Практическая работа №4

Разработка клиент-серверного приложения с использованием технологии Websocket и Spring Framework

Цель работы:

Краткие теоретические сведения:

Websocket позволяет создать канал связи между клиентом и сервером. В частности, канал связи, который использует протокол WebSocket в качестве протокола связи. Протокол WebSocket совместим с протоколом HTTP. который также работает через TCP / IP . Однако он имеет исключительные улучшения, в части меньших накладных расходов, чем HTTP, и двунаправленную веб-связь. Таким образом, они в первую очередь предназначены для веб-приложений, которым требуется постоянное соединение с сервером. Связь WebSocket может использоваться **между любыми типами приложений** , но чаще всего WebSocket используется для облегчения связи между серверным приложением и приложением клиентом на основе браузера.

На рисунке 4.1 показано различие между «традиционным» способом связи с использованием HTTP и связи с использованием WebSockets.

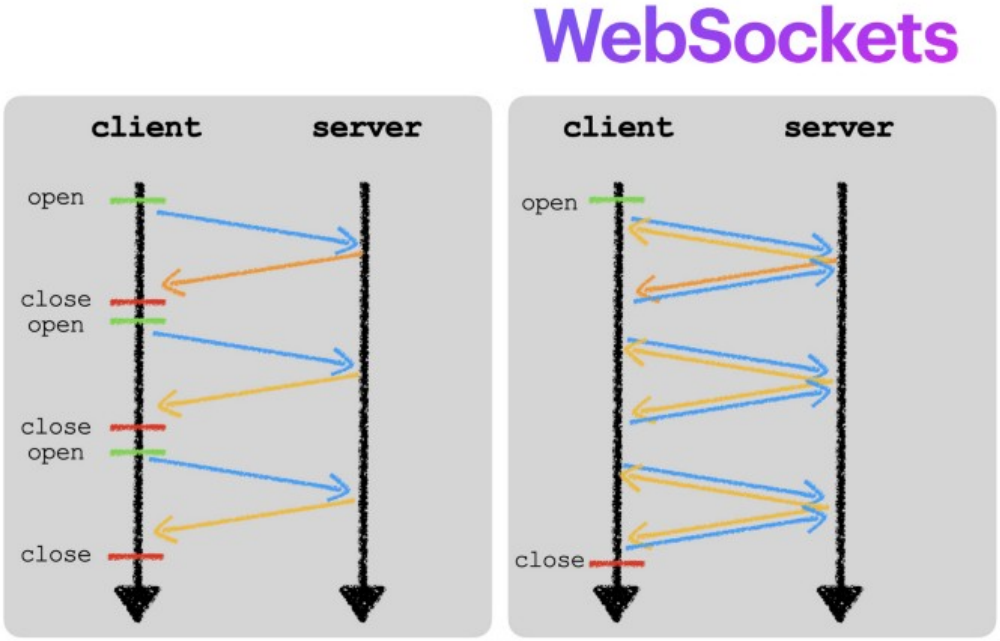


Рисунок 4.1 - Различие между «традиционным» (HTTP-опрос) способом связи с использованием HTTP и связи с использованием WebSockets

Традиционное веб-общение с использованием протокола HTTP работает следующим образом:

- во-первых, клиент (обычно веб-браузер) должен подключиться к серверу;
- затем клиент отправляет запрос ресурса (например, запрос веб-страницы);
- после этого сервер отвечает;
- закрывает канал связи.

С другой стороны, WebSocket работает следующим образом:

- держит соединение открытым;
- кроме того и клиент, и сервер могут делать запросы и отправлять ответы.

Протокол WebSocket призван заменить существующие обходные механизмы HTTP и предоставить эффективный протокол для одновременной двунаправленной связи с малой задержкой между браузерами и серверами по одному TCP-соединению. HTTP изначально был разработан для передачи ресурсов типа "запрос-ответ" в распределенных гипермедийных системах, но не для одновременной двунаправленной связи. Для преодоления этих архитектурных ограничений используются несколько HTTP-механизмов (сгруппированных под неофициальным названием Comet), которые часто бывают сложными и неэффективными.

В этой работе кратко описываются отношения между WebSocket и HTTP / 1.1.

Поскольку HTTP не был разработан для поддержки сообщений, инициируемых сервером, для этого **было разработано несколько механизмов**, каждый из которых имеет свои преимущества и недостатки.

HTTP-опрос

Во время механизма опроса клиент отправляет периодические запросы серверу, и сервер немедленно отвечает. Если есть новые данные, сервер возвращает их, в противном случае сервер возвращает пустой ответ. После получения ответа клиент некоторое время ждет, прежде чем отправить другой запрос.

