**1. Метод и этапы структурного анализа. Программные системы, их ЖЦ. Анализ целевых и системных требований. Разработка требования к программным системам**.

Структурный анализ - метод исследования системы, который начинается с общего обзора ее и затем детализируется, приобретая иерархическую структуру со все большим числом уровней.

Он включает следующие этапы:

- декомпозиция системы на подсистемы и элементы, формирование структур и их описание;

- определение качественных и количественных характеристик выделенных структур;

- формирование критериев и оценка эффективности выделенных структур;

- принятие решения о необходимости совершенствования структурных характеристик системы.

ПС - система, состоящая из ПО и, компьютерного оборудования для его выполнения.

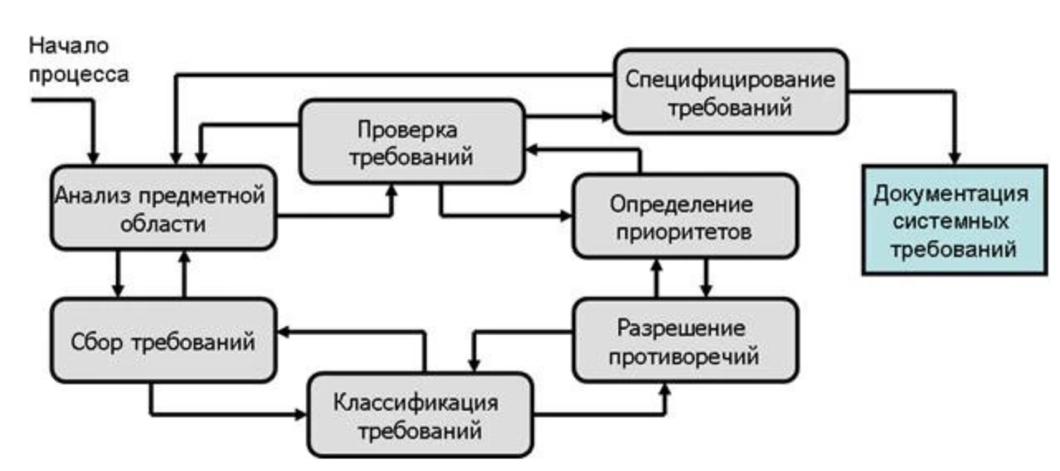
ЖЦ ПС - период времени, который начинается с момента принятия решения о необходимости создания ПС и заканчивается в момент ее полного изъятия из эксплуатации.

5 основных процессов ЖЦ: приобретение, поставка, разработка, эксплуатация и сопровождение.

8 вспомогательных процессов обеспечивают выполнение основных процессов - документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместная оценка, аудит, разрешение проблем. 4 организационных процесса: управление, создание инфраструктуры, усовершенствование и обучение.

На этапе анализа целевых требований определяются общие характеристики системы, которым она должна удовлетворять, устанавливается, что и как должна делать система.

Анализ системных требований - описание, как должны удовлетворяться запросы пользователя, описываются действия предполагаемой системы, хранимые данные, используемый интерфейс.

Требования к ПО - совокупность утверждений относительно атрибутов, свойств или качеств программной системы, подлежащей реализации. Они могут выражаться в виде текстовых утверждений и графических моделей.

**2. Метод и стадии объектно-ориентированного анализа и определение основных абстракций и механизмов**.

Объектно-ориентированный анализ – процедура определения требований к разработке ПО, разработка спецификаций программного обеспечения в терминах объектной модели системы программного обеспечения, которая состоит из взаимодействующих объектов. Он исследует требования с точки зрения классов и объектов, найденных в словаре проблемной области.

Он включает такие этапы:

- идентификация объектов  
- организация объектов путем создания диаграммы объектной модели  
- определение внутренних объектов или атрибутов объекта

- определение поведения объектов, т.е. действий объекта

- описание того, как объекты взаимодействуют

**3. Основы проектирования программного обеспечения, принципы аспекты проектирования. Разработка проектных решений и проектных процедур.**

Проектирование — итерационный процесс, при помощи которого требования к ПС транслируются в инженерные представления ПС. Модели проектирования предоставляют 3 типа информации. Информационная модель описывает информацию, которую, по мнению заказчика, должна обрабатывать ПС. Функциональная модель определяет перечень функций обработки. Поведенческая модель фиксирует желаемую динамику системы (режимы ее работы). На выходе этапа проектирования — разработка данных, разработка архитектуры и процедурная разработка ПС. В проектировании выделяют две ступени: предварительное проектирование и детальное проектирование. Предварительное проектирование формирует абстракции архитектурного уровня, детальное проектирование уточняет эти абстракции, добавляет подробности алгоритмического уровня. Во многих случаях выделяют интерфейсное проектирование, цель которого — сформировать графический интерфейс пользователя (GUI).

Принципы:

*S - Single Responsibility (единственная ответственность),*

каждый класс работает лишь над одной целью. Этот класс несет ответственность только в рамках достижения этой цели и изменяется лишь по одной причине. Общий принцип можно сформулировать так: для каждого класса важно определить единственное назначение, а все ресурсы, которые нужны для имплементации, следует инкапсулировать в данный класс и подчинить единственной задаче.

*O - Open Closed (открытость/закрытость),*

Класс должен быть открыт для расширения, однако закрыт для изменений. То есть мы можем добавить в класс новую функциональность, однако не можем редактировать существующие функции так, чтобы они противоречили используемому коду. Таким образом, программные сущности должны быть открыты для расширения, однако закрыты для модификации.

*L - Liskov substitution (принцип подстановки Барбары Лисков),*

Объекты в разрабатываемой программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.

*I - Interface Segregation (принцип разделения интерфейсов),*

По принципу разделения интерфейсов класс должен обладать способностью реализовывать множество интерфейсов. То есть код надо писать так, чтобы классу не надо было реализовывать функцию, которая не является важной для задач класса. Вывод тут прост: следует  разделять свои интерфейсы на категории. Или же: много интерфейсов, которые специально предназначены для клиентов, -- это лучше, чем 1 интерфейс общего назначения.

*D - Dependency Inversion Principles  (принцип инверсии зависимостей).*

Зависимости строятся на абстракциях, а точнее, отсутствие зависимости на что-либо конкретное.

Проектное решение - промежуточное или конечное описание объекта проектирования, необходимое и достаточное для рассмотрения и определения дальнейшего направления или окончания проектирования.

Этапы проектирования состоят из отдельных проектных процедур, которые заканчиваются частным проектным решением. Наиболее известными проектными процедурами являются: анализ, синтез и оптимизация. Анализ позволяет оценить степень удовлетворения проектного решения заданным требованиям и его пригодность. Процедура синтеза заключается в создании проектного решения (описания) по заданным требованиям, свойствам и ограничениям. Типичной проектной процедурой является оптимизация, которая приводит к оптимальному (по определенному критерию) проектному решению.

**4. Этапы проектирования программного обеспечения. Модульное программирование. Нисходящее и восходящее проектирование программ.**

При проектировании сложных систем выделяется следующие этапы проектирования:

- Стадия предварительного проектирования связана с поиском принципиальных возможностей построения систем, исследованием новых принципов, структур, технических средств, обоснованием наиболее общих решений; результат - техническое предложение.

- Стадия эскизного проектирования или опытно-конструкторских работ – производится детальная проработка всех частей проекта, конкретизируются и детализируются технические решения.

- Стадия рабочего проекта - формируется вся необходимая документация для изготовления изделия.

Модульное программирование — это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определенным правилам. Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок.

Способы проектирования многомодульных программ - восходящее и нисходящее проектирование.

Восходящее проектирование. При использовании восходящего подхода сначала проектируют и реализуют модули нижнего уровня, затем предыдущего и т. д. Каждый модуль при восходящем проектировании автономно программируется, тестируется и отлаживается. После этого отдельные модули объединяются в подсистемы с помощью управляющего модуля, в котором определяется последовательность вызовов модулей, ввод-вывод и контроль данных и результатов. В свою очередь, подсистемы затем объединяются в более сложные системы и в общий программный комплекс, который подвергается комплексной отладке с проверкой правильности межмодульных связей.

При нисходящем проектировании появляется возможность использовать вертикальное управление в схеме иерархии с использованием таких правил: модуль возвращает управление вызвавшему; модуль вызывает только модули более низкого уровня; принятие основных решений возлагается на модули максимально высокого уровня.

**5. Принципы объектно-ориентированного представления программных систем. Классы, объекты, общая характеристика и отношения между классами и объектами. Объектно-ориентированные методы анализа и проектирования.**

Принципы объектно-ориентированного представления программных систем:

- Абстрагирование - каждая абстракция фиксирует основные характеристики объекта, которые отличают его от других видов объектов и обеспечивают ясные границы.

- Инкапсуляция - процесс разделения элементов абстракции на секции с различной видимостью. Обычно скрывается структура данных объектов и реализация их методов.

- Модульность - модули служат физическими контейнерами, в которых объявляются классы и объекты логической разработки. Изменение реализации модулей должно проводиться без знания реализации других модулей и без влияния на их поведение.

- Иерархическая организация - упрощаются понимание проблем заказчика и их реализация — сложная система становится обозримой человеком. Она задает размещение абстракций на различных уровнях описания системы.

Объект — это конкретное представление абстракции. Объект обладает индивидуальностью, состоянием и поведением. Структура и поведение подобных объектов определены в их общем классе. Класс — описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и смысл. Любой объект — просто экземпляр класса.

Три основных отношения между классами:

Наследование – класс есть подвид другого класса.

- Ассоциация - объекты двух классов могут ссылаться один на другой, иметь связь между друг другом.

- Агрегация — отношение когда один объект является частью другого (Студент входит в Класс).

- Композиция — еще более «жесткое отношение, когда объект не только является частью другого объекта, но и вообще не может принадлежат еще кому-то (Машина и Двигатель).

**6. Языки визуального моделирования. Язык UML. Отношения, механизмы и диаграммы. Приемы моделирования.**

Языки визуального моделирования - формализованные наборы графических символов и правила построения из них визуальных моделей. Сейчас известны и активно используются на практике такие языки визуального моделирования, как UML и BPMN.

UML (Unified Modeling Language), — унифицированный язык моделирования. Это графический язык, который с помощью диаграмм и схем описывает разнообразные процессы и структуры.

Механизмы:

- спецификации (Specifications) - задний план, который полностью включает в себя составные части всех моделей системы, согласованные между собой;

- дополнения (Adornments) - каждый элемент нотации UML содержит базовый для него символ, к которому можно добавлять разнообразные специфичные для него дополнения;

- принятые деления (Common divisions) - описывают конкретные способы представления объектов реального мира в модели;

- механизмы расширения (Extensibility mechanisms) - применяются в случае, когда базовые возможности UML не удовлетворяют выдвигаемым требованиям.

Отношения:

Зависимость – семантическое отношение между сущностями, при котором изменение одной из них, независимой, может повлиять на семантику другой(пунктирная стрелка, данные -обработчик данных).

Ассоциация – структурное отношение, описывающее совокупность связей, где под связью понимается некоторая смысловая связь между объектами(сплошная линия, машина-имеет-колесо).

Обобщение (наследование) – отношение, при котором объект специализированного элемента (потомок) может быть подставлен вместо объекта обобщенного элемента (родителя).

Реализация – семантическое отношение между классификаторами, при котором один классификатор определяет обязательство, а другой гарантирует его выполнение. Отношение реализации встречаются между интерфейсами и реализующими их классами или компонентами.

Виды диаграмм UML:

- Структурные (показывают различные уровни абстракции и реализации)

1. Диаграмма классов

2. Диаграмма объектов (показывает системные объекты и их взаимосвязи)

3. Диаграмма компонентов (показывает логические группы элементов и их взаимосвязи)

4. Диаграмма развертывания (показаны аппаратные и программные компоненты и их взаимосвязи)

5. Диаграмма пакетов (зависимости между пакетами, которые составляют модель)

- Поведенческие (показывают функциональные возможности системы и демонстрируют, что должно происходить в моделируемой системе)

1. Диаграмма деятельности (пошаговый процесс с четким началом и концом)

2. Диаграмма вариантов использования (представляет функциональные требования системы)

3. Временная диаграмма (сколько времени занимают события и какие изменения происходят в зависимости от ограничений продолжительности)

4. Диаграмма последовательности (показывая последовательность сообщений и взаимодействий между операторами и объектами в хронологическом порядке)

Моделируя классы на UML, помните, что каждый класс должен отображать некоторую осязаемую или концептуальную абстракцию из предметной области конечного пользователя или разработчика.

Изображая класс в UML, вы должны:

- показать только те его свойства, которые важны для понимания абстракции в данном контексте;

- организовать длинные списки атрибутов и операций, группируя их по категориям;

- представить взаимосвязанные классы на одной диаграмме.

**7. Моделирование основных архитектурных решений с применением UML, моделирование поведения программных систем. Разработка поведенческих моделей.**

Для визуализации, специфицирования, конструирования и документирования программных систем необходимо рассматривать их с различных точек зрения. Архитектура - это совокупность существенных решений касательно организации программной системы; выбора структурных элементов, составляющих систему, и их интерфейсов; поведения этих элементов, специфицированного в кооперациях с другими элементами; составления из этих структурных и поведенческих элементов все более и более крупных подсистем; стиля, направляющего и определяющего всю организацию системы: статические и динамические элементы, их интерфейсы, кооперации и способ их объединения.

Архитектура может быть описана с помощью пяти взаимосвязанных видов:

- Вид с точки зрения прецедентов (диаграммы вариантов использования, use case diagrams)

- Вид с точки зрения проектирования - Вид с точки зрения процессов,

- Вид с точки зрения реализации - Вид с точки зрения развертывания

Модель поведения ‒ описание алгоритма работы системы.

Существует два основных типа поведенческих сущностей:

- взаимодействие - это поведение, суть которого заключается в обмене сообщениями между объектами в рамках конкретного контекста для достижения определенной цели;

- автомат - алгоритм поведения, определяющий последовательность состояний, через которые объект или взаимодействие проходят в ответ на различные события.

Требования, предъявляемые к модели поведения:

- Модель должна быть достаточно детальной для того, чтобы послужить основой для составления компьютерной программы — компьютер не сможет самостоятельно "додумать" опущенные детали.

- Модель должна быть компактной и обозримой, чтобы служить средством общения между людьми в процессе разработки системы и для обмена идеями.

- Модель не должна зависеть от особенностей реализации конкретных компьютеров, средств программирования и технологий, чтобы не сужать область применения языка UML.

- Средства моделирования поведения в UML должны быть знакомыми и привычными для большинства пользователей языка и не должны противоречить требованиям наиболее ходовых парадигм программирования.

**8. Шаблоны проектирования. Виды шаблонов и их классификация, распределение обязанностей между шаблонами.**

Паттерны проектирования — это решения распространенных проблем при разработке кода. В отличие от готовых функций или библиотек, паттерн представляет собой не конкретный код, а общую концепцию решения проблемы, которую еще нужно подстроить под задачи.

1) Порождающие предназначены для создания экземпляра объекта или группы связанных объектов.

Abstract Factory (предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, без непосредственной спецификации их конкретных классов)

Builder - метод создания составного объекта

Factory – логика создания объектов выносится в отдельный класс

Singleton - наличие единственного экземпляра класса

2) Структурные связаны с композицией объектов, с тем, как сущности могут использовать друг друга.

Adapter - применение функций объекта который нельзя модифицировать, специальным интерфейсом.

Bridge - разделяет абстракцию и реализацию таким образом, чтобы они могли меняться независимо

Composite (Компоновщик) - объединять объекты в древовидную структуру, даёт клиентам возможность обращаться к отдельным объектам и к группам объектов одинаковым образом

Decorator - динамическое подключение дополнительного поведения к объекту

Facade - все возможные внешние вызовы сводятся к одному и тому же объекту, который передаёт эти вызовы соответствующим объектам системы

Proxy - объект, контролирующий доступ к другому объекту, перехватывая при этом все вызовы

3) Поведенческие связаны с распределением обязанностей между объектами. Их отличие от структурных шаблонов заключается в том, что они описывают не только структуру, но и способы общения между ними. К ним относятся:

Chain of responsibility (Цепочка обязанностей) - организация уровней ответственности в системе

Iterator - доступ к элементам коллекции, а также навигацию по ним

Mediator — Посредник - позволяет обеспечить взаимодействие нескольких объектов

Memento - закрепление и сохранение внутреннего состояния объекта без нарушения инкапсуляции

Observer - особый механизм у класса, который даёт возможность экземпляру объекта этого класса получать уведомления от остальных объектов.

State - когда в процессе выполнения программы объект должен изменять свое поведение, в зависимости от того, в каком состоянии он находится

Template method - даёт наследникам возможность переопределять те или иные шаги алгоритма, оставляя его общую структуру нетронутой

Паттерны GRASP также называют паттернами распределения обязанностей, которые позволяют структурировать и именовать базовые принципы проектирования. GRASP состоит из 9 паттернов, описывающих базовые принципы объектно-ориентированного проектирования:

1. Создатель - какие условия должны соблюстись, что бы объекты верно порождали друг друга

2. Информационный эксперт – предоставление информации об объекте.

3. Слабая связность (Low Coupling) - объекты в системе знают друг о друге как можно меньше.

4. Контроллер - к кому именно должны обращаться вызовы из V (View), и кому C должен делегировать запросы на выполнение.

5. Высокое зацепление (High Cohesion) - класс должен стараться выполнять как можно меньше не специфичных для него задач, и иметь вполне определенную область применения.

6. Полиморфизм - позволяет обрабатывать альтернативные варианты поведения на основе типа.

7. Чистая выдумка (Pure Fabrication) - Создать новый искусственный класс с высоким зацеплением и слабым связыванием не являющийся частью предметной области.

8. Посредник (Indirection) - назначение обязанностей по взаимодействию классов доп.объекту.

9. Устойчивость к изменениям (Protected Variations) - устранении точек неустойчивости, которые наиболее часто будут подвержены изменению/модификации.

**9. Методы применения шаблонов проектирования. Разработка программной архитектуры и кодирование приложений на основе типовых решений.**

Систематизация приемов программирования и принципов организации классов получила название шаблона. При решении задач с применением шаблонов следует придерживаться следующих рекомендаций:

—шаблоны приносят пользу вне зависимости от того, как часто они используются при программировании;

— не следует торопиться с применением шаблонов при решении новой задачи;

— шаблон в чистом виде, как правило, неприменим, применимы только вариации;

— при применении шаблона следует начинать с его простейшей реализации, а уж затем вносить изменения по адаптации к конкретной ситуации;

— если после применения шаблона код стал хуже, то шаблон лучше убрать.

Архитектура ПО - совокупность важнейших решений об организации программной системы.

Примеры архитектурных шаблонов:

· Многоуровневый шаблон (Layered pattern). Система разбивается на уровни, изображаются один над другим. Каждый уровень может вызывать только уровень на 1 ниже него. разработку каждого уровня можно вести независимо. Недостатки - усложнение системы и снижение производительности.

· Шаблон посредника (Broker pattern). В системе много модулей, сложное взаимодействие друг с другом. Вводится посредник, по которому модули общаются. Повышается совместимость модулей системы. Недостатки - наличие посредника: он понижает производительность, его недоступность может сделать недоступной всю систему, он может стать объектом атак и узким местом системы.

· Шаблон «Модель-Вид-Контроллер» (Model-View-Controller pattern). Интерфейс меняется, сохраняя корректное взаимодействие с данными (чтение, сохранение).

