных, таких как сохранение, поиск, обновление и удаление сущностей. EntityManager управляет жизненным циклом сущностей и обеспечивает контекст персистентности, где изменения объектов могут быть автоматически синхронизированы с базой данных.
- Язык запросов JPQL: JPA включает язык запросов JPQL (Java Persistence Query Language), который позволяет выполнять запросы к базе данных, используя объектную модель данных. JPQL аналогичен SQL, но работает с объектами Java, а не непосредственно с таблицами базы данных.
- Транзакции: JPA обеспечивает механизм управления транзакциями для обеспечения согласованности данных. Разработчики могут использовать аннотацию @Transactional для определения границы транзакции, или использовать программное управление транзакциями через EntityManager.
- Кэширование: JPA поддерживает кэширование данных для улучшения производительности. Разработчики могут настроить кэширование на уровне сущностей, запросов и запросов на уровне вторичного кэша.

35. Архитектура ORM Java Persistence API (JPA).

Java Persistence API (JPA) предоставляет стандартный подход к объектно-реляционному отображению (ORM) в Java. Архитектура JPA включает несколько ключевых компонентов:

- Сущности (Entities): Сущности представляют объекты, которые сохраняются в базе данных. Они обычно являются Java-классами, помеченными аннотацией @Entity, и содержат поля и методы для доступа к данным. Каждая сущность соответствует таблице в базе данных, а поля и свойства сущности отображаются на столбцы таблицы.
- EntityManager: EntityManager является центральным интерфейсом в JPA для выполнения операций с базой данных. Он отвечает за управление жизненным циклом сущностей, сохранение, обновление, удаление и поиск сущностей в базе данных. EntityManager также обеспечивает управление транзакциями и кэширование сущностей.
- Поставщик постоянства (Persistence Provider): Поставщик постоянства (например, Hibernate, EclipseLink или OpenJPA) является реализацией JPA и отвечает за взаимодействие с конкретной базой данных. Он реализует интерфейсы JPA, такие как EntityManager, и обеспечивает механизмы для выполнения операций с базой данных, отображения объектов на таблицы и выполнения запросов.
- Метаданные (Metadata): Метаданные определяют отображение между сущностями Java и таблицами базы данных. Они описывают имена таблиц, столбцов, связей и других атрибутов сущностей. Метаданные могут быть определены с использованием аннотаций, XML-файлов или с использованием программного кода.
- Query Language: JPA предоставляет язык запросов, известный как JPQL (Java Persistence Query Language), который позволяет выполнить запросы к базе данных, используя объектную модель данных, а не SQL. JPQL позволяет выполнять операции выборки, вставки, обновления и удаления сущностей.

36. Основные аннотации Java Persistence API (JPA).

Скорее всего речь идет про Spring JPA

Основные аннотации:

- @EnableJpaRepositories: Аннотация, которая указывает Spring на использование репозиториев JPA в приложении. Она обычно размещается на классе конфигурации и определяет базовый пакет для сканирования интерфейсов репозиториев.

- @EntityScan: Аннотация, которая указывает Spring на базовый пакет для сканирования сущностей JPA. Она обычно размещается на классе конфигурации и определяет пакет, в котором находятся классы с аннотацией @Entity.
- @Transactional: Аннотация, которая указывает Spring на необходимость управления транзакциями для метода или класса. Эта аннотация обычно размещается на методе или классе, который выполняет операции чтения/записи в базу данных, и позволяет автоматически управлять открытием, фиксацией и откатом транзакций.
- @Repository: Аннотация, которая помечает класс в качестве репозитория Spring Data. Эта аннотация позволяет Spring автоматически создавать реализацию репозитория на основе интерфейса и обеспечивает интеграцию с JPA.
- @Query: Аннотация, которая указывает на запрос JPA, который будет выполнен при вызове метода репозитория. Она позволяет определить пользовательский запрос с использованием языка запросов, такого как JPQL или SQL.
- @JoinColumn: Аннотация, которая определяет связь между двумя сущностями с использованием внешнего ключа. Она обычно применяется к полю или свойству, которое представляет внешний ключ, и определяет имя столбца, на который ссылается внешний ключ.
- @GeneratedValue: Аннотация, которая указывает стратегию генерации значений для первичного ключа. Она обычно применяется вместе с аннотацией @Id и определяет способ автоматической генерации значения первичного ключа.

37. Библиотека Hibernate, основные аннотации.