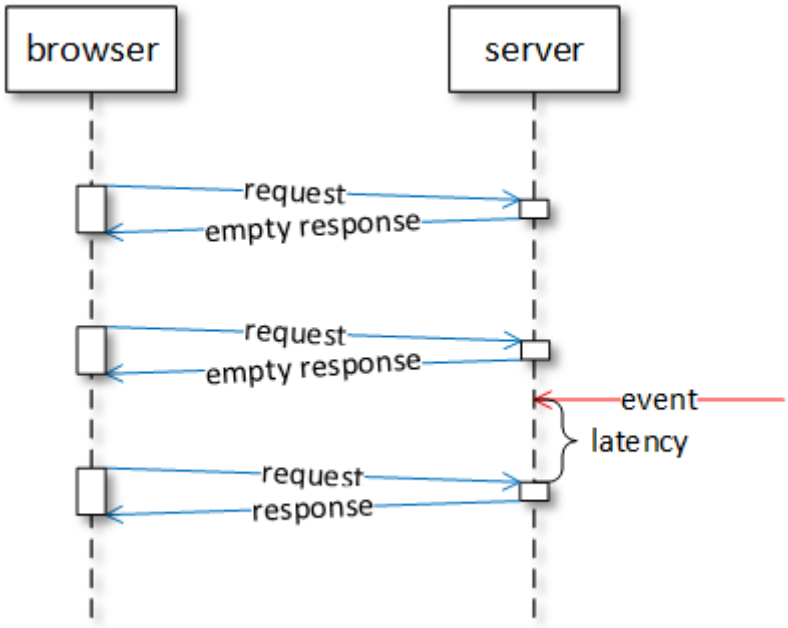


Рисунок 4.2 - Механизма опроса сервера

Опрос может быть эффективным, если мы знаем период обновления данных на сервере. В противном случае клиент может опрашивать сервер либо слишком редко (добавляя дополнительную задержку при передаче данных от сервера к клиенту), либо слишком часто (тратя впустую обработку сервера и сетевые ресурсы).

HTTP длинный опрос

Во время длительного механизма опроса клиент отправляет запрос на сервер и начинает ждать ответа. Сервер не отвечает, пока не появятся новые данные или не истечет время ожидания. Когда новые данные становятся доступными, сервер отправляет ответ клиенту. Получив ответ, клиент сразу же отправляет другой запрос.

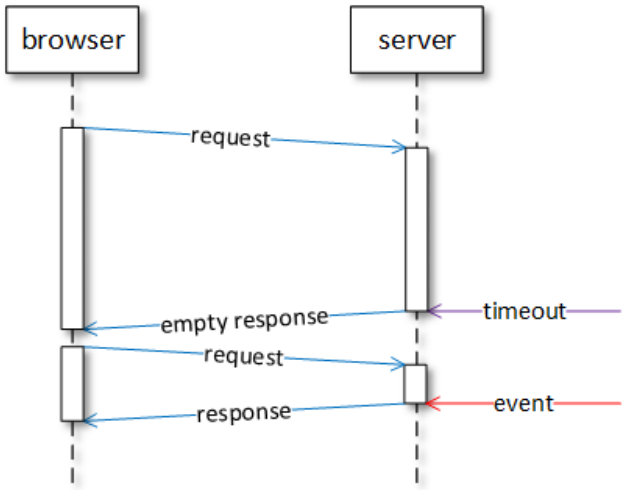


Рисунок 4.2 - Длительный опрос

Длительный опрос снижает использование серверной обработки и сетевых ресурсов для получения обновлений данных с малой задержкой, особенно когда новые данные становятся доступными с нерегулярными интервалами. Однако сервер должен отслеживать несколько открытых запросов. Кроме того, время ожидания длительных запросов может истекать, и новые запросы необходимо отправлять периодически, даже если данные не обновляются.

HTTP потоковая передача

Во время механизма потоковой передачи клиент отправляет запрос на сервер и держит его открытым неопределенное время. Сервер не отвечает, пока не поступят новые данные. Когда новые данные становятся доступными, сервер отправляет их обратно клиенту как часть ответа. Данные, отправленные сервером, не закрывают запрос.

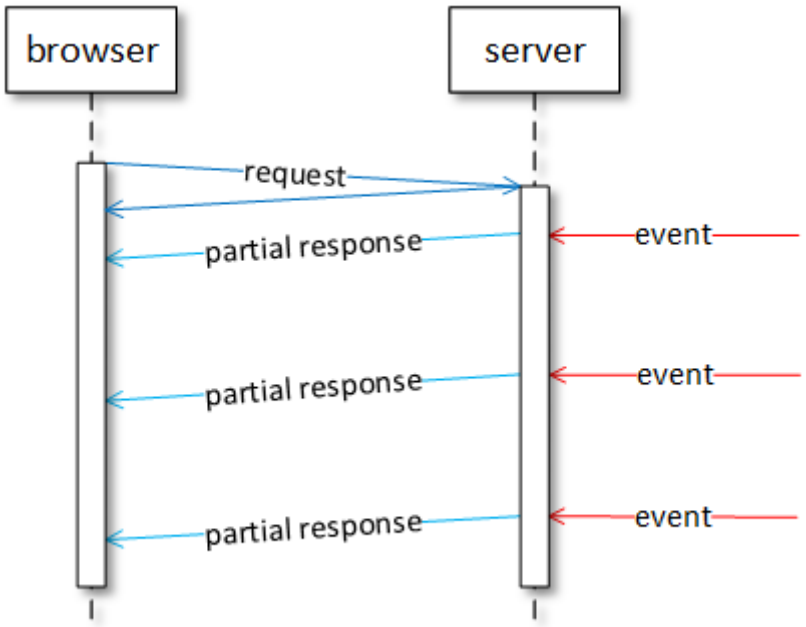


Рисунок 4.2 - Механизм потоковой передачи

Потоковая передача основана на способности сервера отправлять несколько фрагментов данных в одном ответе, не закрывая запрос. Этот механизм значительно снижает сетевую задержку, поскольку клиенту и серверу не нужно отправлять и получать новые запросы.