· Клиент-серверный шаблон (Client-Server pattern). Подход повышает масштабируемость и доступность системы. При его недоступности становится недоступна вся система.

**10. Качество и эффективность ПО. Генерация кода на основе моделей. Конструирование высококачественного кода. Тестирование, отладка, рефакторинг и оптимизация кода**

Качество ПО - это совокупность характеристик ПО, относящихся к его способности удовлетворять установленные и предполагаемые потребности.

Характеристики качества ПО

- Функциональность определяется способностью ПО решать задачи, которые соответствуют зафиксированным и предполагаемым потребностям пользователя, при заданных условиях использования ПО. Т.е. эта характеристика отвечает за то, что ПО работает исправно и точно, функционально совместимо, соответствует стандартам отрасли и защищено от несанкционированного доступа.

- Надежность – способность ПО выполнять требуемые задачи в обозначенных условиях на протяжении заданного промежутка времени или указанное количество операций. Атрибуты данной характеристики – это завершенность и целостность всей системы, способность самостоятельно и корректно восстанавливаться после сбоев в работе, отказоустойчивость.

- Удобство использования – возможность легкого понимания, изучения, использования и привлекательности ПО для пользователя.

- Эффективность – способность ПО обеспечивать требуемый уровень производительности в соответствие с выделенными ресурсами, временем и другими обозначенными условиями.

- Удобство сопровождения – легкость, с которой ПО может анализироваться, тестироваться, изменяться для исправления дефектов, для реализации новых требований, для облегчения дальнейшего обслуживания и адаптироваться к имеющемуся окружению.

- Портативность характеризует ПО с точки зрения легкости его переноса из одного окружения в другое.

Можно генерировать код на основе UML-моделей с помощью специальных приложений.

Конструирование ПО - создание работающего ПО с привлечением методов верификации, кодирования и тестирования компонентов.

Основные техники обеспечения качества, используемые в процессе конструирования, включают: модульное и интеграционное тестирование, разработка с первичностью тестов (тесты пишутся до конструирования кода), пошаговое кодирование (деятельность по конструированию кода разбивается на мелкие шаги, только после тестирования результатов, которых производится переход к следующему шагу кодирования; известен также как итеративное кодирование с тестированием), использование процедур утверждений, отладка, технические обзоры и оценки.

Выбор и использование конкретных техник диктуется стандартами, используемыми проектной командой, а также зависят от опыта специалистов, занимающихся конструированием кода

При конструировании используются две формы тестирования: модульное и интеграционное.

Главная цель тестирования в конструировании уменьшить временной разрыв между моментом проявления ошибок, имеющихся в коде, и моментом их обнаружения. Во многих случаях, тестирование в конструировании производится после того, как код написан. В ряде случаев, тесты пишутся до того, как создается код(TDD). Тесты бывают типов черного, белого и серого ящика.

Отладка, или debugging, — это поиск, анализ и устранение ошибок (компиляции, компоновки, выполнения, логические) в ПО, которые были найдены во время тестирования.

Рефакторинг — это переработка исходного кода программы, чтобы он стал более простым и понятным. Он не меняет поведение программы, не исправляет ошибки и не добавляет новую функциональность, а делает код более понятным и удобочитаемым.

Оптимизация кода —методы преобразования кода ради улучшения его характеристик и повышения эффективности. Среди целей оптимизации можно указать уменьшение объема кода, объема используемой программой оперативной памяти, ускорение работы программы, уменьшение количества операций ввода-вывода.

**11. Внедрение, сборка и поставка проекта. Развёртывание, технологии и средства развертывания, наладки и обслуживания проектов.**

Внедрение ПО — процесс настройки программного обеспечения под определенные условия использования, а также обучения пользователей работе с программным продуктом.

Внедрение проекта включает в себя три этапа:

- подготовка объекта к внедрению проекта;

- опытное внедрение проекта (проверка правильности работы некоторых частей проекта);

- сдача проекта в промышленную эксплуатацию.

Внедрение может осуществляться с использованием следующих методов:

- последовательный метод, когда последовательно внедряется одна подсистема за другой;

- параллельный метод, при котором все задачи внедряются во всех подсистемах одновременно;

- смешанный подход, согласно которому проектировщики, внедрив несколько подсистем первым методом и накопив опыт, приступают к параллельному внедрению остальных.

Автоматизация процесса сборки программного продукта связана с разработкой различных скриптов для выполнения таких действий, как : компиляция исходного кода в бинарный; сборка бинарного кода; разворачивание программы на сервере (удаленном компьютере); оформление сопроводительной документации или описание изменений.

Apache Ant - это императивная командная система, созданная для кроссплатформенного применения.

Он был спроектирован таким образом, что часто применяемые при сборке команды ОС обернуты внутренними командами ant, а скрипты сборки описываются в формате XML. Широкое распространение получила декларативная система автоматизации сборки Apache Maven согласно описанию проектного pom.xml файла формата XML. В файлах проекта pom.xml содержатся не отдельные команды, а описание проекта. Все задачи по обработке файлов maven выполняет с использованием плагинов. Неоспоримым преимуществом maven является автоматическое управление зависимостями, хорошая структурированность проектов и отсутствие скриптов сборки как таковых, а следовательно проблем с ними.

Процесс поставки - процесс, в котором описан полный и интегрированный подход к выполнению проектов определенного типа. В нем содержится полное описание модели жизненного цикла в форме последовательно упорядоченной структуры наполнения методов.

Развёртывание ПО - все действия, которые делают программную систему готовой к использованию.

Jenkins - автономный сервер автоматизации с открытым исходным кодом. С ним стоит работать для автоматизации всех видов задач, связанных со сборкой, тестированием, поставкой или развертыванием программного обеспечения.

TeamCity - разработка от JetBrains. Он хорош простой настройкой и отличным интерфейсом. В дефолтной конфигурации есть большое кол-во функций, постоянно увеличивается число плагинов. Вся информация о результатах сборки хранится в БД.

**12. Фазы, итерации и циклы разработки программного кода. Рабочие процессы, модели и артефакты.**

Фазы разработки ПО

- Анализ требований. В рамках этой стадии происходит взаимодействие клиента и компании-разработчика, в ходе обсуждения деталей проекта помогающих более четко сформулировать требования. Результатом проведенного анализа становится формирование основного регламента, на который будет опираться исполнитель в своей работе — технического задания на разработку ПО. ТЗ должно полностью описывать поставленные перед разработчиком задачи и охарактеризовать конечную цель проекта в понимании заказчика.

- Проектирование. Моделирование теоретической основы будущего продукта. В рамках данного этапа стороны должны осуществить формирование окончательной архитектуры создаваемой системы; анализ необходимости использования готовых решений; проектирование основных элементов продукта — модели базы данных, процессов и кода; выбор среды программирования и инструментов разработки, утверждение интерфейса программы, включая элементы графического отображения данных; определение основных требований к безопасности разрабатываемого ПО.

- Кодирование. Непосредственная работа с кодом, опираясь на выбранный в процессе подготовки язык программирования. Кодирование может происходить параллельно со следующим этапом разработки — тестированием программного обеспечения, что помогает вносить изменения непосредственно по ходу написания кода.

- Тестирование и отладка. Процесс тестирования позволяет смоделировать ситуации, при которых программный продукт перестает функционировать. Отдел отладки затем локализует и исправляет обнаруженные ошибки кода, “вылизывая” его до практически идеального состояния

- Внедрение. Процедура внедрения программного обеспечения в эксплуатацию является завершающей стадией разработки и нередко происходит совместно с отладкой системы. Ключевой целью поэтапного внедрения разработанной программы становится постепенное выявление не обнаруженных ранее ошибок и недочетов кода. Именно на этой стадии выкристаллизовывается окончательная картина взаимодействия пользователя с программой.

Внутри каждой фазы происходит несколько итераций. Итерация представляет полный цикл разработки, от выработки требований во время анализа до реализации и тестирования. Это завершенный этап, в результате которого выпускается версия исполняемого продукта, реализующая часть запланированных функций. Затем эта версия от итерации к итерации наращивается до получения готовой системы. Конечным результатом является выпуск готового продукта. Прохождение через основные фазы - цикл разработки. Каждый цикл завершается генерацией версии системы.

Разработка ПО состоит из девяти рабочих процессов:

- бизнес-анализ – предполагает определение желаемых параметров системы и нужд пользователей;

- требования - формализация образа системы.

- анализ и моделирование - перевод требований в формализованную программную модель.

- реализация, кодирование - предполагает собственно написание кода.

- тестирование – поиск и устранение возможных ошибок

- внедрение - предполагает установку продукта заказчику, подготовку персонала, запуск системы.

- управление конфигурацией и изменениями - управление версиями продукта.

- управление проектом — предполагает набор административных действий управления проектом.

- окружение - создание и поддержка средств анализа, проектирования, разработки, тестирования.

Внутри каждого рабочего процесса сосредоточены связанные между собой артефакты и деятельности. Артефакты — это некоторые продукты проекта, порождаемые или используемые в нем при работе над окончательным продуктом. Модель - упрощение реального продукта; она создается для лучшего понимания разрабатываемой системы.

**13. Архитектура клиент-сервер. Многоуровневая структура веб-приложения. Требования к современному веб-приложению.**

Клиент-серверная архитектура описывает распределенные системы, состоящие из отдельных клиента и сервера и соединяющей их сети. Простейшая форма, называемая 2-уровневой архитектурой – это серверное приложение, к которому напрямую обращаются множество клиентов. Архитектурный стиль клиент-сервер описывает отношения между клиентом и одним или более серверами, где клиент инициирует один или более запросов, ожидает ответы и обрабатывает их при получении. Для связи с клиентом сервер может использовать широкий диапазон протоколов и форматов данных.

Дальнейшее развитие архитектуры Web-приложений и технологии «клиент-сервер» привело к появлению многоуровневой архитектуры, в которой дополнительно вводится сервер приложений. Он является промежуточным уровнем, который обеспечивает организацию взаимодействия клиентов («тонких» клиентов) и сервера БД. В этой модели промежуточные серверы получают запросы клиентов и обрабатывают их, координируя свои действия с подчиненными серверами, применяя бизнес-логику. Связью между клиентом и БД управляет промежуточный прикладной уровень, что позволяет клиентам получать доступ к данным из различных решений СУБД. Трехуровневая архитектура более безопасна, поскольку клиент не имеет прямого доступа к данным.

Основные достоинства многоуровневой архитектуры Web-приложений заключаются в следующем:

1) разгрузка Web-сервера от выполнения части операций, перенесенных на сервер приложений и уменьшение размера модулей расширения сервера на основе разгрузки их от лишнего кода;

2) обеспечение межплатформенного управления между Web-сервером и сервером базы данных;

3) упрощение администрирования и настройки параметров сети - при внесении изменений в ПО или конфигурацию сервера БД не нужно вносить изменения в Web-сервер.

Под требованиями к ПО будем понимать совокупность утверждений относительно свойств программной системы, подлежащая реализации при создании ПО.



**14. Структура и размещение веб-приложения. Этапы разработки и развертывания веб-приложения. Роль веб-сервера в развертывании веб-приложения на Java**

Структура веб-приложения.

Логика распределена между сервером и клиентом, хранение и обработка данных осуществляется, преимущественно, на сервере, её представление в удобном для пользователя виде - в браузере. Обмен информацией происходит по сети. Веб-приложения состоят как минимум из трёх основных компонентов. Серверная часть веб-приложения - программа или скрипт на сервере, обрабатывающая запросы пользователя (точнее, запросы браузера). Система управления базами данных, (СУБД) - программное обеспечение на сервере, занимающееся хранением данных и их выдачей в нужный момент. Клиентская часть веб-приложения – это отображаемый в браузере графический интерфейс.

Размещение web-приложения

Для того, чтобы веб-приложение стало доступно, его необходимо разместить в рамках веб-сервера Tomcat, Glassfish, JBoss, Sun и др. После этого приложение получит свой уникальный адрес в рамках протокола HTTP. Используя этот адрес, пользователь может обратиться к приложению.

Этапы разработки веб-приложения:

- определение целей и задач проекта;

- разработка структуры сайта;

- разработка дизайн-макетов;

- html-вёрстка;

- программирование и контроль качества;

запуск и сопровождение, SEO-оптимизация.

Роль web-сервера в развертывании приложения на java

Веб-сервер – ПО, обеспечивающее доставку контента конечному пользователю по сети. Он реализует серверную часть протокола HTTP. Сервисы – EJB, JMS, управление ресурсами, безопасность, JSF.

Функции и особенности веб-серверов: получение контента от пользователей и его передача; поддержка динамически генерируемых страниц; аутентификация и авторизация пользователей; ведение журнала обращений пользователей к ресурсам; поддержка HTTPS для защищённых соединений с клиентами.

Реализации веб-серверов: Apache Tomcat – контейнер сервлетов, и Glassfish

Apache Tomcat состоит из следующих компонентов:

1. Web Connector Coyote (поддерживает HTTP, для веб-серверов или контейнера приложений, прослушивает входящие соединения на определенном TCP-порту сервера).

2. Web Container Catalina (контейнер сервлетов Tomcat, даёт возможность динамически генерировать веб-контент)

3. Jasper Compiler (анализирует JSP-файлы, чтобы компилировать их в Java-код как сервлеты, которые могут быть обработаны с помощью Catalina)

**15. Структура архива war. Способы создания файла war. Описатель развертывания web.xml: его назначение и содержимое.**

WAR (Web application ARchive) файлы используются для распостранения Java web-приложений. WAR имеет такую же структуру, как и JAR-файл, единый сжатый файл, содержащий несколько файлов внутри него. WAR-файлы используются для объединения JSP-файлов, сервлетов, Java class-файлов, XML-файлов, javascript-библиотек, JAR-библиотек, статических web-страниц и любых других ресурсов, необходимых для работы приложения.

WAR-файлы обычно разворачивают в контейнерах сервлетов, но также возможно разворачивать и в Java EE серверах приложений. Когда WAR-файл разворачивается в контейнере, контейнер обычно распаковывает его для доступа к файлам, а затем запускает приложение. На основе контейнеров сервлетов получается наиболее производительная платформа для Java web-приложений, WAR-файлы являются не только стандартом Java спецификации, но WAR-файлы нельзя редактировать, пока работает приложение. Любые изменения потребуют пересборки файла.

Создать WAR-архив можно следующими способами:

- Паковщиком, входящим в состав J2EE SDK.

- Выполнив в Apache Ant задачу «war».

- Выполнив в Apache Maven команду «mvn clean install».

Файл web.xml хранит информацию о конфигурации приложения. Он не является обязательной частью приложения. Если приложение содержит только JSP-файлы, этот файл не строго обязателен. Данный файл должен располагаться в папке WEB-INF приложения. При запуске Tomcat считывает его содержимое и использует считанную конфигурацию. Если же файл содержит ошибки, то Tomcat отображает ошибку. web.xml имеет определенную структуру. Все вложенные узлы, которые определяют конфигурацию, помещаются в корневой узел <web-app>. У элемента web-app определяется ряд атрибутов. В данном случае атрибуты xmlns и xmlns:xsi указывают на используемые пространства имен xml. Атрибут version указывает на версию спецификации сервлетов или Servlet API, которая используется в приложении. С помощью элемента <servlet-mapping> можно задать сопоставление сервлета с запрашиваемым URL. Прежде всего вначале с помощью элемента <servlet> определяется сервлет. Элемент <servlet-name> задает имя сервлета, на которое будет проецироватья класс, указанный в элементе <servlet-class>. Затем в элементе <servlet-mapping> сервлет сопоставляется с путем.

**16. Порядок компоновки веб-приложения. Технологии сборки веб-приложения. Файл сборки build.xml и его содержимое. Инструменты для сборки веб-приложения.**

Реальное web-приложение обычно включает достаточно большое количество файлов, размещенных в разветвленной системе директорий файловой системы. Для выполнения компоновки используют специальные программные средства, позволяющие упростить этот процесс.

Apache Ant – средство автоматизации, созданное на базе Java и XML, как программное обеспечение с открытым исходным кодом, которое представляет собой инструмент, позволяющий автоматизировать процесс компоновки java-проекта. Сюда относится компиляция java-файлов, создание jar-архивов и специальных файлов для размещения приложения на сервере (war-файлы). Управление процессом сборки происходит посредством XML-сценария, также называемого build-файлом. Этот файл содержит определение проекта, состоящего из отдельных целей (targets).

Типичными примерами целей являются clean (удаление промежуточных файлов), compile (компиляция всех классов), deploy (развёртывание приложения на сервере). Конкретный набор целей и их взаимосвязи зависят от специфики проекта.

Ant позволяет определять собственные типы заданий путём создания Java-классов, реализующих определённые интерфейсы. Ant'у необходима помощь в виде build файла. Build файл показывает Ant'у что надо делать, чтобы превратить то что есть (как правило, исходный код) в то что вы хотите. Сценарий представляет собой детальный план сборки из частей единого целого, включающий ряд операндов, позволяющих выполнять команды копирования, удаления и перемещения файлов, компиляции java-файлов, формирование документации к коду и исполняемого jar-файла.

Корневой элемент сценария **project** может содержать три необязательных атрибута :  
- name - имя проекта;  
- default - цель проекта по умолчанию;  
- basedir - базовая директория, относительной которой будут вычисляться все пути.

Элемент описывающий цель проекта **target** может содержать следующие атрибуты :  
- name - имя цели, обязательный атрибут;  
- depends - промежуточные цели, от которых зависит данная цель; имена перечисляются через запятую;  
- if - определяет какие свойства должны быть равны true для запуска цели;  
- unless - определяет какие свойства должны быть равны false для запуска цели;  
- description - краткое описание цели, что она делает.

Параметр **property** определяет пару имя/значение, которая может многократно использоваться в сценарии подобно переменным. Свойства (настройки) можно определять как внутри **build.xml** файла, так и в отдельных файлах. При определении внутри xml файла свойства могут включать следующие атрибуты :

* name - имя свойства;
* value - значение свойства;
* location - устанавливает значение свойства в абсолютный путь. Если значение уже абсолютный путь, то ничего не меняется, если относительный то подставляется базовая директория.
* refid - ссылка на другой объект, определенный где-либо;
* resource - имя ресурса содержащего настройки в формате настроечного файла;
* file - путь к файлу настройки (в нем свойства определяются как имя=значение на отдельной строке);
* url - адрес настройки;
* environment - префикс используемый для доступа к переменным окружения. Например, если определено myenv, то к переменным обращаются как "myenv.PATH";
* classpath - путь к ресурсам;
* prefix - префикс добавляемый к свойствам загруженных из файла, ресурса, или url. По умолчанию префикс ".".

**17. Понятие сервлета. ЖЦ сервлета: обработка событий, связанных с ЖЦ.**

Сервлет – модуль, расширяющий возможности серверов типа запрос-ответ, таких как Java-совместимые web-сервера.

Servlet API – коды, требующиеся для построения сервлетов, содержат два пакета: – javax.servlet, javax.servlet.http.

Жизненный цикл сервлета:

- сервер загружает и инициализирует объект

- сервер обрабатывает ноль или более запросов клиентов

- сервер выгружает сервлет

Для инициализации сервлета сервер запускает init-метод сервлета.

Когда сервер удаляет сервлет, он запускает метод destroy().