Основные аннотации Hibernate:

- @Entity: Аннотация, указывающая, что класс является сущностью, которая должна быть сохранена в базе данных. Класс, помеченный этой аннотацией, должен иметь открытый (public) или защищенный (protected) конструктор без аргументов.
- @Table: Аннотация, указывающая имя таблицы в базе данных, к которой будет отображаться сущность. Можно также указать дополнительные атрибуты, такие как имя схемы, имя каталога, индексы и другие параметры таблицы.
- @Id: Аннотация, указывающая поле или свойство, которое является первичным ключом сущности. Обычно применяется к полю, которое уникально и идентифицирует каждую запись в таблице.

- @GeneratedValue: Аннотация, указывающая стратегию генерации значений для первичного ключа. Может использоваться совместно с аннотацией @Id. Некоторые из наиболее распространенных стратегий генерации включают AUTO, IDENTITY, SEQUENCE и TABLE.
- @Column: Аннотация, указывающая отображение поля или свойства на столбец в таблице базы данных. Можно указать дополнительные атрибуты, такие как имя столбца, тип данных, ограничения на значения и другие параметры.
- @ManyToOne: Аннотация, указывающая связь "многие к одному" между двумя сущностями. Она указывает, что поле или свойство содержит ссылку на другую сущность. Это обычно применяется к полю, которое представляет внешний ключ в базе данных.
- @OneToMany: Аннотация, указывающая связь "один ко многим" между двумя сущностями. Она указывает, что поле или свойство содержит коллекцию других сущностей. Это обычно применяется к полю, которое содержит список или набор связанных сущностей.
- @ManyToMany: Аннотация, указывающая связь "многие ко многим" между двумя сущностями. Она указывает, что поле или свойство содержит коллекцию других сущностей, и обе сущности могут быть связаны с несколькими экземплярами друг друга. Для этой аннотации также требуется наличие таблицы-связи (join table).

38. Объявление сущности и таблицы в Hibernate.

Для объявления сущности и ее соответствующей таблицы в базе данных используются аннотации или файлы маппинга XML. Вот два распространенных способа объявления сущности и таблицы в Hibernate:

- Аннотации: В этом подходе используются аннотации Java, чтобы указать маппинг между сущностью и таблицей в базе данных. Ниже приведен пример объявления сущности с использованием аннотаций:

```java
    import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

    @Entity
    @Table(name = "employees")
    public class Employee {
        @Id
        private Long id;

        private String name;
        private String department;
```

В приведенном примере аннотация @Entity указывает, что класс Employee является сущностью, а аннотация @Table указывает имя таблицы в базе данных. Аннотация @Id указывает первичный ключ сущности.

- Файлы маппинга XML: В этом подходе используются XML-файлы для определения маппинга между сущностью и таблицей. Ниже приведен пример объявления сущности с использованием XML-файла маппинга:

```xml
    <!-- employees.xml -->
<hibernate-mapping>
    <class name="com.example.Employee" table="employees">
        <id name="id" type="long">
            <column name="id" />
            <generator class="assigned" />
        </id>
        <property name="name" column="name" type="string" />
        <property name="department" column="department" type="string" />
    </class>
</hibernate-mapping>
```

В этом примере файл маппинга employees.xml содержит информацию о классе Employee, его таблице и свойствах, таких как id, name и department.

Оба подхода (аннотации и XML-файлы маппинга) предоставляют возможность определения маппинга между сущностями и таблицами в Hibernate. Выбор конкретного подхода зависит от предпочтений разработчика и требований проекта.

Интерфейс Session является одним из основных интерфейсов в Hibernate и представляет собой главную точку взаимодействия с базой данных. Он

предоставляет методы для выполнения операций CRUD (Create, Read, Update, Delete) и управления состоянием сущностей.

39. Интерфейс Session в Hibernate.

Интерфейс Session является одним из основных интерфейсов в Hibernate и представляет собой главную точку взаимодействия с базой данных. Он предоставляет методы для выполнения операций CRUD (Create, Read, Update, Delete) и управления состоянием сущностей.

Интерфейс Session можно получить из фабрики сессий (SessionFactory) с помощью метода openSession(). Обычно создается один экземпляр Session для каждой операции базы данных или транзакции.

Основные методы:

- save(Object entity): Метод сохраняет новую сущность в базе данных или обновляет существующую, если она уже существует.
- get(Class<T> entityClass, Serializable id): Метод загружает сущность из базы данных по ее идентификатору.
- load(Class<T> entityClass, Serializable id): Метод лениво загружает сущность из базы данных по ее идентификатору. Фактические данные будут извлечены только при обращении к свойствам сущности.
- update(Object entity): Метод обновляет существующую сущность в базе данных.
- delete(Object entity): Метод удаляет сущность из базы данных.
- createQuery(String queryString): Метод создает запрос на языке Hibernate Query Language (HQL) для выполнения сложных запросов к базе данных.
- flush(): Метод сбрасывает текущее состояние изменений сессии на базу данных. Это гарантирует, что все ожидающие изменения будут применены.
- clear(): Метод очищает текущее состояние сессии, сбрасывая все загруженные и измененные сущности.
- beginTransaction(): Метод начинает новую транзакцию базы данных.
- commit(): Метод фиксирует текущую транзакцию и сохраняет все изменения в базе данных.