Однако клиент и сервер должны договориться о том, как интерпретировать поток ответа, чтобы клиент знал, где заканчивается один фрагмент данных и начинается другой. Кроме того, сетевые посредники могут нарушить потоковую передачу - они могут буферизовать ответ и вызывать задержку или отключать соединения, которые остаются открытыми в течение длительного времени.

Отправленные сервером события, как один из механизмов потоковой передачи

Отправленные сервером события (SSE) - это стандартизированный механизм потоковой передачи, который имеет сетевой протокол и API EventSource для браузеров. SSE определяет однонаправленный поток событий в кодировке UTF-8 от сервера к браузеру. События имеют обязательные значения и могут иметь необязательные типы и уникальные идентификаторы. В случае сбоя SSE поддерживает автоматическое переподключение клиента с момента последнего полученного события.

Пример запроса SSE:

```
GET /sse HTTP/1.1
Host: server.com
Accept: text/event-stream
```

Пример SSE ответа:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/event-stream
Transfer-Encoding: chunked

retry: 1000

data: A text message
data: {"message": "a JSON message"}

event: text
data: A message of type 'text'

id: 1
event: text
data: A message of type 'text' with a unique identifier

:ping
```

События, отправленные сервером, могут отправлять потоковые данные только с сервера в браузер и поддерживают только текстовые данные.

Как было уже сказано выше, WebSocket разработан для преодоления ограничений механизмов на основе HTTP (опрос, длинный опрос,

потоковая передача) в *полнодуплексном* взаимодействии между браузерами и серверами:

- При *полнодуплексной* связи обе стороны могут одновременно отправлять и получать сообщения в обоих направлениях.
- При *полудуплексной* связи обе стороны могут отправлять и получать сообщения в обоих направлениях, но одновременно в одном направлении.

Вот основные различия между HTTP и WebSocket:

- HTTP - это текстовый протокол, WebSocket - это двоичный протокол (двоичные протоколы передают по сети меньше данных, чем текстовые протоколы)
- HTTP имеет заголовки запроса и ответа, сообщения WebSocket могут иметь формат, подходящий для конкретных приложений (ненужные метаданные не передаются по сети)
- HTTP - полудуплексный протокол, WebSocket - полнодуплексный протокол (сообщения с малой задержкой могут передаваться одновременно в обоих направлениях)

Архитектурные особенности Websocket

WebSocket - это протокол, который позволяет одновременную двунаправленную передачу текстовых и двоичных сообщений между клиентами (в основном браузерами) и серверами по одному TCP-соединению. WebSocket может обмениваться данными через TCP на порт 80 (схема «ws») или через TLS / TCP на порт 443 (схема «wss»).

WebSocket разработан для добавления поддержки сокетов TCP с минимальными изменениями связи браузера с сервером, обеспечивая необходимые ограничения безопасности в Интернете. WebSocket добавляет минимальную функциональность поверх TCP, не более чем следующее:

- модель безопасности на основе происхождения
- преобразование IP-адресов, используемых в TCP, в URL-адреса, используемые в Интернете
- протокол сообщений поверх протокола байтового потока
- заключительное рукопожатие

Протокол WebSocket разработан как простой протокол и обеспечивает основу для построения на его основе подпротоколов приложений, подобно

тому, как протокол TCP позволяет создавать протоколы приложений (HTTP, FTP, SMTP, POP3, Telnet и т. Д.).

Стандарт WebSocket состоит из двух частей: протокола WebSocket, стандартизованного как RFC 6455, и API WebSocket .

Протокол WebSocket

Сетевой протокол WebSocket состоит из двух компонентов:

- открывающее рукопожатие для согласования параметров соединения WebSocket
- формирование бинарных сообщений для отправки текстовых и бинарных сообщений

Открытие соединения

Перед началом обмена сообщениями клиент и сервер согласовывают параметры устанавливаемого соединения. WebSocket повторно использует существующий механизм обновления HTTP с настраиваемыми заголовками Sec-WebSocket- * для согласования подключения.

Подпротоколы WebSocket - это протоколы верхнего уровня, которые обеспечивают дополнительную функциональность для приложений (например, подпротокол STOMP обеспечивает модель обмена сообщениями публикация-подписка).

Расширения WebSocket - это механизм для изменения фреймов сообщений, не затрагивая протоколы приложений. (например, расширение permessage-deflate сжимает данные полезной нагрузки с помощью алгоритма LZ77).

Пример запроса обновления HTTP на WebSocket:

```
GET /socket HTTP/1.1
Host: server.com
Connection: Upgrade
Upgrade: websocket
Origin: http://example.com
Sec-WebSocket-Version: 8, 13
Sec-WebSocket-Key: 7c0RT+Z1px24ypyYfnPNbw==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp, v12.stomp
Sec-WebSocket-Extensions: permessage-deflate;
client_max_window_bits
```

Пример ответа при обновлении HTTP на WebSocket:

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
```

```
Upgrade: websocket
Access-Control-Allow-Origin: http://example.com
Sec-WebSocket-Accept: O1a/o0MeFzoDgn+kCKR91UkYDO4=
Sec-WebSocket-Protocol: v12.stomp
Sec-WebSocket-Extensions:                          permessage-
deflate;client_max_window_bits=15
```

Открывающее рукопожатие состоит из следующих частей: обновление протокола, согласование политик происхождения, согласование протокола, согласование подпротокола, согласование расширений .