Интерфейсы и их назначение:

- Servlet – объявляет методы цикла жизни для сервлета

- ServletConfig – позволяет сервлетам получать параметры инициализации

- ServletContext – активирует возможности сервлетов для регистрации событий и доступа к информации об их среде

- ServletRequest – используется для чтения данных из запроса клиента

- ServletResponse – используется для записи данных в ответ клиенту

- SingleThreadModel – указывает, что сервлет защищен от многопоточности

Когда сервлет принимает вызов клиента, он получает два объекта:

- ServletRequest (инкапсулирует связь клиента с сервером)

- ServletResponse (инкапсулирует обратную связь сервера с клиентом)

Оба являются интерфейсами.

Пакет HTTP, интерфейсы:

- HttpServletRequest – используется для чтения данных из HTTP (запроса клиента)

- HttpSevletResponse – используется для записи данных в HTTP (ответ клиенту)

- HttpSession – позволяет читать и записывать данные сеанса

- HttpSessionContext – обеспечивает управляемость сеанса

Класс HttpSevlet обеспечивает методы для обработки запросов и ответов HTTP. Его методы:

- doGet – метод, который вызывается при получении HTTP-запроса методом GET. Этот метод принимает два параметра типа HttpServletRequest и HttpServletResponse, которые представляют запрос и ответ соответственно.

- doPost – метод, который вызывается при получении HTTP-запроса методом POST. Этот метод также принимает два параметра типа HttpServletRequest и HttpServletResponse.

- doPut – метод, который вызывается при получении HTTP-запроса методом PUT. Этот метод также принимает два параметра типа HttpServletRequest и HttpServletResponse.

- doDelete – метод, который вызывается при получении HTTP-запроса методом DELETE. Этот метод также принимает два параметра типа HttpServletRequest и HttpServletResponse.

**18. Контейнер сервлетов, его функции и назначение. Развёртывание веб-приложений на примере веб-сервера Apache. Инструменты Catalina, Coyote Jasper и их назначение.**

Контейнер сервлетов – программа, обеспечивающая взаимодействие между сервлетами и клиентами, обрабатывает запроса, полученные от клиентов и передает их соответствующим сервлетам.

Задачи и функции контейнера сервлетов:

- Обмен данными

- Управление жизненным циклом сервлета

- Создание программной среды

- Обработка исключений

- Идентификация-авторизация-аутентификация

- Организация сессии

Apache Tomcat – веб-сервлет. Его составляющие:

- Catalina (контейнер сервлетов Tomcat дает возможность динамические генерировать любой веб-контент).

- Coyote (поддерживает HTTP для веб-серверов или контейнера приложений, прослушивает входящие соединения на TCP порту сервера)

- Jasper – (механизм JSP, отвечает за компиляцию JSP-страниц в сервлеты, которые могут быть запущены в Tomcat-контейнере сервлетов. Jasper также отвечает за выполнение операций, таких как проверка синтаксиса JSP-страниц и преобразование JSP-страниц в сервлеты)

**19. Интерфейс ServletContext: назначение и использование. Сессия: доступ, связывание с объектами, управление. Методы getSession(), setAttribute(), getAttribute().**

ServletContext – активирует возможности сервлетов для регистрации событий и доступа к информации об их среде. ServletContext представляет собой общий объект, доступный всем сервлетам в данном приложении, и содержит информацию, такую как параметры инициализации, атрибуты контекста и другие ресурсы.

Сессия в веб-приложении представляет собой период времени, в течение которого клиент взаимодействует с приложением. Используя сессии, можно сохранять состояние между запросами, например, данные формы, идентификатор пользователя и другую информацию. Контейнер сервлетов обеспечивает механизмы управления сессиями.

Сессии реализуются с помощью объекта интерфейса HttpSession. Он содержит набор атрибутов, которые можно использовать для хранения данных. Контейнер сервлетов генерирует уникальный идентификатор сессии, который используется для связи между клиентом и сервером.

Session методы: putValue, getValue

Для отправки сервлета сервлету используются методы forward(), sendRedirect(). Forward() быстрее, чем sendRedirect().

Методы интерфейса ServletContext, связанные с сессиями:

- getSession() - возвращает текущий объект HttpSession, связанный с текущим запросом. Если такого объекта еще нет, метод создает новую сессию и возвращает ее.

- setAttribute() - используется для связывания объекта сессии с именем атрибута в контексте приложения. Например, можно использовать этот метод для сохранения данных формы в сессии:

- getAttribute() - используется для получения объекта сессии, связанного с именем атрибута в контексте приложения. Например, можно использовать для извлечения данных формы из сессии

**20. Серверные страницы JSP. ЖЦ JSP-страницы. Создание статического и динамического контента.**

JSP расшифровывается как Java Server Pages. Это технология, которая позволяет разработчикам веб-контента создавать содержимое с динамическими и статическими компонентами. Странички JSP включают в себя текст разного типа. Первый вариант – статическая исходная информация. Она может быть представлена в: HTML; WML; XML; SVG.Второй тип текста – JSP элементы. Они отвечают за создание динамического содержимого. Также допускается применение полноценной библиотеки JSP тегов и EL. Все это помогает внедрить Java-код в статичное содержимое JSP-страничек.  
JSP – это некая платформенно-независимая технология, используемая для написания HTML динамического характера. Теги здесь применяются для того, чтобы извлекать информацию из БД, обмениваться имеющимися сведениями и так далее. Для запуска часто применяются среды IDE NetBeans или Eclipse.

Каждый программист при написании HTML странички должен решить, каким функционалом выбранного языка пользоваться. JSP обладает рядом преимуществ перед сервлетами:

- Отвечает за возможность создания и управления HTML динамического типа. В данном случае процессы будут осуществляться более просто.

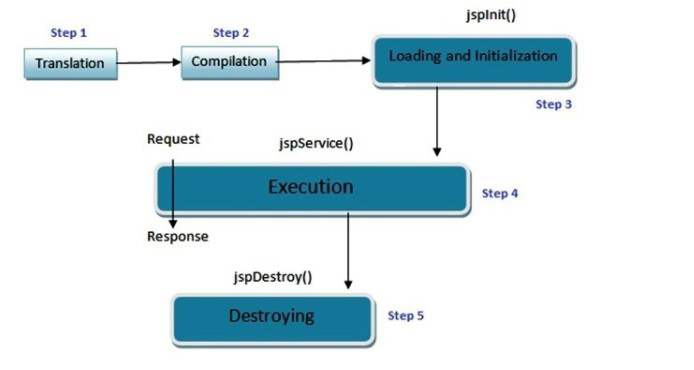
- Для работы не требуется загрузка дополнительного программного обеспечения или документов. Пример – можно обойтись без файла класса java и web.xml.

- Обработка ведется web-container для всех корректировок кода. Это приводит к тому, что JSP помогает избежать перекомпиляции.

- доступ здесь осуществляется напрямую. Сервлеты требуют предварительного проведения сопоставления с web.xml.

### Цикл жизни

Следующий момент, достойный внимания – это жизненный цикл JSP. Он полностью повторяет servlet, но имеет один дополнительный шаг. Во время него происходит компиляция рассматриваемой технологии в сервлет.



#### Перевод – самое начало

Это – первый этап. Здесь происходит следующее:

1. Контейнер переводит JSP документ в код Джавы. Он выступает сервлетом.
2. Перевод проводится в автоматическом режиме через веб-сервер.
3. Последний находит и проверяет достоверность, а затем производит запись сервлета для JPS page.

Здесь никаких проблем с пониманием быть не должно.

#### Компиляция – шаг два

После описанных ранее действий контейнер JSP будет компилировать исходный Java-код. Это необходимо для того, чтобы создавать следующие servlets и сгенерированный классовый файл.

#### Загружаем и инициализируем – шаг три

Третий этап предусматривает следующее:

1. JSP Container загружает ранее созданный servlet.
2. Если процесс прошел правильно, будет создан экземпляр класса сервлета.
3. Важно учесть, что здесь используется конструктор, не имеющий никаких аргументов.
4. Контейнер JSP проводит инициализацию экземпляра объекта через вызов метода init. Он реализуется container посредством обращения к jspInit().



Элемент кода выше наглядно демонстрирует, каким образом осуществляется инициализация экземпляра.

#### Выполнение – шаг четыре

Теперь, когда основные этапы позади, происходит не менее важный момент – выполнение. Здесь технология JSP вызовет метод \_jspService(). У него два параметра:

* httpServletRequest;
* HttpServletResponse.

Вызывается рассматриваемый метод всего один раз для каждого отдельного запроса.



Упомянутый метод будет отвечать за генерацию ответов на посланный запрос.

#### Разрушение – логическое завершение

На данном этапе целесообразно говорить о прекращении жизненного цикла.



При разрушении контейнер будет удалять JSP при помощи jspDestroy(). Как это происходит, указано на наглядном примере выше. Статический контент создаётся в виде HTML. Динамический контент создаётся в JSP элементах

**21. Директивы, объявления. Выражения и скриптлеты в JSP.**

Помимо HTML-тегов JSP страницы содержат теги следующих категорий:

- Директивы: обеспечивают глобальную информацию, касающуюся конкретных запросов, направляемых в JSP, представляют информацию, необходимую на стадии трансляции. Существует 3 типа директив: page (определяет свойства страницы JSP, которые воздействуют на транслятор), taglib (объявляет, что данная страница использует библиотеку тегов), include (позволяет вставлять текст или код в процессе трансляции страницы JSP в сервлет). Синтаксис директив:

<%@ директива имяАтрибута=”значение”%>

- Объявления: для определения переменных и методов на языке скриптов, которые в дальнейшем используются на странице JSP. Синтаксис <%! код Java !%>. Пример: <%! String s = “Hi”; !%>

- Скриплеты: включают разные фрагменты кода, написанного на языке скрипта, определённого в директиве language. Фрагменты кода должны соответствовать синтаксическим конструкциям языка скриплетов, т.е. как правило, синтаксису Java. Синтаксис скриплета: <%Текст скриплета%. Всё что написано вне скобок <%%> преобразуется в out.println(текст)

- Выражения: исполняемое выражение, написанное на языке скрипта, определённого в директиве language. Результат выражения JSP, имеющий обязательный тип String, направляется в поток вывода out с помощью текущего объекта JspWriter. Синтаксис <%= текст выражения%>. Обычно выражения используются для того, чтобы вычислить и вывести на экран представление переменных и методов, определённых в блоке объявлений страницы JSP или полученных от компонентов JavaBeans которые доступны из JSP.

- Комментарии: Выводимый комментарий: комментарий выводится в выходной поток и отображается в браузере. Синтаксис: <!-- Тескт комментария-->

Закомментированный блок: обычный комментарий. Синтаксис: <%-- Текст комментария--%>

**22. Извлечение полей и значений в JSP. Неявные объекты в JSP.**

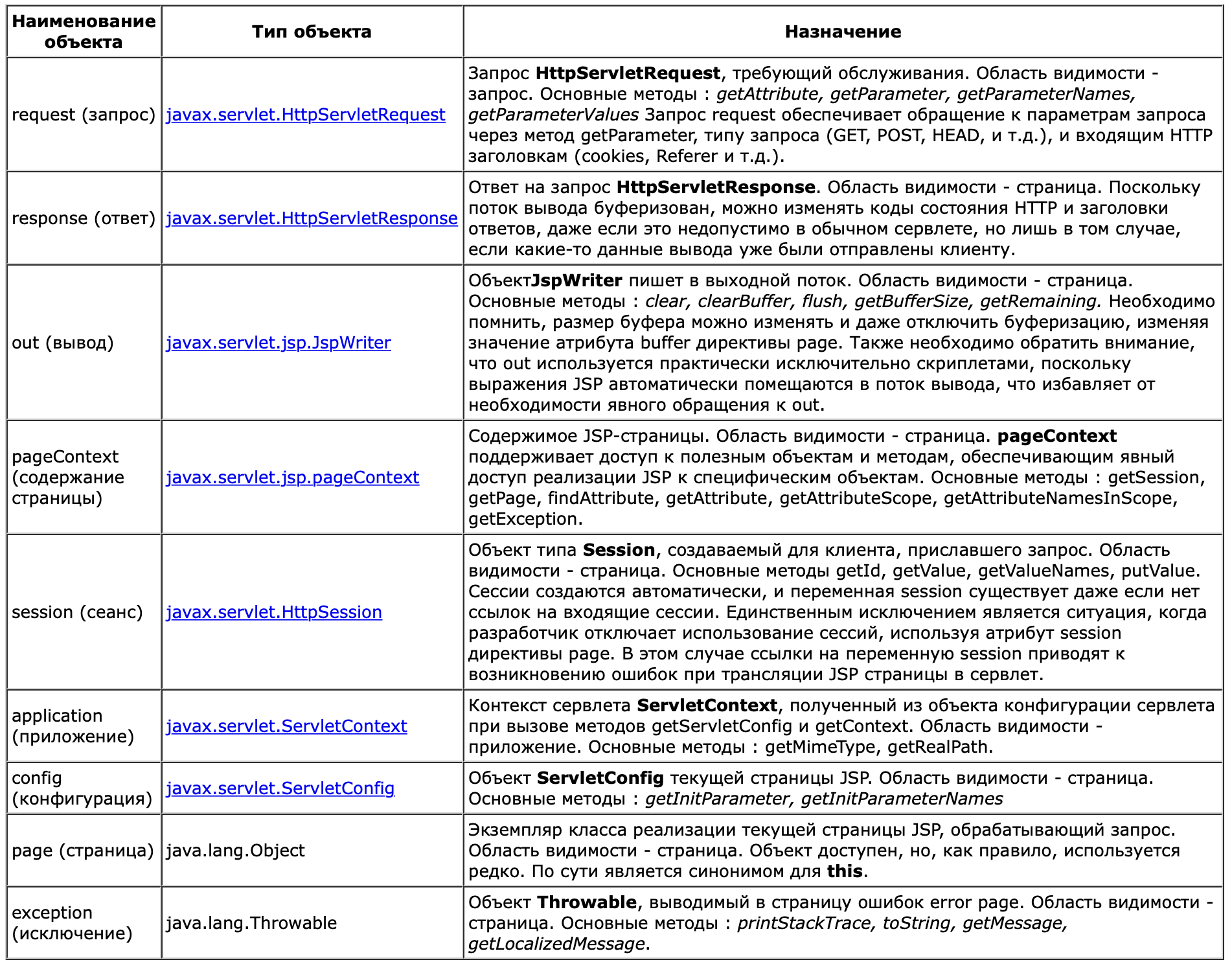
В JSP можно извлекать поля и значения из объектов Java, используя выражения JSP (JSP Expression Language, EL). EL позволяет обращаться к свойствам объектов Java, вызывать методы и выполнять арифметические операции. Например, если у вас есть объект Java с именем "person" и свойствами "name" и "age", вы можете извлечь эти значения в JSP следующим образом:

<p>Имя: ${person.name}</p>

<p>Возраст: ${person.age}</p>

Здесь ${person.name} и ${person.age} - это выражения EL, которые извлекают соответствующие значения из объекта "person".

Неявные объекты. Эти объекты являются объектами Java, которые JSP-контейнер делает доступными для разработчиков на каждой странице, и разработчик может вызывать их напрямую, без явного объявления или импорта. Неявные объекты JSP также называются предопределенными переменными .



**23. Стандартные элементы action в JSP. Передача управления другому веб-компоненту (сервлету, JSP, HTML)**

Действия JSP actions могут воздействовать на стандартный поток вывода, использовать, модифицировать и создавать объекты. actions используют конструкции с синтаксисом XML для управления работой движка сервлета и позволяют, таким образом, динамически подключать файл, многократно использовать компоненты JavaBeans, направлять пользователя на другую страницу или генерировать HTML для Java plugin.

Согласно спецификации JSP, синтаксис **actions** базируется на XML. В таблице представлены соответствия определений JSP и XML.

|  |  |
| --- | --- |
| **Конструкция JSP** | **Эквивалентная конструкция в XML** |
| <% page ... %> | <%jsp:directive.page ... /> |
| <% include ... %> | <%jsp:directive.include ... /> |
| <%! ... %> | <%jsp:declaration>      . . . <%/jsp:declaration> |
| <% ... %> | <%/jsp:scriptlet>      . . . <%/jsp:scriptlet> |
| <%= ... %> | <%/jsp:expression>      . . . <%/jsp:expression> |

Существует набор стандартных действий, которые должны быть в обязательном порядке реализованы любым контейнером JSP, удовлетворяющим спецификации. Кроме этого, возможно создание новых действий с помощью директивы библиотеки тегов *[taglib](https://java-online.ru/jsp-taglib.xhtml)*.

Список стандартных действий представлен в следующей таблице.

|  |  |
| --- | --- |
| **Тип действия** | **Назначение действия** |
| [<jsp:useBean>](https://java-online.ru/jsp-actions.xhtml#useBean) | Объявление объекта JavaBean, который будет использоваться на странице JSP |
| [<jsp:setProperty>](https://java-online.ru/jsp-actions.xhtml#setProperty) | Установление значения свойства объекта JavaBean |
| [<jsp:getProperty>](https://java-online.ru/jsp-actions.xhtml#getProperty) | Чтение значения свойства объекта JavaBean |
| [<jsp:include>](https://java-online.ru/jsp-actions.xhtml#include) | Включение в страницу JSP дополнительных статических и динамических ресурсов |
| [<jsp:forward>](https://java-online.ru/jsp-actions.xhtml#forward) | Перенаправление обработки на другой статический ресурс, например сервлет |
| [<jsp:plugin>](https://java-online.ru/jsp-actions.xhtml#plugin) | Подключение дополнительных программных модулей (компонент JavaBean или апплет) |
| [<jsp:param>](https://java-online.ru/jsp-actions.xhtml#param) | Определение значения параметра |

Действие forward (метод RequestDispatcher) позволяет во время выполнения страницы JSP перенаправлять текущий запрос на другую страницу JSP, некоторый статический ресурс или класс Java-сервлета, находящийся в том же контексте, что и текущая страница JSP.

Для того, чтобы выполнить перенаправление запроса, вначале с помощью метода getServletContext() получаем объект ServletContext, который представляет контекст запроса. Затем с помощью его метода getRequestDispatcher() получаем объект RequestDispatcher. Путь к ресурсу, на который надо выполнить перенаправление, передается в качестве параметра в getRequestDispatcher. Затем у объекта RequestDispatcher вызывается метод forward(), в который передаются объекты HttpServletRequest и HttpServletResponse.

Для переадресации применяется метод sendRedirect() объекта HttpServletResponse. В качестве параметра данный метод принимает адрес переадресации. Адрес может быть локальным, внутренним, а может быть и внешним. В метод sendRedirect передается адрес относительно корня текущего домена.

**24. Библиотека тегов JSTL. Её назначение, виды стандартных библиотек.**

JSTL обозначает стандартную библиотеку тегов страниц сервера Java и представляет собой набор пользовательских библиотек тегов JSP, которые предоставляют общие функции веб-разработки.

Преимущества JSTL

1. Стандартный тег: он обеспечивает богатый уровень переносимых функций страниц JSP. Разработчику легко понять код.

2. Чистый код: скрипты сбивают с толку разработчика, а использование JSTL делает код чистым.

3. Автоматическая поддержка JavabeansInterospection: JSTL имеет преимущество перед JSP-скриптлетами. Язык выражений JSTL очень легко обрабатывает код JavaBean. Нам не нужно понижать объекты, которые были получены как атрибуты области видимости. Использование кода сценариев JSP будет сложным, и JSTL упростил эту задачу.

4. Людям легче читать: JSTL основан на XML, который очень похож на HTML. Следовательно, разработчикам легко понять.