40. Ассоциация сущностей в Hibernate.

Ассоциация сущностей в Hibernate - это способ связывания объектов разных классов в базе данных

Основные типы ассоциаций сущностей в Hibernate:

- Одно к одному (One-to-One): В данном типе ассоциации каждая сущность одного класса связана с одной сущностью другого класса. Например, у каждого пользователя может быть только один профиль. Для реализации такой ассоциации в Hibernate используются аннотации @OneToOne и @JoinColumn.
- Один ко многим (One-to-Many): В этом типе ассоциации одна сущность одного класса связана с несколькими сущностями другого класса. Например, у каждого автора может быть несколько книг. Для реализации такой ассоциации в Hibernate используется аннотация @OneToMany с атрибутом mappedBy или @JoinColumn.
- Многие ко многим (Many-to-Many): В данном типе ассоциации несколько сущностей одного класса связаны с несколькими сущностями другого класса. Например, у каждой книги может быть несколько авторов, и у каждого автора может быть несколько книг. Для реализации такой ассоциации в Hibernate используется аннотация @ManyToMany с использованием таблицы-связи (join table) или атрибута mappedBy.
- Вложенные коллекции (Nested Collections): В этом типе ассоциации одна сущность содержит коллекцию других сущностей. Например, у каждой страны может быть список городов. Для реализации такой ассоциации в Hibernate используется аннотация @ElementCollection.

41. Spring Boot: определение, характеристики, преимущества.

Spring Boot - это фреймворк, разработанный на основе Spring Framework, который упрощает создание и развертывание приложений Java. Он предоставляет удобные средства для настройки и автоматической конфигурации приложения, а также обеспечивает стандартизированный способ создания самостоятельных приложений.

Характеристики и преимущества:

- Простота использования: Spring Boot стремится упростить разработку приложений Java, предоставляя простой и интуитивно понятный подход. Он предлагает множество готовых настроек по умолчанию, автоматическую конфигурацию и удобные средства для разработки.

- Автоматическая конфигурация: Spring Boot основывается на принципе "конфигурации по соглашению" (convention over configuration), что позволяет автоматически настраивать приложение на основе классов пути, зависимостей и других факторов. Это снижает необходимость явной конфигурации и позволяет сосредоточиться на разработке бизнес-логики.
- Встроенные серверы приложений: Spring Boot поставляется с встроенными серверами приложений, такими как Apache Tomcat, Jetty или Undertow. Это позволяет запускать приложения без необходимости настройки отдельного сервера.
- Управление зависимостями: Spring Boot предлагает механизм автоматического управления зависимостями с использованием инструмента управления зависимостями, такого как Maven или Gradle. Он обеспечивает удобный способ добавления и обновления зависимостей проекта.
- Удобство тестирования: Spring Boot предоставляет удобные инструменты для тестирования приложений, включая возможность запуска тестов с использованием встроенных серверов приложений, автоматическую конфигурацию тестовых сред и поддержку различных фреймворков тестирования.
- Готовые стартовые пакеты: Spring Boot предлагает стартовые пакеты (starter packs), которые предварительно настроены для различных типов приложений, таких как веб-приложения, RESTful API, базы данных, безопасность и многое другое. Это позволяет быстро начать разработку с минимальной конфигурацией.
- Монолитные и микросервисные приложения: Spring Boot подходит как для разработки монолитных приложений, так и для создания микросервисных архитектур. Он предоставляет удобные инструменты для разделения функциональности на отдельные модули и управления множеством микросервисов.
- Поддержка большой экосистемы: Spring Boot интегрируется с широким спектром проектов и библиотек в экосистеме Spring, таких как Spring Data, Spring Security, Spring Cloud и других. Это обеспечивает гибкость и мощность разработки приложений, использующих эти технологии.