Чтобы пройти обновление протокола :

- клиент отправляет запрос с заголовками Connection и Upgrade
- сервер подтверждает обновление протокола строкой ответа 101 Switching Protocols и теми же заголовками Connection и Upgrade

Чтобы передать согласование политик происхождения :

- клиент отправляет заголовок Origin (схема, имя хоста, номер порта)
- сервер подтверждает, что клиенту из этого источника разрешен доступ к ресурсу через заголовок Access-Control-Allow-Origin

Чтобы пройти согласование протокола :

- клиент отправляет заголовки Sec-WebSocket-Version (список версий протокола, 13 для RFC 6455) и Sec-WebSocket-Key (автоматически сгенерированный ключ)
- сервер подтверждает протокол, возвращая заголовок Sec-WebSocket-Accept

Эквивалентный код Java для вычисления заголовка Sec-WebSocket-Accept :

```
Base64
        .getEncoder()
        .encodeToString(
                MessageDigest
                        .getInstance("SHA-1")
                                .digest((secWebSocketKey +
"258EAFA5-E914-47DA-95CA-C5AB0DC85B11")
                                 .getBytes(StandardCharset
s.UTF_8)));
```

Чтобы пройти *согласование подпротокола* :

- клиент отправляет список субпротоколов через *Sec-WebSocket-протокол* заголовок
- сервер выбирает один из подпротоколов через заголовок *Sec-WebSocket-Protocol* (если сервер не поддерживает какой-либо подпротокол, соединение отменяется)

Чтобы передать *согласование расширений* :

- клиент отправляет список расширений через заголовок *Sec-WebSocket-Extensions*
- сервер подтверждает *одно или несколько* расширений через заголовок *Sec-WebSocket-Extensions* (если сервер не поддерживает некоторые расширения, соединение продолжается без них)

После успешного рукопожатия клиент и сервер переключаются с текстового протокола HTTP на формирование бинарных сообщений WebSocket и могут выполнять полнодуплексную связь.

Обрамление сообщения

WebSocket использует двоичную структуру сообщения: отправитель разбивает каждое *сообщение* приложения на один или несколько *кадров* , транспортирует их по сети к месту назначения, повторно собирает их и уведомляет получателя после получения всего сообщения.

Обрамление WebSocking имеет следующий формат:

1. FIN (1 бит) - флаг, который указывает, является ли кадр последним кадром сообщения
2. reserve (3 бита) - флаги резерва для расширений
3. operation code - код операции (4 бита) - тип кадра: кадры данных (текстовые или двоичные) или контрольные кадры (закрытие соединения, пинг / понг для проверки работоспособности соединения)
4. mask (1 бит) - флаг, указывающий, замаскированы ли данные полезной нагрузки (все кадры, отправленные от клиента к серверу, замаскированы)
5. payload length (длина полезной нагрузки) (7 бит, или 7 + 16 бит, или 7 + 64 бит) - длина полезной нагрузки переменной длины (если 0–125, то

это длина полезной нагрузки; если 126, то следующие 2 байта представляют длину полезной нагрузки ; если 127, то следующие 8 байтов представляют длину полезной нагрузки)

6. masking key (маскирующий ключ) (0 или 4 байта) - маскирующий ключ содержит 32-битное значение, используемое для XOR данных полезной нагрузки
7. payload data данные полезной нагрузки (n байтов) - данные полезной нагрузки содержат данные расширения (если расширения используются), объединенные с данными приложения

В таком двоичном кадрировании сообщения поле длины полезной нагрузки переменной длины позволяет снизить накладные расходы на кадрирование при обмене сообщениями даже большого размера. Согласно некоторым источникам, протокол WebSocket по сравнению с протоколом HTTP может обеспечить сокращение трафика примерно на 500: 1 и уменьшение задержки на 3: 1.

Заключительное рукопожатие

Любая из сторон может инициировать закрывающее рукопожатие, отправив закрывающий кадр. При получении такого кадра другая сторона отправляет в ответ закрывающий кадр, если он еще не отправил его. После отправки заключительного кадра сторона больше не отправляет никаких данных. После получения заключительного кадра сторона отбрасывает все полученные данные. Как только сторона отправила и получила закрывающий фрейм, эта конечная точка закрывает соединение WebSocket.

Помимо закрытия соединения посредством закрывающего рукопожатия, соединение WebSocket может быть внезапно закрыто, когда другая сторона уходит или закрывается базовая коллекция TCP. Коды состояния в закрывающих кадрах могут определить причину.