5. Компьютерам легче понять: такие инструменты, как Dreamweaver и первая страница, генерируют все больше и больше HTML-кода. Инструменты HTML отлично справляются с форматированием кода HTML. HTML-код смешивается с кодом скрипта. Поскольку JSTL выражается в виде XML-совместимых тегов, для генерации HTML легко выполнить анализ кода JSTL в документе.

Несмотря на то, что JSTL часто называется библиотекой, на самом деле она содержит ряд библиотек:

1) Core: содержит основные теги для наиболее распространенных задач.

Использует префикс "c" и uri <http://java.sun.com/jsp/jstl/core>

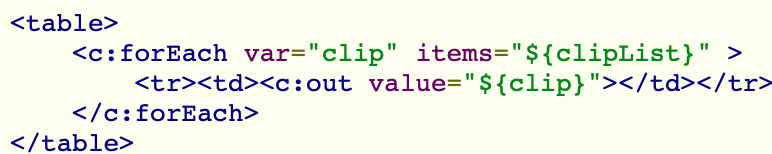
- <c:out>

- <c:set>(определение переменной) <c:set var="user" scope="session" value="Sergey" />

- <c:remove>(удаление переменной) <c:remove var="user" scope="request" />

- <c:if> <c:if test="${param.age > 24}"> Возраст более 24 лет </c:if>

- <c:forEach> позволяет задать цикл



2) Formatting: предоставляет теги для форматирования чисел, дат, времени.

Использует префикс "fmt" и uri "http://java.sun.com/jsp/jstl/fmt"

3) SQL: предоставляет теги для работы с sql-запросами и источниками данных.

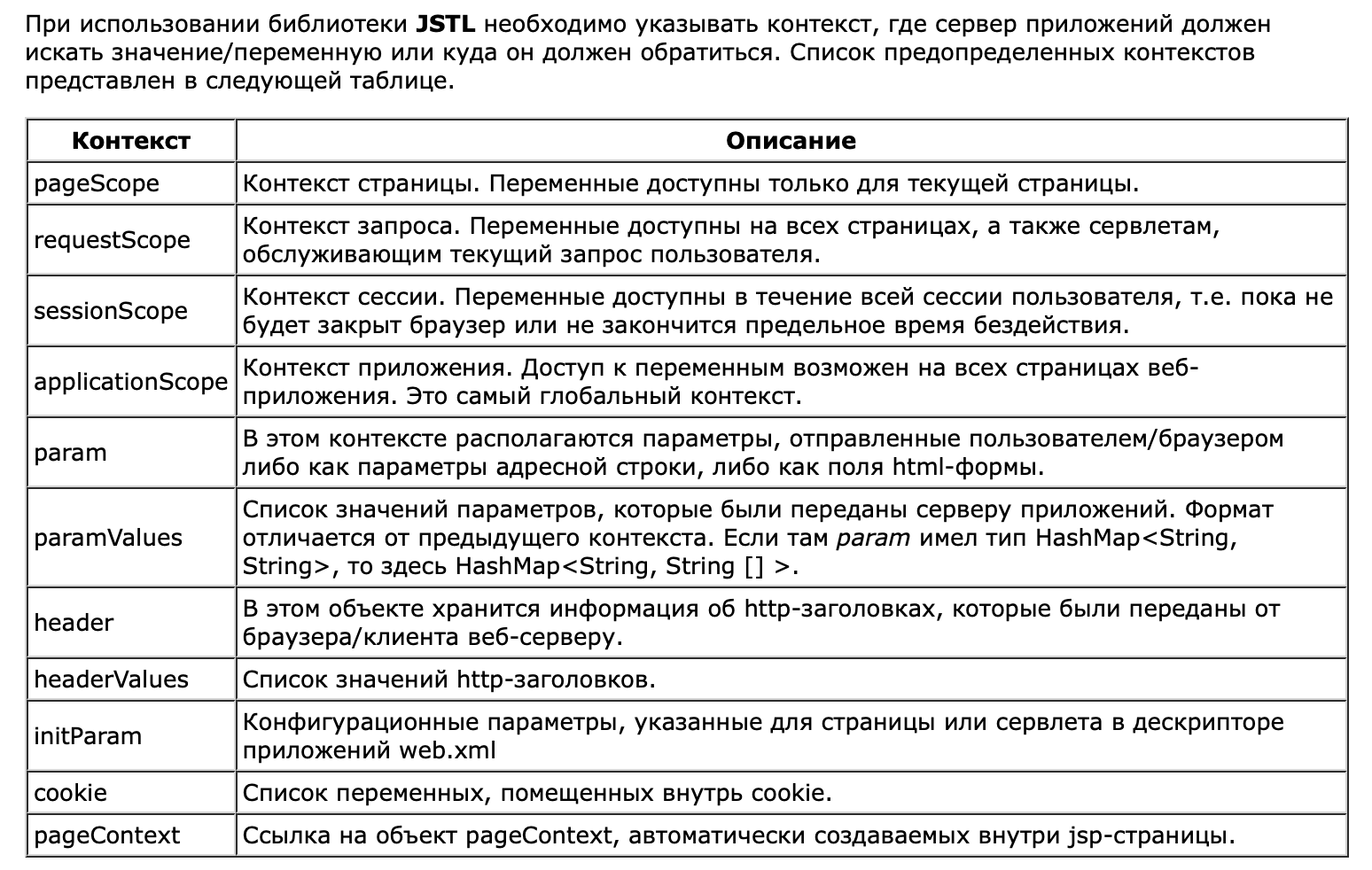
Использует префикс "sql" и uri "http://java.sun.com/jsp/jstl/sql"

4) XML: предоставляет теги для работы с xml.

Использует префикс "x" и uri "http://java.sun.com/jsp/jstl/xml"

5) Functions: предоставляет функции для работы со строками.

Использует префикс "fn" и uri "http://java.sun.com/jsp/jstl/functions"



**25. Особенности спецификаций в Java Platform EE и возможности API в Java EE. Концепция распределенной и удаленной обработки данных.**

Java Platform, Enterprise Edition (Java EE) – это набор спецификаций, API и технологий для разработки и развертывания приложений, которые могут быть запущены на серверах приложений. Java EE содержит множество API и сервисов, которые обеспечивают различные функциональные возможности, такие как веб-сервисы, многопоточность, управление транзакциями, безопасность и др.

Java EE спецификации определяют стандартные интерфейсы и протоколы для взаимодействия между компонентами приложений, такими как сервлеты, EJB и JMS. Кроме того, Java EE включает в себя набор расширений и API для работы с базами данных, веб-сервисами, мессенджерами и другими распространенными технологиями.

Одной из ключевых функций Java EE является концепция распределенной обработки данных.

Распределенная обработка данных позволяет приложениям выполнять вычисления и обработку данных на нескольких серверах или узлах сети, что обеспечивает более высокую отказоустойчивость и масштабируемость.

Для реализации распределенной обработки используются технологии:

- Remote Method Invocation (RMI)

- Java Message Service (JMS)

- Java Naming and Directory Interface (JNDI)

- Enterprise JavaBeans (EJB).

Эти технологии позволяют создавать распределенные приложения, в которых компоненты могут работать на разных серверах или узлах сети, а обмен данными между ними осуществляется посредством специальных протоколов и интерфейсов.

**26. Архитектура и уровни распределенных систем. Модели распределенных вычислений. Многозвенные распределенные системы. Компонентные системы Java EE. Базисные архитектуры.**

Характеристики информационной системы:

- звенья (2 звена: клиент-сервер, 3 звена: клиент-сервер-БД)

- слои

- уровни (уровень клиента, уровень клиента, сервер БД)

Распределенная система – система, функции которой разделены между несколькими ПК; система, способная выполнять функции нескольких клиентов.

«Тонкий» клиент – конфигурационная часть системы, располагающая минимумом аппаратных средств и возможностей.

«Толстый» клиент – конфигурационная часть системы, располагающая максимумом аппаратных средств и возможностей.

Уровни распределенных систем:

1. Уровень приложений - содержит приложения, которые используются в распределенной системе, такие как клиентские приложения, веб-приложения, сервисы и т.д.
2. Уровень сервисов - содержит сервисы, которые предоставляются распределенной системой, такие как веб-сервисы, сервисы обработки сообщений и т.д.
3. Уровень протоколов - содержит протоколы для обмена данными между компонентами системы. Это может быть протокол HTTP, протоколы сериализации данных и т.д.
4. Уровень транспорта - содержит технологии, используемые для передачи данных между устройствами, такие как TCP/IP, UDP и т.д.

Самые распространенные модели распределенных вычислений, определяющие способы взаимодействия между компонентами распределенной системы:

* Модель удаленного вызова процедур (RPC) - позволяет вызывать удаленные процедуры, которые выполняются на другом компьютере в распределенной сети.
* Модель сообщений (Messaging) - позволяет передавать сообщения между компонентами распределенной системы с использованием протоколов, таких как JMS (Java Message Service).
* Модель распределенных объектов (CORBA, RMI) - позволяет создавать объекты, которые могут быть использованы удаленно другими компонентами распределенной системы.

Многозвенные распределенные системы – это системы, которые состоят из нескольких подсистем, которые могут работать независимо друг от друга и взаимодействовать только через определенные интерфейсы. Многозвенные: клиент, бизнес-логика, веб-звено, звено БД

Компоненты приложения: J2EE прикладные клиенты, J2EE web-компоненты, J2EE компоненты EJB

Компонент - самостоятельная функциональная программная единица, которая собирается в приложение J2EE.

Java EE:

- приложения на клиенте

- сервлеты JSP, JSF – веб-компоненты

- EJB- компоненты

Клиент J2EE:

- тонкие клиенты (нет запросов в БД)

- две части: динамические web-страницы, web-браузер

**27. Архитектура RMI: интерфейсы и их реализация, стабы, скелетоны, транспортный уровень. Метод организации вызовов удалённых процедур (RPC)**

Удаленный объект имеет методы и реализует удаленный интерфейс.

RMI (Remote Method Invocation) – программный интерфейс вызова удаленных методов в Java. Поддерживает два класса:

- реализация поведения на сервере

- роль proxy (работает на стороне клиента)

Прокси-сервер – служба, позволяющая клиентам и другим сетевым службам выполнять косвенные запросы. Клиент подключается → прокси запрашивает ресурс → подключается к указанному серверу → получает ресурс.

Удаленный интерфейс должен удовлетворять следующим требованиям:

- удаленный интерфейс должен расширять интерфейс java.rmi.Remote

- каждое объявление метода в удаленном интерфейсе должно включать исключение java.rmi.RemoteException

- в объявлении удаленного метода удаленный объект, объявленный как параметр или возвращаемое значение, должен быть объявлен как удаленный интерфейс

Слои:

- стаб, скелетон

- Reference Layer (слои удаленных ссылок)

- Транспортный (TCP, UDP)

Lookup() – метод для выбора хоста, ищет удаленный хост.

Для передачи сообщений между удаленными объектами используются стабы (заглушка) и скелетоны.

Стаб (заглушка) – локальный представитель удаленного объекта и является удаленной ссылкой. Особенности вызова стаба:

- передача параметров по значению

- обязательно remoteException

- при вызове метода мы работаем с удаленным интерфейсом

- все объекты должны быть serializable

- при передаче удаленного объекта передается стаб

Задачи стаба:

- коммуникация с удаленным объектом

- осуществление обмена информацией клиент-сервер

- (де) маршализация

Работа заглушки:

- прикладной объект реализует интерфейс, объявляющий его прикладные методы

- заглушка реализует тот же интерфейс, но методы заглушки не содержат прикладной логики

- методы заглушки реализуют только некоторые сетевые операции для направления запроса прикладному объекту и принятия результата

- когда клиент вызывает прикладной метод заглушки, запрос передается через сеть посредством отправки имени вызванного метода и значений параметров

- когда приходит входной поток, он анализируется и отправляется запрашиваемый метод

Скелетон (skeleton) – это объект, который находится на удаленной машине и обрабатывает входящие вызовы методов от удаленных клиентов.

RPC (Remote Procedure Call) – технология, позволяющая вызывать функции в другом адресном пространстве.

Два его компонента:

- сетевой протокол для обмена данными в режиме клиент-сервер

- язык сериализации объектов

**28. Инсталляция программных компонентов в RMI, автоматическая загрузки классов, сериализация, параметризация, маршализация, демаршализация.**

Программные компоненты, требующиеся для работы RMI:

- класс с вызовами удаленных методов

- удаленный интерфейс Remote Interface, описывающий методы, которые можно будет вызвать удаленно

- класс, реализующий удаленный интерфейс

- запуск RMI Registry (метод RMI для регистрации) на сервере, который будет обслуживать удаленные методы

- регистрация объекта, использующего удаленный интерфейс (создание объекта стаба)

Сериализация – запись объектов в файлы в форме байтовых потоков.

Маршалинг – процедура на стороне клиента вызова метода в форме удобной для передачи по сети.

Демаршалинг – преобразование, обратное маршалингу на стороне сервера, когда соответствующие компоненты ПО принимается вызов, поступивший в форме пакета данных.

Параметризация – механизм, который позволяет создавать классы, методы и интерфейсы, которые могут работать с различными типами данных, указанными при создании экземпляра класса или вызове метода.

**29. Концепция интеграции различных изолированных объектных систем. Архитектурные решения и механизмы стандарта CORBA.**

Задача интеграции - обеспечить эффективный, надежный и безопасный обмен данными между различными программными продуктами, изначально не предназначенными для совместной работы.

Существует два варианта построения интеграционного решения:

- горизонтальная интеграция - интеграция ИС или приложений, относящихся к одному уровню,

- вертикальная интеграция - интеграция приложений и систем, находящихся на различных уровнях информационной пирамиды.

Для обоснованного выбора способа интеграции необходимо обладать информацией о характеристиках интегрируемых приложений, а также уметь оценивать влияние способа интеграции на общую архитектуру интеграционного решения.

Основные критерии, влияющие на выбор способа интеграции приложений:

1) степень связывания приложений. Зависимости между интегрируемыми приложениями должны быть сведены к минимуму. Сильное связывание - наличие большого числа допущений между приложениями. Результатом сильного связывания являются неустойчивые, плохо масштабируемые решения. В идеале интерфейсы объединяемых приложений должны обеспечивать необходимую функциональность, допуская возможность изменения внутренней реализации;

2) простота поддержки интеграционного решения. Следует стремиться к минимизации изменений, вносимых в объединяемые приложения, и объема кода, необходимого для интеграции;

3) объем данных. Несвоевременная доставка или потеря данных -> рассинхронизация приложений;

4) стоимость решения. Отдельные интеграционные решения требуют применения специального программного обеспечения (middleware), что существенно увеличивает стоимость проекта интеграции. Вместе с тем более бюджетная разработка проекта интеграции «с нуля» намного более рискованна и может свести усилия разработчиков к «изобретению велосипеда».

Термином CORBA (Common Object Request Broker Architecture (общая архитектура брокера объектных запросов) обозначают технологию, архитектуру и набор спецификаций и стандартов промежуточного программного обеспечения (middleware) объектного типа для создания распределенных программных приложений, причем акцент делается на слове "распределенных".

Базовые принципы:

- Независимость от физического размещения объекта. Компоненты ПО не обязаны находиться в одном исполняемом файле или размещаться на одной аппаратной системе.

- Независимость от платформы. Компоненты могут выполняться на различных аппаратных и операционных платформах, взаимодействуя друг с другом в рамках единой системы.

- Независимость от языка программирования. Cвязываемые приложения могут быть написаны на разных языках программирования, в частности, не обязательно объектно-ориентированных.

Это клиент-серверная технология, в которой функциональность объекта предоставляется клиенту посредством обращения к абстрактным интерфейсам. Интерфейс определяет набор методов, которые реализуют функции, присущие данному классу объектов. Интерфейс дает клиенту возможность только вызывать тот или иной метод, скрывая от него все детали его реализации.

Суть – независимая от языка работа с удалёнными объектами + сопутствующие сервисы.

**30. Поддержка и реализация механизмов CORBA на платформе Java. GIOP, IIOP, CORBA-объект. Механизм POA.**

Java предоставляет поддержку для реализации механизмов CORBA с помощью пакета javax.corba.\*.

Взаимодействие систем происходит как на локальной машине, так и по сети.

CORBA-объект - виртуальное понятие: он представляет собой нечто, расположенное на брокере объектных запросов, посылающее запросы к другим CORBA-объектам - серверным объектам и получающее запросы от других CORBA-объектов - клиентов. Не имеет физической реализации, обращение к нему осуществляется по объектной ссылке (object reference), которая в отличие от самого объекта является вполне реальной и представляет собой последовательность символов.

Идентификатор объекта (object ID) - уникальное имя объекта внутри его объектного адаптера. Он представляет собой последовательность байт, которая ассоциируется с объектом в момент его создания. Обеспечивается программным приложением либо портируемым объектным адаптером (Portable Objective Adapter - РОА). Не обязан быть уникальным для сервера. Там объект известен под своей объектной ссылкой. Как правило, идентификатор объекта является частью объектной ссылки. Клиент при обращении к целевому объекту по полной объектной ссылке. Есть еще объектный ключ (тоже часть объектной ссылки), который используется GIOP (General Inter-ORB Protocol), протоколом взаимодействия брокеров объектных запросов, для идентификации конечной точки связи.

Для описания ORB и других компонент CORBA-стандарта используется язык определения интерфейсов (IDL - Interface Definition Language). IDL - это только описания, декларация, пассивные определения атрибутов, родительских классов, типов поддерживаемых событий, методов, включая входные и выходные данные, типы данных, исключительные ситуации для обработки ошибок.

Сервант - серверная программа, написанная на каком-либо из языков программирования и выполняющая CORBA-объект. Это может быть набор функций, представляющих состояние объекта, или реализации определенных классов серверного приложения. Сервант написан на том же программном языке, что и приложение, и является программной реализацией объекта CORBA.

Необходимо отделять CORBA-объекты от сервантов. CORBA-объект не способен ответить на запрос клиента, так что этим занимается сервант. С другой стороны, CORBA-объекты могут иметь состояние, в то время как серванты не обязательно имеют его. Например, состояние CORBA-объекта может храниться в БД. В этом случае соотвествующий сервант занимается извлечением и модификацией объекта - записи базы данных, но сам состояния не имеет.

Скелетон - серверная программа, которая связывает сервант с объектным адаптером, позволяя объектному адаптеру перенаправлять запросы к соответствующему серванту.

Активизация - запуск существующего CORBA-объекта для обработки клиентских запросов. Активизация предполагает, что интересующий клиента объект имеет подходящий сервант. Создание серванта может входить в процесс активизации. Активизированные объекты бывают двух типов: устойчивые и временные. Устойчивые объекты существуют и после останова процесса, который их создал или активизировал. Временные объекты - это объекты, чей жизненный цикл ограничен процессом или даже объектным адаптером, который их создал. Это необходимо, например, в следующем случае: одно приложение запрашивает другое с требованием обратного запроса-отклика, не дождавшись которого завершает свою работу. В этом случае отклик должен быть временным.

Деактивизация - действие, обратное активизации, остановка CORBA-объекта, разрыв связки между объектом и сервантом, в общем случае без разрушения объекта. В дальнейшем CORBA-объект может быть вновь активизирован. Разрушение CORBA-объекта обязательно вызывает деактивизацию.

Инкарнация - связывание серванта с CORBA-объектом для обработки клиентского запроса. Инкарнация материализует в серванте виртуальный CORBA-объект. В отличие от активизации инкарнация относится не к объекту, а к его серванту.