42. Spring Initializer, особенности и преимущества применения.

Spring Initializr - это онлайн-инструмент и CLI (Command Line Interface), предоставляемый Spring, который помогает разработчикам быстро и легко инициализировать новые проекты на основе Spring Framework. Это ускоряет разработку и упрощает конфигурацию

Особенности и преимущества:

- Простота инициализации проекта: Spring Initializr предоставляет простой и понятный интерфейс, в котором разработчик может выбрать необходимые компоненты и настройки для своего проекта. Это включает выбор версии Spring Framework, добавление зависимостей, настройку сборки проекта и другие параметры.
- Генерация структуры проекта: Spring Initializr автоматически генерирует иерархию файлов и структуру проекта на основе выбранных компонентов и настроек. Это включает настройку структуры директорий, файлов конфигурации, исходного кода и других ресурсов проекта.
- Управление зависимостями: Spring Initializr позволяет разработчикам управлять зависимостями проекта, путем выбора и добавления необходимых библиотек и фреймворков. Это включает популярные зависимости, такие как Spring Boot, Spring Data, Spring Security и многие другие.
- Интеграция со средами разработки: Spring Initializr предлагает возможность скачать сгенерированный проект в виде ZIP-архива, который можно импортировать в различные интегрированные среды разработки (IDE) такие как IntelliJ IDEA, Eclipse и другие. Это упрощает начало работы с проектом и интеграцию с инструментами разработки.
- Поддержка различных языков и фреймворков: Spring Initializr позволяет разработчикам выбирать язык программирования (Java, Kotlin, Groovy) и фреймворк (Spring Boot, Spring MVC, Spring Cloud) для своего проекта. Это дает возможность создавать приложения с использованием различных комбинаций технологий.
- Встроенная конфигурация и настройки: Spring Initializr предоставляет возможность выбора настроек и конфигурации проекта, таких как система сборки (Maven или Gradle), язык программирования, версия Java и другие параметры. Это помогает разработчикам быстро настроить проект согласно своим требованиям.
- Поддержка стартовых пакетов: Spring Initializr предлагает стартовые пакеты (starter packs), которые предварительно настроены для определенных целей, таких как веб-приложения, RESTful API, базы данных, безопасность и другие. Это упрощает начало работы с различными типами проектов и предоставляет базовую конфигурацию и зависимости.

43. Структура фреймворка JUnit.

44. JUnit аннотации @Test, @DisplayName.

45. JUnit аннотации @BeforeEach, @AfterEach.

46. Тестовые классы и методы JUnit.

47. Утверждения JUnit. Класс Assert.

48. Тестирование исключений JUnit.

49. Генератор документирования Javadoc. Виды комментариев.

Javadoc - это инструмент, встроенный в JDK (Java Development Kit), который позволяет автоматически генерировать документацию на основе комментариев в исходном коде Java. Он обрабатывает специально отформатированные комментарии, расположенные перед объявлениями классов, методов, полей и других элементов кода.

Комментарии в формате Javadoc начинаются с двойного знака /** и заканчиваются символами */. Они могут содержать общее описание, описание параметров и возвращаемых значений, информацию о выбрасываемых исключениях, ссылки на связанные элементы и другую полезную информацию для документирования кода.

При использовании комментариев в формате Javadoc и запуске утилиты Javadoc из JDK, будет сгенерирована подробная документация, представляющая классы, методы, поля и другие элементы кода в виде HTML-страниц. Это позволяет разработчикам легко создавать и поддерживать понятную и полезную документацию к их Java-проектам.

Виды комментариев:

- Комментарии для класса:

```
/**
 * Краткое описание класса.
 *
 * Более подробное описание класса.
 */
new *
public class MyClass {
    // Код класса
}
```

- Комментарии для метода:

```java
/**
 * Краткое описание метода.
 *
 * Более подробное описание метода.
 *
 * @param param1 Описание первого параметра.
 * @param param2 Описание второго параметра.
 * @return Описание возвращаемого значения.
 * @throws InterruptedException Описание исключения.
 */
new *
public int myMethod(int param1, String param2) throws InterruptedException {
    // Тело метода
}
```

- Комментарии для поля:

```java
/**
 * Краткое описание поля.
 *
 * Более подробное описание поля.
 */
private int myField;
```

50. Дескрипторы Javadoc.

Дескрипторы Javadoc - это специальные теги, используемые в комментариях Java для документирования кода. Они помогают создавать документацию, автоматически генерируемую инструментами Javadoc.

@param - Описывает параметр метода. Используется для указания имени параметра и его описания.

@return - Описывает возвращаемое значение метода. Используется для указания типа и описания значения, которое возвращает метод.

@throws или **@exception** - Описывает исключение, которое может быть выброшено методом. Используется для указания типа и описания возможного исключения.

@see - Связывает элемент документации с другими классами, методами или полями. Используется для создания ссылок на связанные элементы.

@since - Указывает, с какой версии программного обеспечения был введен элемент. Используется для указания начальной версии, с которой элемент стал доступным.

@deprecated - Помечает элемент как устаревший. Используется для указания, что элемент больше не рекомендуется использовать и может быть удален в будущих версиях.

@author – Указание автора.

@version – Указание версии.

@value – Используется для документирования значений констант и полей.