Существуют различные библиотеки Java, которые позволяют разработчикам создавать приложения на основе Websocket на Java. Далее рассмотрим пример использования Websocket в Spring Framework.

Websocket в Spring Framework

Когда вышел Spring, это была более простая и легкая альтернатива J2EE, облегчающая разработку J2EE. Enterprise Java относится к корпоративному программному обеспечению Java. Это компьютерное программное обеспечение, используемое для удовлетворения потребностей организации, а не отдельных пользователей.

Каковы преимущества использования Spring при создании корпоративных сложных приложений?

Spring Framework использует Java POJO (простой старый объект Java), что значительно упрощает создание приложений корпоративного класса по сравнению с тяжелыми EJB (из более ранних версий J2EE). POJO - это объект Java, на который не накладываются никакие ограничения, кроме тех, которые установлены спецификацией языка Java. Таким образом, POJO не должен расширять заранее заданные классы или реализовывать заранее заданные интерфейсы во фреймворке.

Преимущества использования Spring Framework:

- Способствует ослаблению зацепления за счет использования внедрения зависимости . Таким образом, вместо того, чтобы жестко кодировать зависимости объекта, вы просто указываете зависимости через файл конфигурации (или аннотации, или код Java).
- Использует **АОП (аспектно-ориентированное программирование).** Это позволяет отделить сквозные задачи (например, ведение журнала, аудит, декларативные транзакции, безопасность, кеширование и т. Д.) от бизнес-логики.
- **Минимизирует шаблонный код Java**. В Spring есть набор вспомогательных пакетов и классов, чтобы упростить работу и избежать шаблонного кода, и вам нужно беспокоиться только о тех классах, которые вам нужны, и игнорировать остальные.
- Он использует некоторые из **существующих технологий,** таких как фреймворки ORM, фреймворки журналирования, таймеры JEE, Quartz и JDK и другие.
- Основные функции среды Spring можно использовать при разработке любого приложения Java, но есть расширения для создания **веб-приложений** на основе платформы Java EE.

Чуть подробнее рассмотрим основные концепции, которые лежат в основе Spring Framework.

Внедрение зависимостей (DI)

Инверсия управления (IoC) является принцип конструкции , в которой поток управления системой инвертируется. Это общая концепция, конкретным примером которой является внедрение зависимостей. Внедрение зависимостей отделяет объект от его зависимостей, поэтому мы можем решить, какие зависимости должны быть внедрены во время выполнения. Это решение принимается в Spring в файле конфигурации

beans (или с использованием аннотаций или кода Java) вместо непосредственного изменения кода.

Преимущество заключается в том, что мы можем заменять и расширять функциональность зависимостей (служб), вообще не меняя зависимый класс (приложение).

Аспектно-ориентированное программирование (АОП)

Именно АОП позволяет отделить сквозные задачи от бизнес-логики. Сквозные задачи - это функции, которые охватывают разные части приложения и влияют на всё приложение. Например, ведение журнала, декларативные транзакции, безопасность, кеширование и т. Д.

Что произойдет, если нам нужно изменить, например, конкретный механизм безопасности для всего приложения. Такое изменение потребует больших усилий, особенно если этот аспект безопасности используется во многих методах.

Таким образом, АОП пытается решить эту проблему, позволяя разработчикам выражать сквозные проблемы в отдельных модулях, называемых «аспектами».

Например, «аспект» модуля безопасности может включать в себя «совет», который выполняет проверку безопасности (фактический фрагмент кода, который будет выполнен), и «точку соединения», которая определяет, когда (т.е. когда код выполняется) в соответствующем запрограммированном коде данный аспект должен быть выполнен.

Это может быть сделано либо с использованием синтаксиса XML, либо на основе синтаксиса « AspectJ », который использует код Java.

Модульная архитектура

Spring организован по модульному принципу, что позволяет беспокоиться только о тех модулях, которые вам нужны, и игнорировать остальные. Модульное программирование - это метод разработки программного обеспечения, который разделяет функциональные возможности программы на независимые модули, каждый из которых содержит одну конкретную функциональность. Модульность и АОП - Ключевой единицей модульности в ООП является класс, тогда как в АОП единицей модульности является аспект.

В Spring предусмотрено около 20 модулей. Все модули построены на верхней части его основного контейнера. Эти модули предоставляют все, что может понадобиться разработчику для разработки корпоративных приложений.