Эфемеризация - в противоположность инкарнации разрушение связки CORBA-объект - сервант со стороны серванта. Так же как инкарнация, эфемеризация является операцией серванта.

Карта активных объектов (Active Object Map) - таблица объектного адаптера, в которой он ведет реестр активных CORBA-объектов и связанных с ними сервантов. Первые представлены в карте своими идентификаторами.

За исключением редкого случая прямых вызовов методов между классами одного и того же языка программирования необходим механизм кодирования вызова метода в некоторую последовательность байт (byte stream) у клиента и декодирования этой последовательности у сервера. Для этой цели спецификация CORBA определяет Общий Протокол обмена между Брокерами Объектных Запросов (General Inter-Orb Protocol - GIOP). Кроме того, определен протокол передачи сообщений протокола GIOP поверх транспортного протокола TCP/IP, являющегося основным видом взаимодействия в Internet, ввиду чего этот протокол получил название Протокола обмена между Брокерами Объектных в Internet (Internet Inter-Orb Protocol - IIOP). Протокол IIOP должен поддерживаться всеми Брокерами Объектных Запросов независимо от особенностей их реализации, что является главным требованием для обеспечения взаимодействия между произвольными ORB-ами двух разных и совершенно независимых производителей.

Есть два стандартных объектных адаптера — BOA (Basic Object Adapter) и POA (Portable Object Adapter).

В CORBA-приложениях используется древовидная иерархия объектных адаптеров. В корне ее находится так называемый Root POA — объектный адаптер по умолчанию. Дочерние POA создаются с помощью обращения к уже созданным POA как к фабрикам. Перед созданием дочерних POA желательно создать так называемый «менеджер» POA. Он отвечает за распределение клиентских запросов между сервантами, находящимися под управлением различных POA, а также за управление их циклом жизни. Фабрикой такого менеджера может являться Root POA. При создании дочерних объектных адаптеров вы указываете менеджер POA в качестве аргумента. Если вы не создали свой менеджер, то будет неявно создан и использован менеджер по умолчанию. Процесс уничтожения объектных адаптеров происходит так — сначала дочерние POA, затем их «родители».

Объектный адаптер POA, по определению OMG, отображает понятие программно-реализованных сервантов на концепцию CORBA-объектов. Объектный адаптер превращает серверные объекты в CORBA-объекты или извлекает CORBA-объекты из серверного приложения.

Обязанности POA: создавать CORBA-объекты и их объектные ссылки; демультиплексировать запросы на каждый серверный CORBA-объект; перенаправлять запросы к соответствующему серванту, который беспечивает реализацию серверного CORBA-объекта; активизировать и деактивизировать CORBA-объекты и, соответственно, инкарнировать и эфемеризировать соответствующие серванты.

Адаптер переводит обращение к CORBA-объекту на язык, понятный серванту, т.е программному приложению. В свою очередь, скелетон может использоваться для перевода параметров клиентского запроса в форму аргументов операций серванта.

Работа сервера на POA:

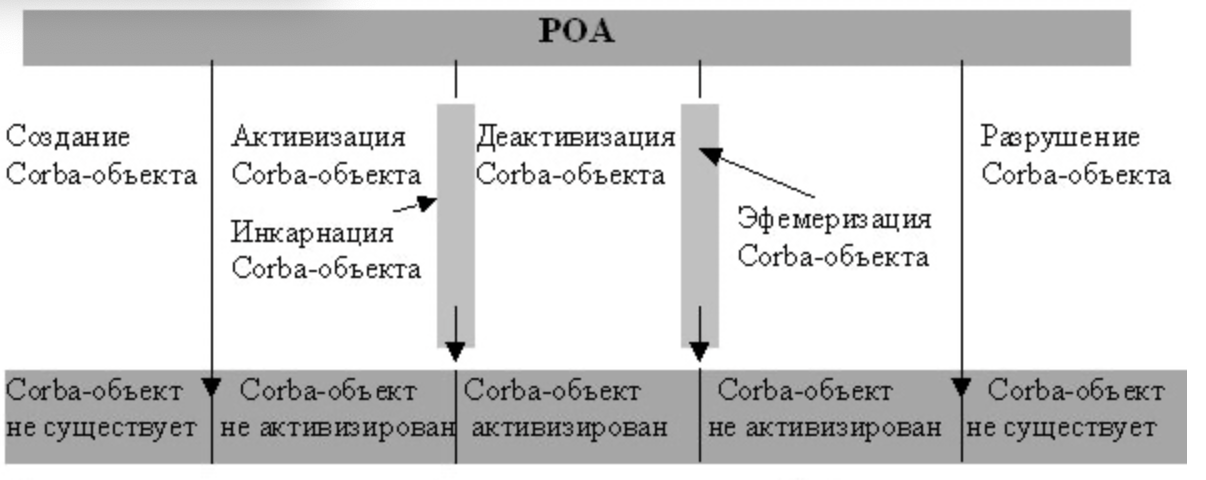
1) получение ссылки на корневой адаптер POA

2) определение нового POA

3) создание POA

4) создание серванта и его активация

5) активация POA вызовом его менеджера



Архитектура:

1) ORB определяет объектную шину

2) Сервисы CORBA

3) Общие сервисы (создание стабов и скелетонов)

4) Application Objects – прикладные объекты и приложения

Процесс разработки приложения:

1. Описание интерфейса удаленного объекта с помощью idl (interface definition language)

2. Обработка IDL компилятором idjl. В результате генерируются программные прослойки, содержащие Java-версию описания интерфейса и код классов для них (stub у клиента и скелетон у сервера). Прослойки обеспечивают подключение к ORB.

3. Программирование сервера и клиента . Клиент стартует свой ORB, ищет нужный ему сервер через службу имен, получает ссылку на удаленный объект и вызывает его методы

4. Старт службы имен, потом сервер/клиент

CCM – компонентная модель CORBA. Она описывает стандартный каркас приложения для компонент CORBA и содержит 16 сервисов (имён, управления событиями, транзакций, запросов, безопасности).

**31. Язык описания интерфейсов IDL: синтаксис, интерфейсы, ограничения, компиляторы IDL. Стандартизация для различных языков программирования.**

Чтобы объекты могли взаимодействовать так, как если бы они были на одном языке, они должны использовать общее внешнее представление. Этот промежуточный язык должен быть простым в использовании, легким в изучении и обусловливать минимальные расходы. Таким промежуточным языком является язык OMG Interface Definition Language,обычно называемый просто IDL.

Значительное влияние на разработку синтаксиса языка OMG IDL оказал популярный в настоящее время объектно-ориентированный язык C++. Определение интерфейса в IDL начинается с ключевого слова interface. Это определение состоит их четырех частей: имя интерфейса, базовое имя интерфейса, тело интерфейса и атрибуты интерфейса. Тело интерфейса представляет собой просто набор определений методов и операторов определения типов

В IDL выделяются типы данных. К числу базовых типов отнесены типы данных, традиционно используемые в популярных языках программирования, - тип целых чисел в различных представлениях, тип чисел с плавающей точкой, литерный и строковый типы, тип логических значений, 8-битовый тип, значения которого не подвергаются каким-либо преобразованиям при передаче данных, и, наконец, тип "any" (любой), обозначающий любой базовый или конструируемый тип.

Конструируемые типы включают: тип "struct", определяемый как упорядоченное множество пар (имя, значение), тип "union" (объединение), а также перечислимый тип "enum", определяющий упорядоченную последовательность идентификаторов.

Для удобства введены также шаблонные типы, к числу которых отнесены тип "sequence" (последовательность), определяющий переменной длины массивы значений одного типа с заданным или неопределенным числом элементов, а также тип "string" (строка), определяющий последовательности символов алфавита языка заданной или неопределенной длины.

Налагаются также некоторые ограничения на синтаксические конструкции, общие для обоих языков. Так, в IDL обязательно должен специфицироваться тип возвращаемого функцией значения. Для каждого формального параметра в объявлении операции должно специфицироваться имя. Исключаются спецификации типа "char" с ключевыми словами "signed" или "unsigned" и т.д.

IDL-компилятор представляет собой инструмент для перевода IDL-определений класса в код, который состоит из коллекции «каркасных» определений классов, перечислимых типов.

Когда компилируется idl, создаются пакет (module) и 6 файлов:

⁃ WordCollection - код Java-интерфейса, сгенерированный из idl

⁃ WordCollectionOperations - содержит методы из операций idl, он используется стабами клиентов и скелетонами серверов

⁃ WordCollectionPOA - скелетон, обеспечивающий базовую функциональность CORBA для сервера

⁃ \_WordCollectionStub - стаб, базовая функциональность CORBA для клиента

⁃ WordCollectionHelper - вспомогательная функциональность, например метод narrow (преобразование обьектных ссылок корба к типам джавы)

⁃ WordCollectionHolder - операции чтения/записи в потоки корбы

**32. Унифицированные средства, службы, механизмы и ЯП в технологии CORBA. Брокер объектных запросов и его основные задачи.**

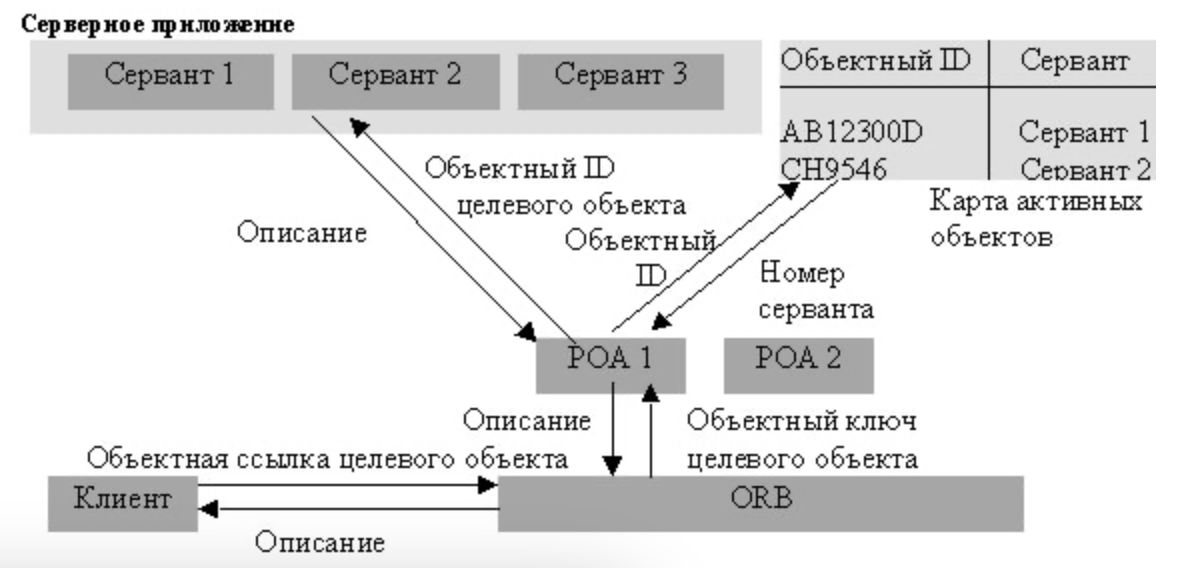
CORBA включает в себя несколько групп реализаций объектов, а именно прикладные объекты, объектные службы, общие средства и домены. Прикладные объекты (Application Objects) представляют собой реализации объектов для конкретных пользовательских приложений, например, объекты для поддержки специфических бизнес-процессов.

Реализации объектов, предоставляющие общие для любой объектно-ориентированной среды возможности, входят в категорию объектных служб (CORBA services): служба имен, служба событий, служба сохранения в долговременной памяти, служба транзакций и т.д.

Общие средства (CORBA facilities) - это реализации объектов, необходимые для большого числа приложений, например, поддержка составных документов, потоков заданий и др.

Ядром архитектуры CORBA является брокер объектных запросов (Object Request Broker, ORB). Это объектная шина, по которой в стиле классического механизма RPC, происходит взаимодействие локальных и удаленных объектов. Помимо самого вызова метода удаленного объекта, ORB отвечает за поиск реализации объекта, его подготовку к получению и обработке запроса, передачу запроса и доставку результатов клиенту.

Как осуществляется вызов клиентом объекта серверного приложения? Этот вызов обращается к объекту по объектной ссылке, которая уникально идентифицирует CORBA-объект в соответствующем пространстве имен. Через брокер объектных запросов запрос передается на сервер. Как уже упоминалось, часть объектной ссылки, объектный ключ, уникально идентифицирует объект в его серверном приложении. Этот объектный ключ позволяет ORB выбрать именно тот адаптер, который отвечает за вызываемый объект (к одному приложению может относиться несколько адаптеров). POA выделяет из объектного ключа объектный идентификатор, и по объектному идентификатору определяет, какой сервант связан с вызываемым объектом. Адаптер может узнать это по специальной Карте Активных Объектов, или вызвать приложение и попросить его обеспечить нужный сервант по идентификатору объекта, или использовать сервант, выставленный приложением по умолчанию. Далее с запросом начинает работать сервант. Он отвечает за выполнение запроса и возвращает результаты обратно к объектному адаптеру, который передает их брокеру объектных запросов, а тот в свою очередь - клиенту. Таким образом запрос передается из рук в руки, как эстафетная палочка.



**33. Понятие, назначение EJB-технологии. Создание корпоративных приложений на основе распределенных компонентных технологий.**

Enterprise JavaBeans (EJB) — спецификация технологии написания и поддержки серверных компонентов, содержащих бизнес-логику.

Компоненты EJB выполняются внутри EJB-контейнера, который, в свою очередь, выполняется внутри EJB-сервера. EJB-компонент - Java-класс, который реализует некую бизнес-логику.

EJB-контейнер — это то место, где «живет» EJB-компонент. EJB-контейнер реализует для находящихся в нем компонентов такие сервисы, как транзакции, управление ресурсами, управление версиями компонентов, их настраиваемостью, мобильностью, жизненным циклом. Как правило, в одном EJB-контейнере живет несколько однотипных EJB-компонентов. Клиентские приложения вызывают методы на удаленных EJB-компонентах через EJB-объект. EJB-объект реализует «удаленный интерфейс» EJB-компонента на сервере. Суть в том, что находящийся на сервере EJB-компонент, помимо бизнес-функций должен реализовывать также некоторые функции, которые служат для «управления» EJB-компонентом со стороны контейнера. EJB-объект реализует лишь бизнес-интерфейс для EJB-компонента, являясь «промежуточным» звеном между клиентом и EJB-компонентом. У EJB компонентная структура. Приложения, написанные однажды, могут быть развёрнуты на любой серверной платформе, поддерживающей спецификацию EJB.

Компонентно-ориентированный подход (КОП) к проектированию и реализации программных систем и комплексов является в некотором смысле развитием объектно-ориентированного подхода и более пригоден для разработки крупных распределенных программных систем (например, корпоративных приложений). С точки зрения КОП программная система - это набор компонентов с четко определенным интерфейсом. В отличие от других подходов программной инженерии, изменения в систему вносятся путем создания новых компонентов или изменения старых, а не путем рефакторинга существующего кода. Программный компонент - автономный элемент ПО, предназначенный для многократного использования, который может распространяться для использования в других программах в виде скомпилированного кода. Подключение к программным компонентам осуществляется с помощью открытых интерфейсов, а взаимодействие с программной средой осуществляется по событиям, причем в программе, использующей компонент, можно назначать обработчики событий, на которые умеет реагировать компонент. Применение компонентного программирования призвано обеспечить более простую, быструю и прямолинейную процедуру первоначальной инсталляции прикладного программного обеспечения, а также увеличить процент повторного использования кода, т.е. усилить основные преимущества ООП. Компоненты – это существенно более крупные единицы, чем объекты + независимы от ЯП.

**34. Классы, интерфейсы и методы обеспечения работы бинов и EJB. ЖЦ EJB-бина и его содержимое. Удаленный и локальный интерфейсы.**

Приложение-клиент соединяется с EJB-сервером и посылает ему запрос на создание ``бина'' для обработки своих запросов. Сервер отвечает на такой запрос созданием объекта на стороне сервера (экземпляр EJB-компоненты) и возвращает клиенту прокси-объект (EJB-объект), чей интерфейс совпадает с интерфейсом созданной EJB-компоненты и чьи методы перенаправляют вызовы собственно экземпляру компоненты. После этого приложение-клиент работает с EJB-объектом как с локальным объектом, не подозревая, что всю работу выполняет не EJB-объект, а удаленная компонента на сервере. Созданием и удалением EJB-компонент на сервере занимается EJB-контейнер.

У каждой EJB-компоненты есть то, что называют ``home interface'', который определяет методы создания, инициализации, удаления и (в случае entity beans) поиска экземпляров EJB-компонент на стороне сервера. Он описывает возможные взаимодействия между компонентой и контейнером.

Home interface для EJB-компоненты наследуется от интерфейса javax.ejb.EJBHome, который представляет базовую функциональность для взаимодействия между контейнером и компонентой. Все методы этого интерфейса должны быть RMI-совместимы. Интерфейс также описывает один или более create() методов, которые все называются ключевым словом create, но тело которых различно. Все create методы возвращают объект с ``remote'' для данной компоненты интерфейсом.

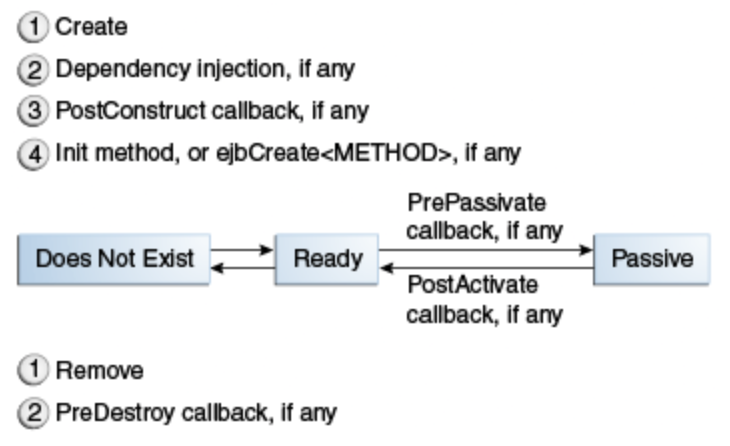
Когда приложение-клиент хочет создать ``бин'' на сервере, оно вначале использует Java naming and Directory Interface (JNDI) для нахождения объекта с искомым интерфейсом. JNDI представляет собой стандартное расширение Java-пакета и предоставляет глобальный сервис для любого Java-окружения, позволяя Java-программам находить и использовать существующие ресурсы просто по имени, а также позволяя просто извлекать информацию о ресурсах. Использование EJB сервисов, аналогичных JNDI, еще раз подчеркивает тот факт, что цель EJB-систем - полная совместимость с Java API.

Когда клиент получил ссылку на объект с ``home'' интерфейсом, он вызывет у этого объекта любой из методов create для создания экземпляра компоненты на сервере. При получении вызова create локальный (находящийся у клиента) объект с ``home'' интерфейсом удаленно вызывает метод у EJB-контейнера, который и создает экземпляр ``бина'', возвращая клиенту EJB-объект. После этого приложение-клиент вызывает у EJB-объекта методы, исполнение которых перенаправляется на сервер (сначала контейнеру, а потом уже компоненте). Контейнер, при получении вызова к компоненте, просто вызывает у нее соответствующий метод. Единственное, за что ответственен при этом контейнер - обработка некоторых ошибок (таких, например, как отсутствие искомого экземпляра класса) и соответственное возбуждение исключительных ситуаций.

