# Поднятие контекста и конфиг SpringMvcKarkas

Если настройка через Java, а не через xml, то **WebApplicationInitializer заменяет web.xml, работает с Servlet 3.0**. (Кстати есть еще AbstractAnnotationConfigdispatcherservletInitializer, читай ниже) Это класс из спринг web, соответственно он и поднимает контекст и передает его в метод onstartup. Этот метод будет автоматически вызываться сервлетом при загрузке приложения.

[**AbstractAnnotationConfigDispatcherServletInitializer**](https://ru.stackoverflow.com/questions/954151/%d0%9e%d1%82%d0%bb%d0%b8%d1%87%d0%b8%d0%b5-webapplicationinitializer-%d0%be%d1%82-abstractannotationconfigdispatcherservletin)один из классов который наследует интерфейс WebApplicationInitializer, он нужен что бы сделать жизнь проще. Просто в определенных местах можно указать конфиг, кодировку, гласный путь и тд. Так жн он имеет фильтры, например на ту же кодировку. В WebApplicationInitializer можно все тоже самое, плюс вообще полнейшую настройку сделать, то есть полностью сконфигурировать контекст вручную. [AbstractAnnotationConfigDispatcherServletInitializer](https://ru.stackoverflow.com/questions/954151/%d0%9e%d1%82%d0%bb%d0%b8%d1%87%d0%b8%d0%b5-webapplicationinitializer-%d0%be%d1%82-abstractannotationconfigdispatcherservletin) это упрощение, используя его получаешь из коробки почти готовый контекст сервлета.

В примере каркасе я привожу оба способа поднять контекст и засунуть в него конфиг.

Есть еще ApplicationContextInitializer, но это отдельная штука, она не является заменой web.xml А используется вместе с ним, а возможно и с WebApplicationInitializer. За исключением суффикса Initializer, оба WebApplicationInitializer и ApplicationContextInitializer служат довольно различным целям. Принимая во внимание, что **WebApplicationInitializer используется контейнером сервлетов при запуске веб-приложения и предоставляет способ программного создания веб-приложения (замена файла web.xml)**, ApplicationContextInitializer предоставляет ловушку для настройки контекста приложения Spring до его полного создания.  По сути, это код, который выполняется до того, как полностью создается контекст приложения Spring. (Где он применяется пока не ясно, просто наткнулась на эту штуку и записала сюда.)

Еще немного про WebApplicationInitializer. Все классы, которые находятся в classpath и наследуют от WebApplicationInitializer контейнер сервлетов (Tomcat) найдет и вызовет и передаст в метод этого класса onStart объект ServletContext.

То есть обратить внимание, есть servletContext, а есть ApplicationContext (spring контекст) разобрать это. Вспомнить у нас на стажировке что-то говорили про наследование контекстов, возможно это оно.

# CORS

CORS – браузеры по умолчанию не дают обмениваться данными, если бэкенд и фронтенд лежит на разных серверах. Для этого надо задать какие адреса серверов мы считаем надежными. Можно хоть все разрешить. Это политика CORS. Записана в конфиге приложения.

# DriverManagerDataSource

В этом приложении используется DriverManagerDataSource, это простое подключение к БД, без пула коннектов. Конесно на реальных проектах лучше настроить пулл.

Привести пример с пуллом

# WebMvcConfigurer

Расписать что это?

# GIT

Создаем в корне файл .gitignore и записываем туда то, что не должно в git попадать

Например

.idea  
target  
\*.iml

Далее

Git init инициализировать репозиторий

Git add . добавить все файлы

git reset example.txt - удалить случайно добавленный командой add файл

Далее делаем коммит. Удобнее через идею.

Перенести сюда напоминалку по гит, залить ее на github

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

Поподробнее что вот это делает

public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {  
 log.info("configuredMessageConverters");  
 ObjectMapper mapper = new ObjectMapper();  
 mapper.registerModule(new Hibernate5Module());  
 mapper.registerModule(new JavaTimeModule());  
 mapper.configure(SerializationFeature.*WRITE\_DATES\_AS\_TIMESTAMPS*, false); // Задаем дату в читабельном виде  
 mapper.setVisibility(PropertyAccessor.*ALL*, JsonAutoDetect.Visibility.*NONE*); // Отключаем чтение через шз геттеры и сеттеры  
 mapper.setVisibility(PropertyAccessor.*FIELD*, JsonAutoDetect.Visibility.*ANY*); // Подключаем чтение напрямую из полей  
 mapper.setSerializationInclusion(JsonInclude.Include.*NON\_NULL*); // Не выводить null поля  
  
 converters.add(new MappingJackson2HttpMessageConverter(mapper)); // Задаем кодировку и медиатайп  
 StringHttpMessageConverter stringHttpMessageConverter = new StringHttpMessageConverter();  
 stringHttpMessageConverter.setSupportedMediaTypes(Arrays.*asList*(new MediaType("text", "plain", Charset.*forName*("UTF-8")),  
 new MediaType("text", "html", Charset.*forName*("UTF-8"))));  
 converters.add(stringHttpMessageConverter);  
}

***Обратить внимание, есть Spring orm, а есть Spring data jpa. Настажировке оба этих способа работают с Hibernate.Это просто два отдельных вреймворка, orm и data. Просто data поновее, помоднее. Но работают они на основе нашего spring-jdbc.***