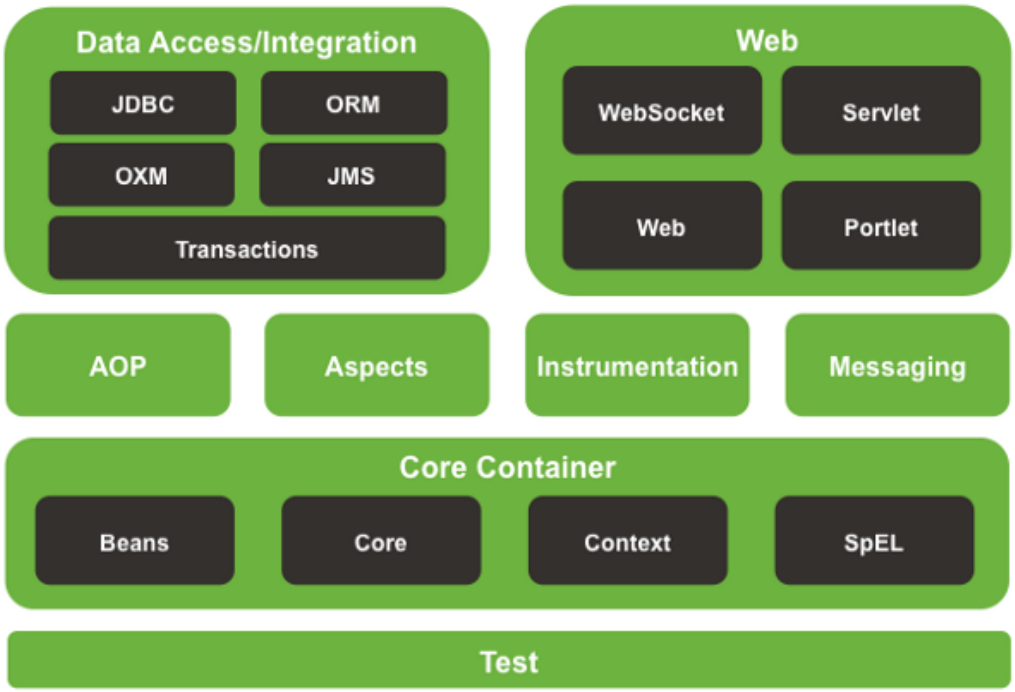


Рисунок 4.3 - Модульность в Spring

Основной контейнер

Он используется для создания bean-компонентов и управления зависимостями bean-компонентов.

- Core и Bean модули обеспечивают основные части структуры, в том числе особенности инъекций IoC и зависимостях.
- Модуль Bean предоставляет BeanFactory , которая представляет собой сложную реализацию фабричного шаблона. Он отделяет конфигурацию и спецификацию зависимостей от фактической логики программы.
- Модуль Context основан на модулях Core и Beans. Это средство доступа к любым определенным и настроенным объектам.
- Модуль SpEL предоставляет мощный язык выражений для запросов и управления объектами во время выполнения.

Модули доступа к данным / интеграции
Он используется для связи с базой данных.

- **JDBC** : Предоставляет уровень абстракции JDBC, который устраняет необходимость в кодировании, связанном с JDBC.

- **ORM** : предоставляет уровни интеграции для популярных API объектно-реляционного сопоставления, включая JPA, JDO, Hibernate и iBatis.
- **OXM** : предоставляет уровень абстракции, который поддерживает реализации сопоставления объектов / XML.
- **Служба обмена сообщениями Java (JMS)** : содержит функции для создания и использования сообщений (электронной почты).
- **Transactions**: поддерживает управление транзакциями для классов.

Веб-модули

Это реализация MVC в рамках фреймворка Sring . Можно интегрировать другие технологии, такие как JSF.

- **Internet**: предоставляет базовые функции интеграции, ориентированные на веб-интерфейс, такие как возможность загрузки нескольких файлов и инициализация контейнера IoC с использованием прослушивателей сервлетов и контекста веб-приложения.
- **Web-MVC**: содержит реализацию Spring Model-View-Controller (MVC) и веб-служб REST для веб-приложений.
- **Веб-сокеты**: обеспечивает поддержку двусторонней связи на основе **веб-сокетов** между клиентом и сервером в веб-приложениях.
- **Web-Portlet**: Предоставляет реализацию MVC для использования в среде портлета и отражает функциональность модуля Web-MVC.

АОП и инструменты

- **АОП** : обеспечивает реализацию аспектно-ориентированного программирования, позволяющую отделить сквозные задачи от бизнес-логики (как объяснялось ранее).
- **Аспекты** : это отдельный модуль, обеспечивающий интеграцию с AspectJ.
- **Инструментарий** : обеспечивает поддержку инструментовки класса и реализаций загрузчика классов.

Обмен сообщениями

Он служит основой для приложений на основе обмена сообщениями.