Entity ``бины'', помимо упоминавшихся, имеют дополнительный интерфейс finder, который используется для поиска по уникальному ключу экземпляра ``бина''.

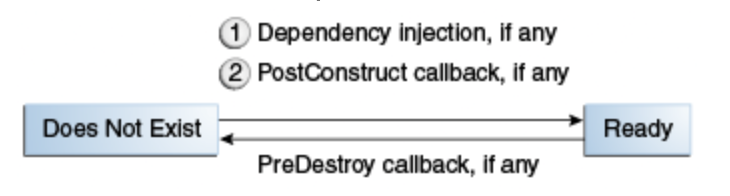
И, как уже говорилось, ``home interface'' включает в себя методы, сообщающие контейнеру о необходимости удалить экземпляр компонента. В этом случае сервер производит удаление экземпляра, и любая следующая попытка вызвать метод у удаленной компоненты приведет к возникновению исключительной ситуации.

Жизненный цикл сессионного бина с сохранением состояния

Клиент инициирует жизненный цикл, получая ссылку на сессионный компонент с состоянием. Контейнер выполняет инъецирование всех зависимостей, а затем вызывает метод @PostConstruct, если таковой имеется. Теперь компонент готов к вызову его бизнес-методов клиентом.

На этапе готовности (Ready) контейнер EJB может решить деактивировать (Deactivate или Passivate) компонент путём перемещения его из памяти во вторичное хранилище. (Как правило, контейнер EJB берёт самый редко используемый компонент для деактивации.) Контейнер EJB вызывает аннотированный @PrePassivate метод, если таковой имеется, непосредственно перед его деактивацией. Если клиент вызывает бизнес-метод бина, который находится на этапе деактивации, EJB-контейнер активирует его, вызывает аннотированный @PostActivate метод , если таковой имеется, и затем переводит его на этап готовности. В конце ЖЦ клиент вызывает аннотированный @Remove метод, а контейнер EJB вызывает аннотированный @PreDestroy метод, если таковой имеется. Объект компонента готов к сборке мусора. Ваш код управляет вызовом только одного метода ЖЦ — аннотированного @Remove. Все остальные методы вызываются контейнером EJB.

Жизненный цикл сессионного бина без сохранения состояния

Поскольку сессионный компонент без сохранения состояния никогда не деактивируется, его жизненный цикл состоит только из двух этапов: несуществующего и готового к вызову бизнес-методов.

Контейнер EJB обычно создаёт и поддерживает пул сессионных компонентов без сохранения состояния. Контейнер выполняет инъецирование всех зависимостей, а затем вызывает аннотированный @PostConstruct метод, если он существует. Теперь компонент готов к тому, чтобы его бизнес-методы вызывались клиентом. В конце ЖЦ контейнер EJB вызывает аннотированный @PreDestroy метод, если он существует. Объект компонента готов к сборке мусора.

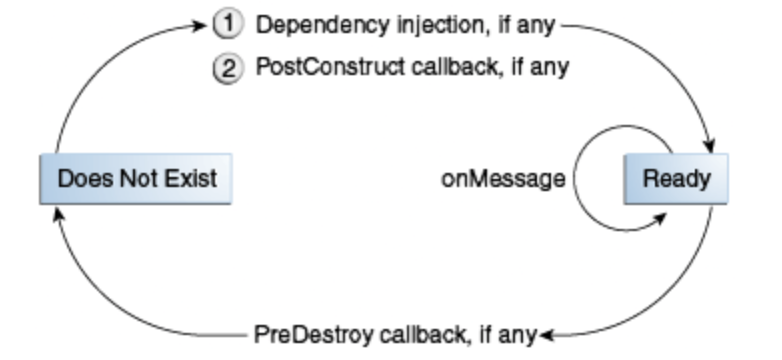
Жизненный цикл сессионного бина-синглтона

Как и сессионный компонент без сохранения состояния, сессионный компонент-синглтон никогда не деактивируется и имеет только два этапа: не существует и готов к вызову бизнес-методов.

Контейнер EJB инициирует жизненный цикл сессионного компонента-синглтона, инстанцируя объект его класса. Если синглтон помечен аннотацией @Startup, это происходит при развёртывании приложения. Контейнер выполняет инъецирование всех зависимостей и затем вызывает метод, аннотированный @PostConstruct, если он существует. Теперь сессионный компонент-синглтон готов к тому, чтобы его бизнес-методы вызывались клиентом. В конце жизненного цикла контейнер EJB вызывает аннотированный @PreDestroy метод, если он существует. Сессионный компонент-синглтон теперь готов для сборки мусора.

Жизненный цикл бина, управляемого сообщениями

Контейнер EJB обычно создаёт пул объектов, управляемых сообщениями. Если управляемый сообщениями компонент использует инъецирование зависимостей, контейнер инъецирует зависимости до инстанцирования объекта. Контейнер вызывает аннотированный @PostConstruct метод, если таковой имеется. Как и сессионный компонент без сохранения состояния, управляемый сообщениями компонент никогда не деактивируется и имеет только два состояния: не существует и готов к приёму сообщений. В конце ЖЦ контейнер вызывает аннотированный @PreDestroy метод, если таковой имеется. Объект компонента готов к сборке мусора.



Компонент может иметь следующие интерфейсы:

Удаленный интерфейс (Remote Interface) определяет прикладные методы компонента, доступные из приложений, внешних по отношению к контейнеру EJB, то есть прикладные методы, которые компонент представляет внешним приложениям. Удаленный интерфейс расширяет (extends) интерфейс javax.ejb.EJBObject.Внешние по отношению к EJB приложения вызывают методы удаленного интерфейса, а получают эти методы они при помощи домашнего интерфейса.

Домашний интерфейс (Home Interface) определяет методы жизненного цикла компонента, доступные из приложений, внешних по отношению к контейнеру EJB, то есть методы создания новых компонентов, удаления и поиска существующих компонентов. Домашний интерфейc расширяет интерфейс javax.ejb.EJBHome. Чтобы создать новый EJB или обратиться к существующему EJB, внешнее приложение должно сначала вызвать соответствующий метод create() его домашнего интерфейса.

Локальный интерфейс определяет прикладные методы EJB, которые могут использоваться другими компонентами, размещенными в том же контейнере EJB,но не внешними по отношению к контейнеру EJB приложениями. Он позволяет компонентам, находящимся в одном контейнере EJB, взаимодействовать без дополнительных затрат, связанных с применением протокола распределенных объектов, что повышает их производительность. Локальный интерфейс расширяет интерфейс javax.ejb.EJBLocalObject.Внешние по отношению к EJB компоненты, находящиеся в том же самом контейнере, работают с методами локального интерфейса, а получают эти методы они при помощи локального домашнего интерфейса.

Локальный домашний интерфейс (Local Home Interface) определяет методы ЖЦ компонента, доступные из компонентов в том же контейнере EJB (методы создания новых компонентов, удаления и поиска существующих компонентов). Это позволяет компонентам взаимодействовать без дополнительных затрат, связанных с использованием протокола распределенных объектов, что улучшает их производительность. Домашний интерфейс расширяет интерфейс javax.ejb.EJBLocalHome. Чтобы создать новый EJB или обратиться к существующему EJB,; компонент, находящийся в том же контейнере EJB, должен сначала вызвать соответствующий метод его домашнего интерфейса.

Важным отличием между удаленным и локальным интерфейсом является то, что вызовы методов локального интерфейса не используют RMI и не генерируют RemoteException.Для корректной работы не обязательно предоставлять классу компонента все четыре интерфейса. К примеру, если компонент не используется никаким клиентским приложением (а нужен только другим компонентам, работающим под управлением того же EJB контейнера), то нет необходимости предоставлять удаленный интерфейс. То есть в зависимости от тех задач, которые решает данный компонент, ему может быть предоставлен либо локальные, либо удаленные, либо и те и другие интерфейсы.

**35. Сессионные компоненты в EJB. Виды сессионных бинов, сравнительная хар-ка, использование. Особенности stateful-бинов**

Сессионный компонент Session Beans, иначе называемый сеансовый, вызывается клиентом (браузером) для выполнения таких операций, таких как проверка кредитной истории клиента. Слово «сессионный» предполагает, что экземпляр компонента доступен только на время выполнения определенной задачи сервером, и безвозвратно уничтожается в случае аварии или остановки сервера. Для доступа к серверной части приложения, клиент вызывает методы сессионного компонента, выполняющего определенные бизнес-задачи внутри сервера.

Сеансовый компонент с сохранением состояния EJB (stateful) автоматически сохраняет свое состояние между обращениями к нему от одного и того же клиента, и завершает свое существование либо по таймауту, либо по явному запросу клиента. Типичным примером компонента с сохранением состояния является корзина с покупками в интернет-магазине. Для каждого stateful-бина может существовать только 1 клиент. Состояние объекта состоит из значений переменных его экземпляра.

Сеансовые компоненты без сохранения состояния EJB (stateless) не хранят информации о своем состоянии и являются прикладными службами, которые выполняют все действия в рамках запроса. EJB stateless можно использовать для реализации таких операций, как перевод средств на кредитную карту. На основе stateless-бинов проектируются WEB-сервисы. Он недолговечен, ЖЦ ограничен циклом клиента, работает от лица конкретных отдельных клиентов.

При создании сеансового компонента без состояния (Stateless Session Bean) контейнер EJB выполняет следующую последовательность действии:

- При помощи метода Class.newInstance () создается экземпляр класса компонента.

- Вызывается метод SessionBean.setSessionContext( SessionContext context). При помощи этого метода экземпляру компонента передается ссылка на объект класса EJBContext, предоставляющий интерфейс для взаимодействия с контейнером EJB (в частности, в этом объекте есть метод lookup(), позволяющий искать другие объекты по путям JNDI).

- Вызывается метод ejbCreate(). У stateless сеансовых компонентов этот метод не имеет параметров. В этом методе компонент может создать подключения к базам данных и к другим ресурсам.

- Контейнер помещает созданный компонент в пул готовых компонентов.

Компоненты-одиночки EJB (singleton) используются совместно всеми клиентами, имеющими к ним доступ, и продолжают свое существование на протяжении всего времени работы приложения. Информацию о своем состоянии EJB singleton сохраняет. Компонент-одиночку можно использовать, к примеру, в интернет-магазине для реализации скидки, поскольку правила предоставления скидки фиксированы и распространяются на всех клиентов.

Сеансовые компоненты могут вызываться локально или удаленно, посредством Java RMI. Компоненты-одиночки и компоненты без сохранения состояния могут также экспортироваться в виде веб-служб SOAP (Simple Object Access Protocol) или REST (Representational State Transfer).

Создание session beans:

1) должен иметь минимум 1 метод

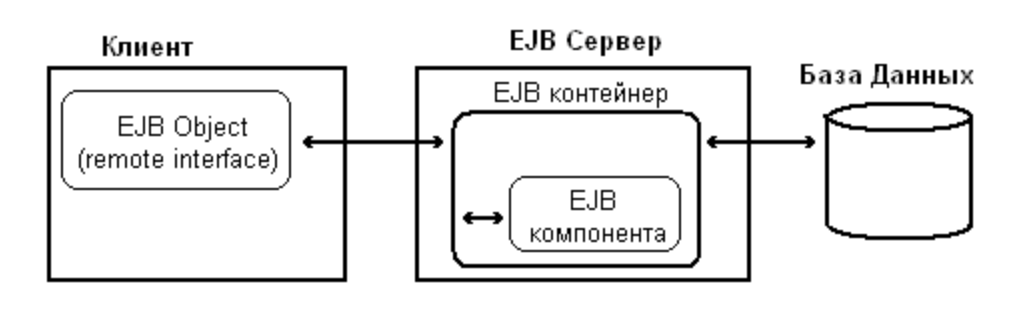
2) не должен быть абстрактным

3) должен иметь конструктор по умолчанию

4) методы не должны начинаться с ejb

5) свойства класса должны реализовывать интерфейс Serializable или объявлены примитивами

**36. Методы ЖЦ EJB. Взаимодействие клиента с компонентами EJB. POJO, POJI. ЖЦ компонента. Получение доступа к компоненту (аннотации, контект, jndi)**

Клиентские приложения вызывают методы на удаленных EJB-компонентах через EJB-объект (EJB-object). EJB-объект реализует ``удаленный интерфейс'' EJB-компоненты на сервере. Суть в том, что находящаяся на сервере EJB-компонента, помимо бизнес-функций, ради которых она была разработана, должна реализовывать также некоторые функции, определяемые спецификацией, которые служат для ``управления'' EJB-компонентой со стороны контейнера. EJB-объект реализует лишь бизнес-интерфейс для EJB-компоненты, являясь, в некотором смысле, ``промежуточным'' звеном между клиентом и EJB-компонентой.

EJB-объекты и EJB-компоненты представляют собой разные классы, хотя ``снаружи'' (при взгляде на их интерфейсы), они выглядят одинаково. Это происходит потому, что они реализуют один и тот же интерфейс (а именно, интерфейс, описанный для EJB-компоненты). Однако при этом они выполняют совершенно разные функции. EJB-компонента выполняется на сервере, внутри EJB-контейнера и реализует бизнес-логику, в то время как EJB-объект выполняется у клиента и удаленно вызывает методы у EJB-компоненты.

Разработчику не нужно самому реализовывать EJB-объект. Этот класс создается специальным кодогенератором, поставляемым вместе в EJB-контейнером. Как уже было сказано, EJB-объект (созданный с помощью сервисов контейнера) и EJB-компонента (созданная разработчиком), реализуют один и тот же интерфейс. В результате, когда приложение-клиент хочет вызвать метод у EJB-компоненты, то сначала вызывается аналогичный (по имени) метод у EJB-объекта, что находится на стороне клиента, а тот, в свою очередь, связывается с удаленной EJB-компонентой и вызывает у нее этот метод (с теми же аргументами).

Объектная компонента:

1) a) home-интерфейс (для нахождения компонента) б) home-объект

2) a) remote-интерфейс (отображает те методы, которые надо показать внешнему миру) б) объект EJB, реализующий remote-интерфейс

3) Enterprise Bean (реализовать интерфейс EnterpriseBean, не реализует никаких авторизаций, многопоточности или транзакций, обеспечивает реализацию бизнес-методов, которые выполняет компонент)

4) Описатель установки и применения (информация о компоненте)

POJO-класс не имплементирует интерфейсы, содержит только геттеры/сеттеры, не может быть наследован. Простой старый интерфейс Java (POJI) - это базовая форма интерфейса Java , приемлемая в тех случаях, когда более сложные интерфейсы Java не допускаются.

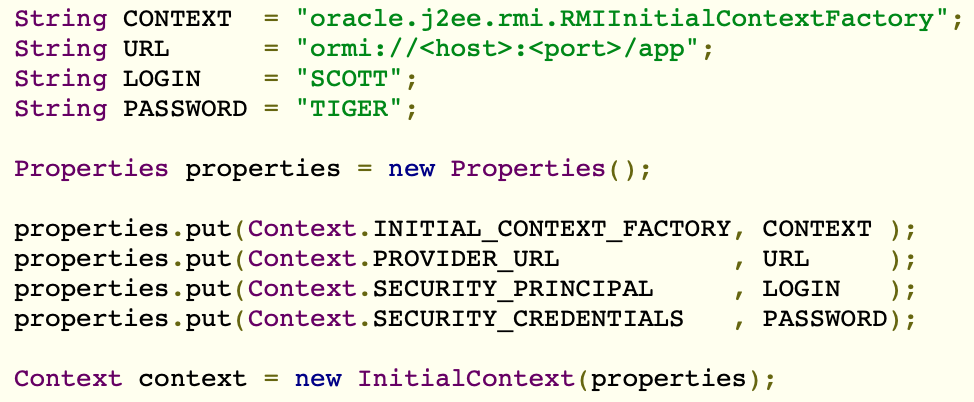
Концепция JNDI основана на двух основных определениях :

- ассоциация (binding) — соответствие JNDI-имени определенному объекту;

- контекст (context) — среда, в которой хранится набор ассоциаций между объектами и именами.

Чтобы воспользоваться хранящимся в контексте JNDI ресурсом, необходимо инициализировать контекст javax.naming.Context и найти требуемый ресурс.

При работе в окружении Java EE (WEB), сервер приложений загружает все необходимые библиотеки для доступа к окружению JNDI. В противном случае необходимо настроить свое приложение, указав, какие библиотеки JNDI оно должно использовать. Сделать это можно, например, создав объект Properties и передав его конструктору InitialContext



Метод lookup интерфейса javax.naming.Context возвращает ресурс с именем name, который следует привести к требуемому типу. Object lookup (String name) Если в аргументе name передать пустую строку, то возвращается новый экземпляр Context.

Формат именования компонентов EJB имеет следующий вид :

java:workspace/[app-name]/module-name/ejb-name[!object-name], где : workspace - пространство имен; app-name - наименование приложения; module-name - наименование модуля; ejb-name - наименование компонента; object-name - полное наименование объекта.

Доступ к файлу возможен также через аннотации:

@EJB — помечается bean, который мы собираемся использовать.

@Stateless — говорит контейнеру, что класс будет stateless session bean. Для него контейнер обеспечит безопасность потоков и менеджмент транзакций.

@Local — относится к интерфейсу и говорит, что bean реализующий интерфейс доступен локально.

@Remote — относится к интерфейсу и говорит, что bean доступен через RMI.

@Stateful — говорит контейнеру, что класс будет stateful session bean.

@Remove — говорит контейнеру, что после его исполнения нет больше смысла хранить bean, т.е. его состояние сбрасывается.

@WebService — говорит, что интерфейс или класс будет представлять web-сервис.

**37. Сущностные и stateless компоненты EJB: понятие, основные механизмы организации взаимодействия с БД. Методы организации работы с компонентами в EJB.**

Entity bean представляет собой компоненту, работающую с постоянной (persistent) информацией, хранящейся, например, в базе данных. Entity beans ассоциируются с элементами баз данных и могут быть доступны одновременно нескольким пользователям. Так как информация в базе данных является постоянной, то и entity beans живут постоянно, ``выживая'', тем самым, после сбоев сервера (когда сервер восстанавливается после сбоя, он может восстановить ``бин'' из базы данных).

Например, entity bean может представлять собой строку какой-нибудь таблицы из базы данных, или даже результат операции SELECT. В объектно-ориентированных базах данных, entity bean может представлять собой отдельный объект, со всеми его атрибутами и связями.

Клиент создает на EJB-сервере объекты и работает с ними так же, как если бы это были локальные объекты. Это до предела упрощает разработку клиентов -- практически нет разницы между написанием клиента для локальной машины и для клиента EJB-сервера. Разработчик может легко создавать, использовать и уничтожать объекты, а эти объекты, в свою очередь, выполняются на сервере. Итак, session beans в большинстве своем работают с информацией, относящейся к ``диалогу'' между клиентом и сервером, в то время как entity beans представляют собой экземпляры данных и работают с ``постоянными'' данными из баз данных.

Каждый экземпляр ``бина'' имеет свой уникальный идентификатор. Для entity объектов уникальный идентификатор, как правило, совпадает с (или каким-либо образом получается из) уникального идентификатора данных, которые он представляет. То есть в данном случае существует нечто вроде первичного ключа для всех entity beans. Идентификатор же session beans может быть практически любым -- например он может состоять из имени сервера и порта удаленного соединения, а может создаваться случайным образом.