Итого

1. spring-jdbc это наш первый способ работы с базой, через jdbcTemplate и mapper, это тот случай, где resultSet и прочие походие на него штуки, которые я выписывала. Настройка БД делается через DriverManagerDataSourse если нам нужно одно соединение(в продакшен так лбычно не делают) или через пул коннектов, если нужно много соединений.
2. второй spring orm + hibernate(мнтересно, а без хибера можно?), через entityManager и квери, неймедквери . Подключение к базе делается через DriverManagerDataSourse, так же указываем EntityManagerFactory, так как не используем распределенные транзакции, то класс EntityManagerFactory это LocalContainerEntityManagerFactoryBean. Но это уже не база, а настройка для гибернейта.
3. третий это spring data jpa+ hibernate(мнтересно, а без хибера можно?) через jpaRepository. Настройка БД и Хибернейт аналогична пункту 2.

На этой стажировке хорошо запомнился момент с тем, что у спринг есть контекст. Мы его или явно прописываем и скармливаем ему либо xml либо @Config с настройками, либо он должен подниматься через ContextLoaderListener. В context-param прописываются xml файлы контекста. Прописывается он в web.xml, ну логично, а где еще? Web.xml это первый файл нашего приложения. Если используем сервлеты, а не mvc, то у сервлета можно получить этот контекст через getServletContext и передать его в webContext. Ну вот такая система. Главное, что наш сервлет может получить контекст, который поднял ContextLoaderListener.

Профили кстати тоже в web.xml задаются, если контекст через ContextLoaderListener поднимается, иначе там же, где мы руками поднимаем спринг контекст, указываем текущий профиль.

Модель и репозиторий их автоскан прописывается в файле БД, а если настройка через Java, то у EntityManagerFactory или как свойство, либо модель как свойство, а репозиторий можно или как свойство или через аннотацию @EnableJpaRepository

Если настройка через Java, а не через xml, то WebApplicationInitializer заменяет web.xml. (Кстати есть еще AbstractAnnotationConfigdispatcherservletInitializer) Это класс из спринг web, соответственно он и поднимает контекст и передает его в метод onstartup. Этот метод будет автоматически вызываться сервлетом при загрузке приложения.

Написать конспект по jackson

# Далее по Jackson.

<https://www.baeldung.com/jackson-bidirectional-relationships-and-infinite-recursion>

Одна из самых популярных аннотаций @JsonAutoDetect. Эта аннотация ставится над всей энтити, а не над полями. Она позволяет настраивать сериализацию json, как и Object Mapper. Но посто аннотацией. То есть это упрощение. Например вот такая строчка @JsonAutoDetect(fieldVisibility = ANY, getterVisibility = NONE, isGetterVisibility = NONE, setterVisibility = NONE)

fieldVisibility – говорит видеть все поля

getterVisibility, isGetterVisibility, setterVisibility –не видить все геттеры, сеттеры и is геттеры.

Так же сможно указывать сериализовать только поля с определенными модификаторами доступа.

@JsonAutoDetect(fieldVisibility = JsonAutoDetect.Visibility.PUBLIC\_ONLY,

getterVisibility = JsonAutoDetect.Visibility.PUBLIC\_ONLY,

setterVisibility = JsonAutoDetect.Visibility.PUBLIC\_ONLY,

isGetterVisibility = JsonAutoDetect.Visibility.PROTECTED\_AND\_PUBLIC)

Эти четыре параметра регулируют процесс сериализации. @JsonAutoDetect имеет еще пятый метод, который отвечает за десериализацию.

@JsonAutoDetect(creatorVisibility = JsonAutoDetect.Visibility.PROTECTED\_AND\_PUBLIC)

Эта аннотация работает с конструкторами и с фабричными методами (методы, которые создают объект при обращении к ним). Поясню. К нам прилетает json, который надо превратить в объект. Jackcon ищет конструктор у этого объекта и запихивает прилетевший json в этот объект. В строке выше говорим что нам нужен конструктор только или с protected или c publick видимостью.

Далее сто бы десериализовать json мы создаем ObjectMapper и работаем через readValue. Это уже смотреть в коде. Есть еще writeValue. Записать в json.

Есть еще @JsonIdentityInfo – она так же как и ***JsonBackReference и @JsonManagedReference*** отвечает за сериализацию связных данных, но на месте связных данных она не формирует json, а пишет их количество.

Аннотация @JsonView - позволяет раскидать json из связных данных двух энтити по другит классам.

Можно написать кастомный сериализатор, пометив данные @JsonSerialize.

Так же можно десериализовать данные с **помошью @JsonIdentityInfo.**

И написать кастомный десериализатор **используя** @JsonDeserialize

Хорошая ссылка по всему, что описано выше <https://www.baeldung.com/jackson-bidirectional-relationships-and-infinite-recursion>

## Есть еще аннотация @JsonUnwrapped - используется для развертывания значений объектов во время сериализации или десериализации. (подробно не разбирала, что она делает)

jackson-datatype-jsr310 – библиотека, которую мы подключаем, что бы наша дата и всемя приходили в читабельном виде, нужно для java8 date API. То есть даты будут записываться в виде timestamp. Настраивается в ObjectMapper.

Вообще в Jackson очень много настроек делается через ObjectMapper mapper = new ObjectMapper();