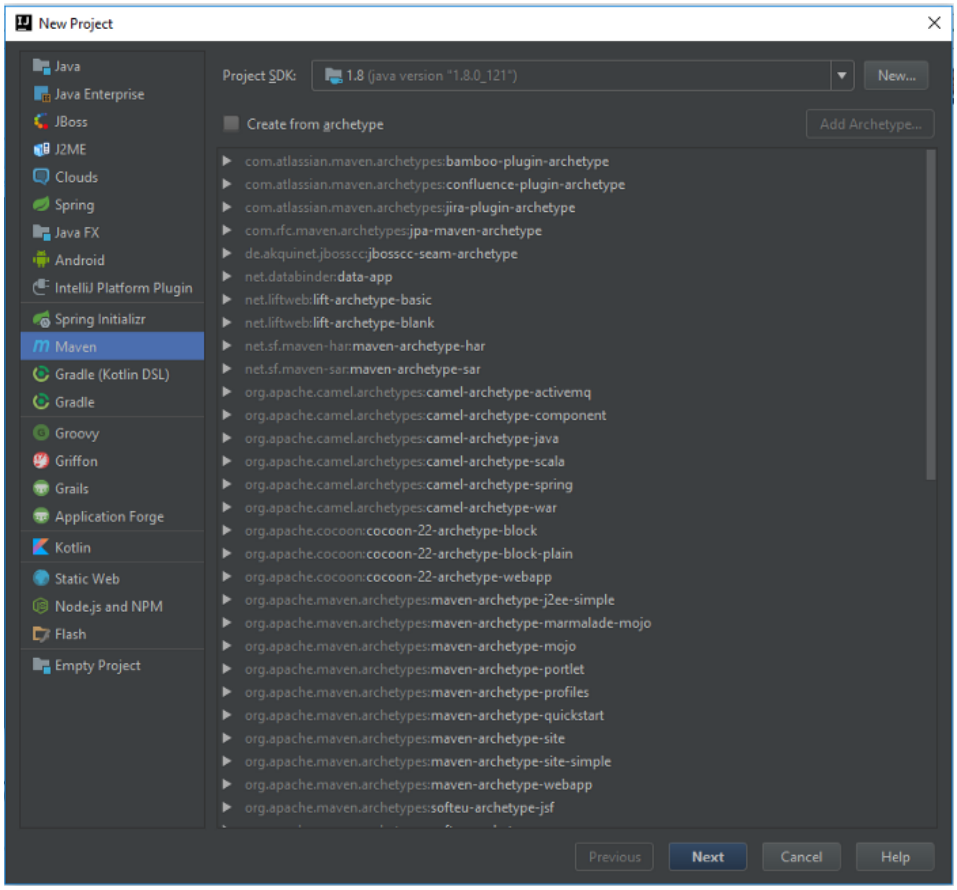
Он предоставляет функции асинхронной системы обмена сообщениями и обеспечивает поддержку STOMP для использования через WebScokets для обмена сообщениями между пользователями.

Далее рассмотрим пример создания простейшего приложения с использованием Websocket и Spring.

Простейшее приложение с использованием Websocket и Swing

Создадим простейший чат с помощью Spring. Предполагается, что первоначальная установка и настройка Spring произведена (можно прочитать об этом на сайте разработчика). Крайне рекомендуется использовать Maven, что избавит от необходимости вручную загружать файлы jar. Преимущества такого подхода заключается в том, что вы просто добавляете все зависимости в файл «pom.xml», который можно найти внутри проекта, при его создании изначально или после его преобразовании в проект Maven и далее через меню обновляете свой проект.

На рисунке ниже показан пример создания Maven проекта в IntelliJ IDEA.



После завершения настройки у вас должен быть пустой проект с файлом конфигурации Maven pom.xml. Поскольку Maven является диспетчером зависимостей, вы можете добавить к нему все зависимости проекта.

```
<dependency>
```

```
        <groupId>org.springframework</groupId>
        <artifactId>spring-websocket</artifactId>
        <version>5.2.2.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-messaging</artifactId>
        <version>5.2.2.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.10.2</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.10.2</version>
    </dependency>
```

Если нужны последние версии библиотек, то можно поискать их в https://search.maven.org/classic/.

Далее нужно включить возможности WebSocket в проекте. Для этого нужно добавить конфигурацию в приложение и аннотировать этот класс с помощью @EnableWebSocketMessageBroker .

Как следует из названия, он позволяет обрабатывать сообщения WebSocket при поддержке брокера сообщений (файл WebSocketConfig.java):

```
    @Configuration
    @EnableWebSocketMessageBroker
    public class WebSocketConfig extends
AbstractWebSocketMessageBrokerConfigurer {

        @Override
            public void configureMessageBroker(MessageBrokerRegistry
config) {
            config.enableSimpleBroker("/topic");
            config.setApplicationDestinationPrefixes("/app");
        }

        @Override
```

```
          public void registerStompEndpoints(StompEndpointRegistry
registry) {
            registry.addEndpoint("/chat");
            registry.addEndpoint("/chat").withSockJS();
        }
    }
```

Здесь метод configureMessageBroker используется для настройки брокера сообщений . Во-первых, мы разрешаем брокеру сообщений в памяти передавать сообщения обратно клиенту в места назначения с префиксом «/ topic».

Завершается простая настройка, назначив префикс «/app» для фильтрации адресатов, нацеленных на аннотированные методы приложения (через @MessageMapping).

Метод registerStompEndpoints регистрирует конечную точку «/chat», обеспечивая поддержку STOMP в Spring . Имейте в виду, что мы также добавляем сюда конечную точку, которая работает без SockJS ради эластичности.

Эта конечная точка с префиксом «/app» является конечной точкой, для обработки которой сопоставляется метод ChatController.send () .

Он также включает резервные параметры SockJS , чтобы можно было использовать альтернативные параметры обмена сообщениями, если WebSockets недоступны. Это полезно, поскольку WebSocket еще не поддерживается во всех браузерах и может быть заблокирован ограничивающими сетевыми прокси.