EJB тесно связан с двумя спецификациями: JPA, которая является стандартом хранения данных для Java EE и CDI (Contexts and Dependency Injection) которая обеспечивает возможность внедрения зависимостей и предоставляет службы управления контекстом для всех компонентов Java EE, включая EJB. Возможность автоматического сохранения объектов в реляционной БД с использованием технологии объектно-реляционного маппинга (ORM) - так называемый механизм работы с persistence объектами, является одним из главных достоинств EJB. EntityManager API - это интерфейс, который связывает класс сущности приложения (Entity Bean) и её представления в БД. EntityManager знает, как нужно добавлять сущности в БД, обновлять и удалять их, а также предоставляет механизмы для настройки производительности, кэширования, транзакций и т.д. Для этого используется язык запросов JPQL, очень похожий на SQL.

**38. Разработка приложений на основе MOM. Обработка синхронных и асинхронных сообщени JMS**

MOM (Message-Oriented Middleware) - это подход к разработке приложений, который основан на передаче сообщений между компонентами системы. Для создания приложения на основе MOM вам понадобится выбрать подходящий MOM-сервер и использовать его API для отправки и получения сообщений. Некоторые из популярных MOM-серверов, которые вы можете использовать для разработки приложений, включают в себя Apache ActiveMQ, RabbitMQ, IBM MQ и Apache Kafka.

JMS (Java Message Service) - API для отправки и получения сообщений между клиентами. JMS поддерживает как синхронную, так и асинхронную обработку сообщений.

Синхронная обработка сообщений означает, что отправитель ждет ответа от получателя, прежде чем продолжить выполнение своей работы. Для этого отправитель использует синхронный метод, который блокирует выполнение до получения ответа от получателя.

Асинхронная обработка сообщений, с другой стороны, позволяет отправителю продолжить выполнение своей работы, не дожидаясь ответа от получателя. Для этого отправитель использует асинхронный метод, который не блокирует выполнение.

В JMS для синхронной обработки сообщений используется объект типа MessageConsumer, который блокирует выполнение до получения сообщения. Для асинхронной обработки сообщений используется объект типа MessageListener, который регистрируется в объекте типа MessageConsumer и вызывается автоматически при получении сообщения. Асинхронная обработка сообщений обычно используется в случаях, когда отправитель не должен ждать ответа от получателя, например, когда отправитель отправляет сообщение для обновления состояния системы и не нуждается в ответе от получателя.

**39. Основы и концепция JMS. Построение JMS приложения, основные модели и механизмы**

Java Message Service (JMS) – это Java API (то есть набор интерфейсов и классов) для работы с Message-Oriented Middleware (МОМ). Данный набор определен в пакете javax.jms в дереве пакетов J2EE.

В Messaging System приложения общаются не напрямую, а посредством MOM (промежуточного программного обеспечения). Если один компонент системы хочет послать сообщение другому компоненту, он посылает данное сообщение MOM, а уж MOM затем пересылает его адресату.

MOM реализует асинхронность обмена сообщениями.

Существует две "основных" модели обмена сообщениями:

• point-to-point

• publish-subscribe (pub-sub)

Point-to-point модель применяется, когда одному или нескольким компонентам (так называемые senders) необходимо послать сообщение одному компоненту-адресату (receiver).

Publish-subscribe (Pub-sub) модель применима, когда одному или нескольким компонентам (publishers) необходимо послать сообщение одному или нескольким компонентам-адресатам (subscribers). Данная модель основана на понятии message topic.

ConnectionFactory – это обьект, ответственный за создание JMS Connection. Администратор МОМ создает данный обьект и связывает его с деревом JNDI, так что клиент JMS может получить доступ к ConnectionFactory используя стандартный JNDI lookup-механизм.

Connection – абстрактное представление реального соединения между клиентом JMS и MOM.

Session – обьект, создаваемый JMS Connection и используемый клиентами для посылки и принятия сообщений.

Destination – это либо queue, либо topic – в зависимости от используемой модели. Как и ConnectionFactory, destination связывается с деревом JNDI.

MessageProducer – обьект, который, собственно, и посылает сообщения.

MessageConsumer – обьект, принимающий сообщения.

Message– сообщение.

Алгоритм реализации:

• Используем JMS и JNDI пакеты, инициализируем контекст сервиса JNDI;

• достаем ссылку на ConnectionFactory, опираясь на заданное нами имя при создании (развертывании) ресурса;

• создадим Connection (абстакция реального соединения);

• Создадим Session;

• Находим Destination, либо создаем ее;

• Создадим простейшее текстовое сообщение ;

• Создадим MessageProducer

• Активизация связи Connection

• Посылаем сообщение

Существует два пути получения сообщений:

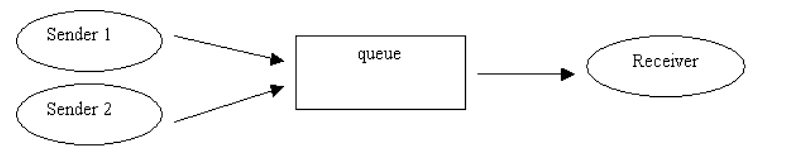
- синхронное затребование сообщений из queue, используя метод receive() интерфейса javax.jms.QueueReceiver.

- асинхронное получение сообщений как только они становятся доступны – используя интерфейс javax.jms.MessageListener.

Особенности использования модели Point - to – Point

Point-to-point модель применяется, когда одному или нескольким компонентам (так называемые senders) необходимо послать сообщение одному компоненту-адресату (receiver).

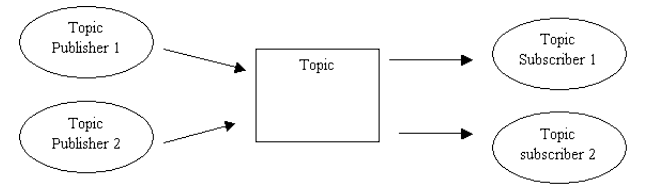
Модель основана на понятии message queue (очередь). Senders посылают сообщения в queue, а Receiver читает сообщения из queue.



Особенности использования модели PUB-SUB

Publish-subscribe (Pub-sub) модель применима, когда одному или несколькимм компонентам (publishers) необходимо послать сообщение одному или неск-м компонентам-адресатам (subscribers). Данная модель основана на понятии message topic(топики). Сам объект Topic предст. незав. от сетевой реализации пункт назначения, в к-й будет отпр-но сообщение. Publishers посылают сообщения в topic, и все subscribers данного topic получают эти сообщения.

Тема представляет аналог списка рассылки: любое приложение, обладающее достаточными правами, может принимать сообщения из темы и посылать сообщения в тему. Когда клиент JMS принимает сообщения из темы, говорят, что клиент подписан (subscribed) на данную тему. JMS разделяет прикладные программы, позволяя им посылать сообщения друг другу через пункт назначения, который работает в качестве виртуального канала.



В Pub-Sub subscriber существует два способа получения сообщения:

• синхронный, с использованием метода receive() интерфейса TopicSubscriber

• асинхронный, с использованием интерфейса MessageListener.

В синхронном варианте, получив (либо создав) Topic, создадим consumer (объект принимающий сообщения): TopicSubscriber subscriber = session.createSubscriber(topic);

После активизации connection, вы можете получать сообщения: TextMessage textMessage = (TextMessage)subscriber.receive();

В случае асинхронного получения сообщений реализуйте метод onMessage интерфейса MessageListener, и зарегистрируйте новый Listener с помощью метода setMessageListener интерфейса TopicSubscriber.

**40. Основные объекты JMS. Порядок действий на клиентской стороне при отправке сообщения. JMS Client. Объекты Connection и ConnectionFactory.**

*! Основные объекты описаны в 39 вопросе.*

На клиентской стороне для отправки сообщения через JMS необходимо выполнить следующие шаги:

1. Создать объект **ConnectionFactory**, который будет использоваться для создания соединения с JMS-провайдером.

2. Создать объект **Connection**, который представляет соединение с JMS-провайдером.

3. Создать объект **Session**, который представляет собой сессию обмена сообщениями между клиентом и JMS-провайдером.

4. Создать объект **Destination**, который представляет собой адресат сообщения (очередь или тему).

5. Создать объект **MessageProducer**, который будет использоваться для отправки сообщения на указанный **Destination**.

6. Создать объект **Message**, который представляет собой отправляемое сообщение.

7. Заполнить объект **Message** данными, которые необходимо отправить.

8. Вызвать метод **MessageProducer.send()**, чтобы отправить сообщение на указанный **Destination**.

9. Закрыть все созданные объекты (**Session, Connection, ConnectionFactory** и т.д.) после завершения работы с ними.

Объект Connection представляет собой соединение между клиентом и JMS-провайдером. Он используется для создания сессий обмена сообщениями между клиентом и JMS-провайдером. Connection может быть настроен для использования различных протоколов и настроек, в зависимости от конкретного JMS-провайдера.

При создании объекта Connection, необходимо указать ConnectionFactory, который будет использоваться для создания соединения. Кроме того, при создании Connection можно указать логин и пароль для аутентификации клиента на JMS-провайдере, а также другие настройки, такие как таймауты и размер буфера.

**41. Роль сессии в контексте соединения JMS. Создание провайдера(продюсера) для обмена сообщений. Модели передачи сообщений.**

В контексте соединения JMS, сессия представляет собой логическое соединение между клиентом и JMS-провайдером. Она используется для обмена сообщениями между клиентом и JMS-провайдером.

Сессия может быть создана с помощью объекта Connection. Она может быть создана как в режиме AUTO\_ACKNOWLEDGE, так и в режиме CLIENT\_ACKNOWLEDGE. В режиме AUTO\_ACKNOWLEDGE, JMS-провайдер автоматически подтверждает получение сообщения после его обработки. В режиме CLIENT\_ACKNOWLEDGE, клиент должен явно подтвердить получение сообщения.

Сессия может быть использована для создания объектов MessageProducer и MessageConsumer, которые используются для отправки и получения сообщений соответственно. Сессия также может быть использована для создания объектов TemporaryQueue и TemporaryTopic, которые используются для временного хранения сообщений.

*! Модели передачи сообщений описаны в 39*

**42. Получение JMS сообщения в режиме извлечения. Виды JMS-сообщений. Создание получателя сообщений. Объект MessageConsumer**

В JMS получение сообщения в режиме извлечения происходит с помощью объекта MessageConsumer, который создается в контексте сессии. После создания MessageConsumer, он может быть использован для получения сообщений из очереди или топика.

Для получения сообщения в режиме извлечения, необходимо вызвать метод receive() объекта MessageConsumer. Этот метод блокирует выполнение текущего потока до тех пор, пока не будет получено сообщение или не истечет время ожидания, заданное при создании объекта Connection.

Если сообщение было успешно получено, метод receive() возвращает объект типа Message. Если сообщение не было получено до истечения времени ожидания, метод возвращает null.

Сообщение в JMS - это объект Java, состоящий из двух частей: заголовка (header), properties и тела (body) сообщения. В заголовке находится служебная информация, тело сообщения содержит в себе пользовательские данные, которые могут быть разной формы: текстовой, сериализуемых объектов, байтовых потоков и т. д.

JMS API определяет несколько типов сообщений:

- BytesMessage предназначен для передачи потока байт, который система никак не интерпретирует;

- MapMessage предназначен для передачи множества элементов типа "имя-значение", где имена являются объектами строкового типа, а значения - объектами примитивных типов данных Java ;

- ObjectMessage предназначен для передачи сериализуемых объектов;

- StreamMessage предназначен для передачи множества элементов примитивных типов данных Java (они могут быть последовательно записаны, а затем прочитаны из тела сообщения этого типа);

- TextMessage предназначен для передачи текстовой информации.

Пример. Получение JMS-сообщения в режиме извлечения

// Создание получателя сообщений

MessageConsumer msgConsumer = session.createConsumer(destination);

conn1.start(); // Запуск соединения

Message msg = msgConsumer.receiveNoWait(); // Попытка получить сообщение

if (msg instanceof TextMessage) { // Проверка наличия текстовых сообщений

TextMessage txtMsg = (TextMessage)msg; // Приведение сообщения к нужному типу

System.out.println(txtMsg.getText()); } // Вывод содержимого сообщения

**43. Использование и назначение MDB. Создание бина, управляемого сообщениями**

MDB (Message-Driven Bean) - компонент EJB, который позволяет приложению получать и обрабатывать сообщения, поступающие в очередь сообщений JMS.

MDB может быть создан и управляться контейнером EJB, что обеспечивает автоматическую обработку сообщений и управление жизненным циклом компонента.

Для создания MDB бина, управляемого сообщениями, необходимо выполнить следующие шаги:

1. Создать класс, который реализует интерфейс javax.jms.MessageListener. В этом классе должен быть реализован метод onMessage(), который будет вызываться при получении нового сообщения.

2. Аннотировать класс аннотацией @MessageDriven, указав параметры, такие как тип очереди или топика, из которых будут получаться сообщения.

3. Разместить класс в EJB контейнере, который будет управлять жизненным циклом бина.

Пример создания MDB:

@MessageDriven(activationConfig = {

@ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),

@ActivationConfigProperty(propertyName = "destination", propertyValue = "myQueue")

})

public class MyMDB implements MessageListener {

public void onMessage(Message message) {

// обработка сообщения

}

}

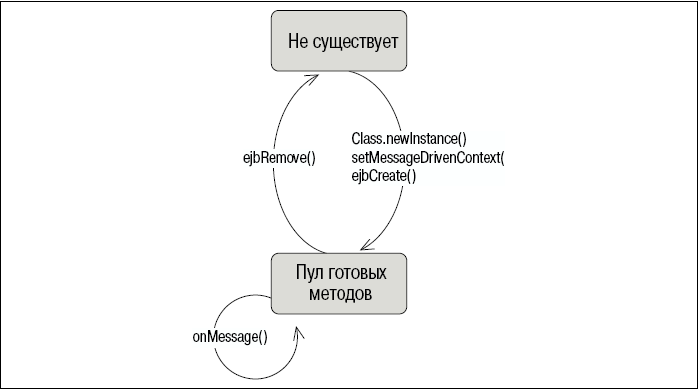
Данный код создает MDB, который будет получать сообщения из очереди "myQueue" и обрабатывать их в методе onMessage(). Приложение может отправлять сообщения в эту очередь, и MDB автоматически получит их и обработает.

**44. MDB – компонента. Реализация интерфейса javax.jms.MessageListener**

Компоненты, управляемые сообщениями (Message Driven Beans, MDB) – не имеющие состояния, серверные компоненты для обработки асинхронных сообщений JMS, поддерживающие транзакции.

Один из наиболее важных аспектов компонентов, управляемых сообщениями, состоит в том, что они могут получать и обрабатывать сообщения параллельно. Эта возможность дает им значительное преимущество перед обычными клиентами JMS, в которых управление ресурсами, транзакциями и безопасностью в многопоточном окружении должно выполняться самим клиентом.

Жизненный цикл экземпляра MDB имеет два состояния: «не существует» и «пул готовых методов». Пул готовых методов похож на пул экземпляров, используемый для сеансовых компонентов без состояния. Как и для компонентов stateless, для жизненных циклов MDB определен пул экземпляров.

Не существует. Когда экземпляр MDB находится в состоянии «не существует», он не является экземпляром в памяти системы. Другими словами, он еще не создан.

Пул готовых методов Экземпляры MDB переходят в пул готовых методов по мере того, как они стан-тся нужны Конт-ру.

Переход в пул готовых методов Когда экземпляр перемещается из состояния «не существует» в пул готовых методов, над ним выполняются три действия. Сначала посредством метода Class.newInstance() класса MDB создается экземпляр компонента. Далее контейнер вызывает метод setMessageDrivenContext(), предоставляя экземпляру MDB ссылку на его EJBContext. Ссылка на MessageDrivenContext может быть сохранена в поле экземпляра MDB. Наконец, контейнером вызывается безаргументный метод ejbCreate() экземпляра компонента. У MDB есть только один метод ejbCreate(), не принимающий никаких параметров. Метод ejbCreate() вызывается только один раз в течение жизненного цикла MDB. MDB не являются объектами, подлежащими активации, поэтому они могут поддерживать открытые соединения с ресурсами на всем протяжении их жизненного цикла. Метод ejbRemove() должен закрывать все открытые ресурсы прежде, чем MDB будет удален из памяти в конце своего жизненного цикла.

Жизнь в пуле готовых методов Если экземпляр находится в пуле готовых методов, он готов к обработке входящих сообщений. Сообщение, поступающее к MDB, перенаправляется любому доступному экземпляру, находящемуся в пуле готовых методов. Пока экземпляр выполняет запрос, он не может обрабатывать другие сообщения. MDB может одновременно обрабатывать несколько сообщений, делегируя обязанность по обработке каждого сообщения другому экземпляру MDB. Когда сообщение направлено экземпляру контейнером, MessageDrivenContext экземпляра MDB изменяется, чтобы отразить новый контекст транзакции. Экземпляр, закончивший обработку, сразу же становится доступным для обработки нового сообщения.

Переход из пула готовых методов: смерть экземпляра MDB Экземпляры компонентов переводятся из пула готовых методов в состояние «не существует», когда они становятся ненужными серверу. Это происходит, когда сервер принимает решение уменьшить общий размер пула готовых методов, удаляя из памяти один или несколько экземпляров. Этот процесс нач. с вызова метода ejbRemove() экземпляра.

Вот пример реализации интерфейса javax.jms.MessageListener:

import javax.jms.Message;

import javax.jms.MessageListener;

public class MyMessageListener implements MessageListener {

@Override

public void onMessage(Message message) {

try {

// Обработка полученного сообщения

System.out.println("Получено сообщение: " + message.getBody(String.class));

} catch (JMSException e) {

e.printStackTrace();

}

}

}

В этом примере создается класс MyMessageListener, который реализует интерфейс MessageListener. Метод onMessage вызывается при получении нового сообщения из очереди или топика JMS. Внутри метода onMessage можно написать код для обработки полученного сообщения. В данном примере, мы выводим текст сообщения в консоль. Обратите внимание, что метод getBody используется для извлечения содержимого сообщения в виде строки. Если сообщение содержит другой тип данных, необходимо использовать соответствующий метод для извлечения данных.

**45. Технология JSF. Компоненты ввода. Создание пользовательских компонентов: роль JSF компонентов, ЖЦ, обработчики, инструменты отображения, структура JSF компонента**

JSF (Java Server Faces) - это фреймворк для создания пользовательских интерфейсов веб-приложений на языке Java. Он предоставляет множество компонентов пользовательского интерфейса, включая компоненты ввода, такие как текстовые поля, выпадающие списки, кнопки и т.д.

Роль JSF компонентов заключается в предоставлении пользовательского интерфейса и обработке пользовательского ввода.

Жизненный цикл JSF компонента включает несколько этапов, таких как создание, инициализация, обработка событий, обновление и отображение. На каждом этапе компонент может выполнять свои собственные действия, например, инициализировать свои свойства или обрабатывать события, которые произошли в пользовательском интерфейсе.

Обработчики JSF компонентов используются для обработки событий, которые происходят в пользовательском интерфейсе (щелчок на кнопке или изменение значения в текстовом поле).

Инструменты отображения JSF компонента определяют, как компонент будет отображаться в пользовательском интерфейсе. Это может быть HTML, CSS или любой другой язык разметки, который поддерживается браузером. Структура JSF компонента включает в себя несколько основных элементов, таких как файлы конфигурации, файлы ресурсов, классы обработчиков и файлы отображения. Все эти элементы объединяются в единый компонент, который может быть использован в различных частях приложения.