Резервные варианты позволяют приложениям использовать API WebSocket, но при необходимости постепенно переходить на альтернативы, не относящиеся к WebSocket, во время выполнения.

Далее нужно создать **модель сообщения для отправки**. Конечная точка будет принимать сообщения, содержащие имя отправителя и текст в сообщении STOMP, тело которого является объектом JSON .

Сообщение может выглядеть так:

```
{
   "from": "John",
   "text": "Hello!"
}
```

Чтобы смоделировать сообщение, несущее текст, необходимо создать простой объект Java со свойствами from и text (файл Message.java) :

```
public class Message {

   private String from;
```

```
    private String text;

    // методы get и set
}
```

По умолчанию Spring будет использовать библиотеку jackson для преобразования объекта модели в JSON и обратно.

Подход Spring к работе с обменом сообщениями STOMP заключается в связывании метода контроллера с настроенной конечной точкой. Это стало возможным благодаря аннотации @MessageMapping .

Связь между конечной точкой и контроллером дает возможность при необходимости обрабатывать сообщение:

```
@MessageMapping("/chat")
@SendTo("/topic/messages")
public OutputMessage send(Message message) throws Exception {
        String time = new SimpleDateFormat("HH:mm").format(new
Date());
                return new OutputMessage(message.getFrom(),
message.getText(), time);
}
```

Создадим другую модель объекта с именем OutputMessage, представляющая собой сообщение вывода, которое отправляется на указанное место назначения. Заполняем объект отправителем и текстом сообщения, взятым из входящего сообщения, и дополняем его меткой времени.

После обработки нашего сообщения отправляем его в соответствующее место назначения, определенное аннотацией @SendTo . Все подписчики на пункте назначения « / topic / messages » получат сообщение.

После настройки на стороне сервера в данном проекте будет использоваться библиотека sockjs-client для создания простой HTML-страницы, которая взаимодействует с разработанной системой обмена сообщениями.

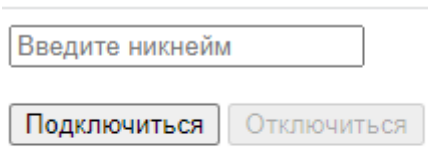
Прежде всего, нужно импортировать клиентские библиотеки Sockjs и Stomp Javascript. Затем можно создать функцию connect () для открытия связи с конечной точкой, функцию sendMessage () для отправки сообщения STOMP и функцию disconnect () для закрытия связи:

```
<html>
    <head>
        <title>Chat WebSocket</title>
```

```
        <script src="resources/js/sockjs-0.3.4.js"></script>
        <script src="resources/js/stomp.js"></script>
        <script type="text/javascript">
            var stompClient = null;

            function setConnected(connected) {
                        document.getElementById('connect').disabled =
connected;
                  document.getElementById('disconnect').disabled = !
connected;
                 document.getElementById('conversationDiv').style.vis
ibility
                  = connected ? 'visible' : 'hidden';
                document.getElementById('response').innerHTML = '';
            }

            function connect() {
                var socket = new SockJS('/chat');
                stompClient = Stomp.over(socket);
                stompClient.connect({}, function(frame) {
                    setConnected(true);
                    console.log('Connected: ' + frame);
                             stompClient.subscribe('/topic/messages',
function(messageOutput) {
                         showMessageOutput(JSON.parse(messageOutput.b
ody));
                    });
                });
            }

            function disconnect() {
                if(stompClient != null) {
                    stompClient.disconnect();
                }
                setConnected(false);
                console.log("Disconnected");
            }

            function sendMessage() {
                var from = document.getElementById('from').value;
                var text = document.getElementById('text').value;
                stompClient.send("/app/chat", {},
                  JSON.stringify({'from':from, 'text':text}));
            }

            function showMessageOutput(messageOutput) {
                var response = document.getElementById('response');
                var p = document.createElement('p');
                p.style.wordWrap = 'break-word';
                 p.appendChild(document.createTextNode(messageOutput.
from + ": "
```

```
                    + messageOutput.text + " (" + messageOutput.time +
")"));
                response.appendChild(p);
            }
        </script>
    </head>
    <body onload="disconnect()">
        <div>
            <div>
                  <input type="text" id="from" placeholder="Измените
никнейм"/>
            </div>
            <br />
            <div>
                                                  <button   id="connect"
onclick="connect();">Подключиться</button>
                        <button  id="disconnect"  disabled="disabled"
onclick="disconnect();">
                    Отключиться
                </button>
            </div>
            <br />
            <div id="conversationDiv">
                   <input type="text" id="text" placeholder="Введите
текст сообщения..."/>
                                               <button   id="sendMessage"
onclick="sendMessage();">Отправить</button>
                <p id="response"></p>
            </div>
        </div>

    </body>
</html>
```

Чтобы проверить наш пример, можно открыть пару окон браузера и получить доступ к странице чата по адресу:
http://localhost:8080



Задание на практическую работу

Используя информацию из данной практической работы, необходимо реализовать клиент- серверное приложение с использованием Websocket. Суть приложения заключается в следующем. При обращении клиентской части по адресу /webs необходимо выполнять обработку Websocket. В случае, получения в вебсокете данных, необходимо ответить их же содержимым.