**46. Стратегии обработки асинхронных запросов в контексте ЖЦ JSF. Визуальное построение интерфейса с использованием компонентов JSF.**

Стратегия обработки асинхронных запросов в контексте жизненного цикла JSF включает в себя использование AJAX (Asynchronous JavaScript and XML) для отправки запросов на сервер без перезагрузки страницы. Когда пользователь выполняет действие в пользовательском интерфейсе, например, щелчок на кнопке, JSF отправляет AJAX-запрос на сервер, который обрабатывает запрос и возвращает результат в формате XML или JSON. Затем JSF использует этот результат для обновления соответствующих частей пользовательского интерфейса без перезагрузки всей страницы.

В контексте жизненного цикла JSF, обработка асинхронных запросов происходит на этапах обработки событий и обновления компонентов. При получении AJAX-запроса, JSF вызывает соответствующую обработчик события, который обрабатывает запрос и возвращает результат. Затем JSF использует этот результат для обновления компонентов, связанных с запросом. Если в результате обработки AJAX-запроса происходит изменение состояния компонента, JSF сохраняет это состояние в контексте запроса, чтобы оно могло быть использовано в последующих запросах. Это позволяет сохранять состояние пользовательского интерфейса между запросами, что особенно важно для приложений с большим количеством интерактивных элементов.

Пример JSF файла

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:h="http://java.sun.com/jsf/html"

xmlns:ui="http://java.sun.com/jsf/facelets">

<h:head>

<title>Успешная авторизация</title>

</h:head>

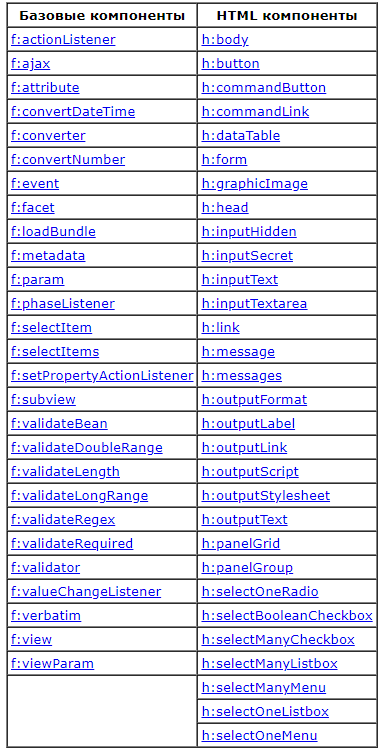
<h:body>

<h:outputText value="Добро пожаловать: #{Main.login}!" />

</h:body>

</html>

**Компоненты JSF:**

****

К базовым относятся компоненты JSF, которые не связаны с рендерингом страницы HTML или с каким-либо другим механизмом отображения информации. Среди прочего они используются для преобразования типов и проверка допустимости значений.

**f:actionListener**

Тег определяет выполнение метода действия слушателя processAction(), определяемый атрибутом type тега. Значение атрибута type должно быть полностью определенным (квалифицированным) именем класса, реализующего интерфейс javax.faces.event.ActionListener. Этот тег обычно является дочерним тегом h:commandButton или h:commandLink. Когда пользователь щелкает по родительскому компоненту, метод processAction() объявленной реализации ActionListener выполняется автоматически.

<h:commandButton action="save" value="Сохранить">

<f:actionListener type="com.test.CustomActionListener" />

</h:commandButton>

**f:attribute**

Тег устанавливает атрибут родительского компонента с ключом, определенным атрибутом name тега и значением, определенным атрибутом value тега. Все атрибуты компонента позже могут быть получены программно, как структура данных — карта Map, вызовом метода getAttributes() соответствующего экземпляра javax.faces.component.UIComponent. Этот тег часто используется в сочетании с классом f:actionListener, чтобы передать параметру методу действия слушателя.

<h:commandButton action="save" value="Save">

<f:actionListener type="com.test.CustomActionListener" />

<f:attribute name="attributeName" value="someValue" />

</h:commandButton>

**47. Facelets - как основа технологии управления представлением для Java Server Faces (JSF) и альтернатива ранее используемой в JSF технологии Java Server Pages (JSP).**

JavaServer Faces (JSF) – это фреймворк для разработки веб-приложений на Java EE, который предоставляет набор компонентов пользовательского интерфейса, событий и обработчиков для создания динамических веб-страниц и приложений.

Преимущества JSF 2 по сравнению MVC (servlet + jsp):

* JSF предоставляет набор API, который позволяет создавать свои компоненты
* С JSF легче указать, благодаря какому коду будет обрабатываться форма
* JSF расширяет/добавляет возможность валидировать/конвертировать значения
* JSF помогает НЕ использовать JavaScript, формы создаются благодаря компонентам
* JSF помогает валидировать поля форм
* JSF поддерживает шаблонизацию страниц
* Позволяет выносить данные в конфигурационный файл

Недостатки JSF 2 по сравнению MVC (servlet + jsp):

* меньше документации на JSF
* тяжело оптимизировать и отследить определенные компоненты JSF
* на некоторых IDE не работает JSF, так как JSF это новая технология (2009 год выпуска)

Теги в Facelets:

* <ui:define> - используется для определения области содержимого на странице, которая может быть заполнена дочерними шаблонами.
* <ui:include> - используется для включения содержимого из других файлов шаблонов.
* <h:outputText> - используется для отображения текстового содержимого.
* <h:form> - используется для создания формы на странице.
* <h:inputText> - используется для создания текстового поля на странице.
* <h:commandButton> - используется для создания кнопки на странице, которая может отправлять форму на сервер.
* <h:dataTable> - используется для создания таблицы на странице и отображения данных из источника данных, такого как база данных.
* <ui:composition> - используется для определения шаблона страницы и включения других компонентов внутри шаблона.

**48. Интеграция технологии AJAX в Java Enterprise приложение. Динамическая подгрузка данных на страницу. Использование Java Query API.**

Ajax (Asynchronous JavaScript and XML) – это технология, которая позволяет обновлять части веб-страницы без перезагрузки всей страницы. Ajax может использоваться для улучшения пользовательского интерфейса и оптимизации производительности веб-приложений.

Ajax-запрос (Asynchronous JavaScript and XML request) – это технология обмена данными между сервером и клиентом без перезагрузки всей страницы.

Ajax можно интегрировать с помощью технологий и фреймворков:

* JavaServer Faces (JSF): JSF имеет встроенную поддержку Ajax через библиотеку компонентов Ajax-enabled.
* Spring Framework: Spring имеет модуль Spring Web MVC, который предоставляет поддержку Ajax через библиотеку компонентов Spring Web Flow.
* jQuery: jQuery является одним из самых популярных JavaScript фреймворков, который предоставляет множество методов для работы с Ajax.

Основные события из пользовательского интерфейса:

* ActionEvent – вызывает компонент (кнопка или ссылка), активированный пользователем
* ValueChangeEvent – вызывает компонент (check-box, box/radio button, list box/combo), чье состояние было изменено
* AjaxBehaviorEvent – вызывает ajax компонент

Динамическая подгрузка данных на страницу – это процесс загрузки данных с сервера и отображения их на веб-странице без перезагрузки страницы.

JQuery – это библиотека JavaScript, которая облегчает работу с DOM-деревом и упрощает разработку динамических веб-страниц.

С помощью JQuery нужно выполнить следующее для динамической подгрузки данных на страницу:

* Создать элемент HTML, который будет отображать данные, например, таблицу или список.
* Создать скрипт JavaScript с использованием JQuery, который будет получать данные с сервера и обновлять элемент HTML на странице.

- Связать скрипт JavaScript с элементом HTML на странице.

**49. Библиотеки и компоненты JSF. Возможности библиотеки визуальных компонентов RichFaces в качестве Faces компонентов для JavaServer.**

Базовые компоненты JSF:

* f:actionListener – тег определяет выполнение метода действия слушателя processAction(), определяемый атрибутом type тега.
* f:ajax – тег определяет поведение AJAX. Этот тег обычно вкладывается в другой тег JSF типа h:commandButton.
* f:attribute – тег устанавливает атрибут родительского компонента с ключом, определенным атрибутом name тега и значением, определенным атрибутом value тега
* f:convertDateTime – тег используется для преобразования значения родительского компонента в экземпляр java.util.Date. Этот тег позволяет присваивать правильно отформатированную строку, вводимую пользователем, полю даты в управляемом бине.
* f:converter – регистрация преобразователя пользователя, указанныго атрибутом converterId тега, в родительском теге.
* f:convertNumber – преобразование значения родительского компонента в экземпляр java.lang.Number. Данный тег позволяет правильно отформатировать строку, вводимую пользователем и присваиваемую числовому полю в управляемом бине.
* f:event – тег вызова метода компонента JavaBean при возникновении конкретного события
* f:metadata – данный тег является родительским для тега f:viewParam, который используется для отображения параметров GET-запроса на значения управляемого бина.
* f:param – тег используется как дочерний с тегами h:commandLink и h:outputForm. Для тега h:commandLink, он генерирует параметр запроса, определенный его атрибутами name и value. Для тега h:outputForm он определяет подстановочное значение атрибута value.
* f:selectItem – тег используется в качестве дочернего компонента для тегов h:selectManyCheckBox, h:selectManyListBox, h:selectManyMenu, h:selectOneListBox, h:selectOneMenu и h:selectOneRadio. Добавляет элемент выбора родительскому компоненту.
* f:selectItems – добавляет набор выбираемых элементов родительскому компоненту. f:subview – предназначен для использования JSP в качестве представления. Включение JSP-страницы выполняется с помощью тега jsp:include или JSTL-тега c:import
* f:validateLength – Проверка введеной строки, чтобы размера находился в диапазоне между значениями, определенными атрибутами minimum и maximum.
* f:validator – Проверка значения родительского компонента с использованием компонента JavaBean верификации, реализующего интерфейс javax.faces.validator.Validator. Компонент проверки должен быть или аннотирован @FacesValidator, или обьявлен в дескрипторе конфигурации приложения - файл faces-config.xml.
* f:viewParam – тег используется для установления связи параметра HTTP-запроса GET со свойством управляемого бина.

Библиотеки и компоненты JSF:

* PrimeFaces - библиотека компонентов пользовательского интерфейса, которая содержит более 100 готовых компонентов, включая таблицы, формы, диаграммы и многое другое.
* RichFaces - набор компонентов пользовательского интерфейса, который предоставляет богатые возможности для создания интерактивных веб-приложений.
* Apache MyFaces - реализация JSF, которая включает в себя множество компонентов и библиотек для создания веб-приложений.
* ICEfaces - набор компонентов пользовательского интерфейса, который предоставляет возможности для создания богатых интерактивных приложений с использованием AJAX.
* BootsFaces - библиотека компонентов пользовательского интерфейса, которая предоставляет компоненты, построенные на основе Bootstrap.

Компоненты RichFaces включают в себя:

* Календарь - позволяет пользователю выбирать дату из календаря.
* Таблица - позволяет отображать данные в виде таблицы с возможностью сортировки, фильтрации и пагинации.
* Дерево - позволяет отображать иерархические данные в виде дерева.
* Панель - предоставляет пользовательский интерфейс для создания вкладок, панелей и других контейнеров.
* Графики - предоставляет возможность создания различных типов графиков.
* Всплывающие окна - предоставляет возможность отображения всплывающих окон с различным содержимым, таким как формы или диалоговые окна.
* Drag and drop - позволяет пользователю перетаскивать элементы на странице.
* Автодополнение - предоставляет пользовательский интерфейс для автоматического завершения ввода текста на основе ранее введенных значений.

RichFaces использует Ajax для обновления частей страницы без перезагрузки страницы.

Архитектура RichFaces:

* Ajax Filter – чтобы получить все преимущества использования RichFaces разработчику необходимо зарегистрировать фильтр в файле web.xml приложения. Фильтр распознаёт различные типы запросов.
* AJAX Action Components – AjaxCommandButton, AjaxCommandLink, AjaxPoll, AjaxSupport и другие компоненты могут использоваться для отправки Ajax запросов с клиентской стороны.
* AJAX Containers (AjaxContainer) – это интерфейс, который описывает область на странице JSF, которая должна быть декодирована во время запроса Ajax. AjaxViewRoot и AjaxRegion представляют собой реализации этого интерфейса.
* JavaScript Engine – RichFaces JavaScript-движок работает на стороне клиента. Он обновляет различные области на странице JSF на основе информации полученной из Ajax ответа. JavaScript-движок предоставляет API, поэтому разработчику не нужно создавать собственные функции JavaScript.

**50. Перспективные технологии разработки веб-ориентированных систем. Создание многоуровневого распределения Java EE совместимого приложения с веб-приложением.**

Технологии разработки должны включать в себя:

* Системы управления базами данных: СУБД, они позволяют эффективно хранить и обрабатывать данные в распределенной среде. Они предлагают высокую доступность, масштабируемость и отказоустойчивость, что делает их привлекательными для использования в многоуровневых распределенных приложениях.
* Фреймворки для разработки веб-приложений: фреймворки, такие как React, Angular и Vue.js, облегчают создание интерактивных веб-приложений с использованием JavaScript и HTML/CSS. Они обеспечивают высокую производительность и масштабируемость веб-приложений.

Создание многоуровневого приложения (можно использовать JMS, EJB, JPA):

* Определение архитектуры приложения: необходимо определить, какие уровни архитектуры будут использоваться, например, клиентский уровень, бизнес-логика и уровень данных.
* Разработка веб-интерфейса: создание пользовательского интерфейса с помощью фреймворков и технологий веб-разработки, таких как HTML, CSS и JavaScript.
* Разработка бизнес-логики: написание кода, который будет обрабатывать запросы от веб-интерфейса и выполнять необходимые операции на уровне бизнес-логики.
* Разработка уровня данных: создание базы данных или интеграция с уже существующей базой данных для хранения и извлечения данных.
* Развертывание приложения: после написания кода и тестирования его на локальной машине необходимо развернуть приложение на сервере или в контейнере. Для этого можно использовать инструменты автоматизации развертывания, такие как Docker.
* Обеспечение безопасности: необходимо убедиться, что приложение защищено от внешних атак. Для этого можно использовать механизмы аутентификации и авторизации.
* Тестирование приложения: проведение тестов приложения на всех уровнях архитектуры для обеспечения высокой производительности, масштабируемости и отказоустойчивости.
* Обслуживание и поддержка: необходимо обеспечить регулярное обслуживание и поддержку приложения, включая обновление его до новых версий и исправление ошибок.

**51. J2EE как стандарт API для разработки многоуровневых распределенных систем. Организация компонентного подхода с возможностью повторного использования компонентов.**

Java 2 Enterprise Edition (J2EE) – стандарт API для разработки многоуровневых распределенных систем.

Компонентный подход в J2EE представляет собой методологию, которая разбивает приложение на независимые компоненты, каждый из которых выполняет определенную функцию. Эти компоненты могут быть разработаны, тестированы и собраны отдельно, а затем объединены в одно приложение.

Эти компоненты могут включать:

* Сервлеты (компоненты, которые выполняют функции на стороне сервера и предназначены для обработки HTTP-запросов от клиента)
* JSP-страницы (компоненты, которые позволяют создавать динамические веб-страницы с помощью языка разметки HTML и технологии JavaServer Pages)
* EJB-компоненты (компоненты, которые предоставляют бизнес-логику приложения и обеспечивают транзакционную обработку данных)
* JMS-сообщения (компоненты, которые обеспечивают асинхронную обработку сообщений между компонентами)
* Другие компоненты

Организация компонентного подхода с возможностью повторного использования компонентов в J2EE достигается за счет использования контейнеров компонентов.

Контейнер компонентов – среда выполнения для компонентов J2EE.

Функции контейнера компонентов:

* Управление жизненным циклом компонентов: контейнер компонентов отвечает за создание, инициализацию, активацию, деактивацию и уничтожение компонентов.
* Обеспечение безопасности: контейнер компонентов обеспечивает безопасность приложения путем авторизации и аутентификации пользователей, контроля доступа к ресурсам.
* Транзакционная обработка данных: контейнер компонентов обеспечивает целостность данных и предотвращает их потерю.
* Управление ресурсами: контейнер компонентов управляет доступом к ресурсам, таким как базы данных, пулы соединений и другие.
* Обеспечение масштабируемости и производительности: контейнер компонентов оптимизация использования ресурсов.

**52. Распределенные многоуровневые системы. Абстракции Layers, Tiers, Capabilities. Гетерогенность, безопасность, масштабируемость, производительность, параллелизм для характеристики информационных систем.**

Распределенные многоуровневые системы – это информационные системы, которые состоят из нескольких компонентов, работающих на разных уровнях или слоях, каждый из которых выполняет свои функции и взаимодействует с другими уровнями.

Abstraction Layer – слой ИС, реализованный в ПО, находящийся между физическим уровнем аппаратного обеспечения и программным обеспечением, запускаемом на этом компьютере. AL предназначен для скрытия различий в аппаратном обеспечении от основной части ядра операционной системы, таким образом, чтобы большая часть кода, работающая в режиме ядра, не нуждалась в изменении при её запуске на системах с различным аппаратным обеспечением.

Виды слоев:

* Сетевой слой (Network layer) – обеспечивает передачу данных между устройствами на сетевом уровне (протоколы IP, ICMP, ARP).
* Транспортный слой (Transport layer) – обеспечивает надежную передачу данных между устройствами на транспортном уровне (протоколы TCP, UDP).
* Прикладной слой (Application layer) – обеспечивает взаимодействие пользователя с приложением (протоколы HTTP, SMTP, FTP).
* Слой данных (Data layer) – отвечает за хранение данных и доступ к ним. На этом слое может работать база данных или другие механизмы хранения данных.

Абстракция Tiers – уровень ИС, который используется для логического разделения распределенной системы на компоненты, которые выполняют определенные функции и взаимодействуют друг с другом для достижения общей цели.

В распределенных системах используется три уровня:

* Уровень представления (Presentation tier) – обеспечивает взаимодействие пользователя с системой и представление данных в удобной форме (может быть в качестве интерфейса пользователя или веб-страницы)
* Уровень приложений (Application tier) – выполняет логическую обработку данных и бизнес-логику системы. На этом уровне могут работать сервисы и приложения, которые обрабатывают запросы пользователя и обеспечивают доступ к данным и функциональности системы.
* Уровень данных (Data tier) – отвечает за хранение данных и доступ к ним. На этом уровне может работать БД.

Абстракция Capabilities – возможности ИС, используются для описания прав и возможностей, которые имеют различные компоненты системы при выполнении определенных задач.

Возможности в ИС:

* Чтение данных - возможность получать данные из системы.
* Запись данных - возможность записывать данные в систему.
* Удаление данных - возможность удалять данные из системы.
* Аутентификация - возможность проверки подлинности пользователей системы.
* Авторизация - возможность контроля доступа к определенным ресурсам системы.
* Шифрование - возможность защиты данных от несанкционированного доступа.

При разработке и использовании распределенных многоуровневых систем необходимо учитывать ряд факторов, включая:

* Гетерогенность - система может содержать компоненты различных типов, которые работают на различных платформах и с различными технологиями.
* Безопасность - в распределенной системе могут быть уязвимости, связанные с защитой данных и контролем доступа.
* Масштабируемость - система должна быть способна масштабироваться для поддержки большого количества пользователей и обработки большого объема данных.
* Производительность - система должна быть способна обеспечивать быструю обработку запросов и высокую скорость передачи данных между компонентами.
* Параллелизм - система должна поддерживать параллельную обработку данных, чтобы повысить производительность и масштабируемость.