

Руководство пользователя Amplicode

Список терминов и сокращений.....	7
Установка.....	9
Работа с проектом.....	9
Открытие проекта.....	9
Запуск проекта.....	13
Настройки конфигурации.....	14
Основные функции плагина.....	16
Панели инструментов.....	16
Палитра.....	18
Инспектор.....	19
Amplicode Explorer.....	20
Работа с JPA сущностями.....	21
Действия на боковой панели.....	22
Создание сущности.....	24
ID и выбор стратегии генерации.....	25
Валидация полей.....	27
JPA конверторы.....	28
Поддержка аудита сущностей.....	30
Работа с MongoDB документами.....	32
Действия на боковой панели.....	33
Создание MongoDB документа.....	35
Поддержка аудита MongoDB документов.....	37
Spring Data репозитории.....	37
Создание репозитория.....	37
Репозитории на панели Amplicode Explorer.....	43
Создание запросов и методов.....	44
Пример.....	44
Entity Intention.....	46
Некорректные ссылки.....	47
Изменение существующих запросов и методов.....	47
Поддержка EntityGraph.....	48
Проекция.....	50
Синхронизация проекции и сущности.....	52

Навигация между сущностью и проекциями.....	52
Работа с DTO	52
Различные опции при создании DTO	55
Поддержка Lombok	57
Вложенные DTO для атрибутов ассоциации	58
Поддержка Java Records.....	59
Пример	59
Создание JPA сущностей из DTO	60
Настройка ассоциативных атрибутов	60
Поддержка MapStruct.....	61
Генерация методов	61
Использование дженерик MapStruct интерфейса	61
Синхронизация DTO и доменного объекта.....	62
Изменение существующих атрибутов	62
Добавление атрибутов.....	63
Настройки создания DTO	65
Правила валидации	66
Навигация между сущностью и DTO.....	66
Соединение с базой данных	68
Подключение к нестандартной схеме базы данных.....	68
PostgreSQL.....	68
Microsoft SQL Server.....	69
Oracle.....	70
MySQL & MariaDB	72
Драйверы баз данных	72
Соединение с MongoDB	73
Версионирование баз данных	75
Начинаем с базы данных.....	75
Начинаем с исходного кода	75
Поддержка различных библиотек версионирования баз данных	76
Стандартная последовательность действий при генерации дифференциальных скриптов.....	78
Опции при генерации дифференциальных миграционных скриптов	79

Использование базы данных.....	79
Использование модели данных	79
Использование snapshot модели данных	80
Генерация скриптов инициализации.....	82
Окно предпросмотра миграционного скрипта.....	83
Объединение действий в скрипте.....	85
Поддержка Liquibase	86
Окно предпросмотра changelog	87
Первичные и вторичные changelogs.....	89
Создание и изменение changelogs	89
Выполнение Liquibase changelogs и предпросмотр SQL без плагинов Gradle/Maven	92
Настройки Liquibase	95
Базовые настройки.....	95
Шаблоны changelog-файлов.....	95
Типы БД.....	96
Шаблоны для changeset	97
Поддержка Flyway	99
Окно предпросмотра миграции	99
Миграция на Java	100
Flyway Callbacks.....	101
SQL Callbacks	102
Java Callbacks.....	103
Настройки Flyway	105
Преобразование нестандартных типов.....	106
Конвертеры.....	107
Аннотация @JavaType	111
Стратегия именованя и настройки по максимальному идентификатору	113
Максимальный идентификатор	113
Reverse Engineering	115
Введение	115
Визард Entities from DB	117
Конфигурация	117

Мапированные отношения, таблицы и представления	117
Миграция атрибутов	118
Преобразование представлений БД (Views) на JPA сущности	121
Колонки для функции reverse engineering	122
Умный поиск отношений	123
@OneToOne (один к одному)	124
@OneToMany и @ManyToOne (один ко многим и многие к одному)	124
@ManyToMany (многие ко многим)	124
Работа с удаленной БД	124
Настройки	125
Общие	125
Комментарии к таблицам и колонкам	126
Правила именования	127
Преобразование типов	129
Spring MVC	133
Создание контроллеров	133
Делегирование методов в контроллер	136
Отображение методов в дереве Amplicode Explorer	140
Инспектор методов контроллера	140
CRUD REST Controllers	141
Создание новых методов в контроллере	144
Spring Security	146
Создание Security Configuration	147
Аутентификация с использованием HTTP сессией	148
JWT аутентификация с Keycloak	149
OIDC аутентификация с использованием HTTP сессии	150
Настройка правил авторизации в Spring Security	152
Настройка правил авторизации для Security Configuration	152
Настройки ролевого доступа в методах MVC/REST контроллеров	154
Отображение Spring Security в дереве Amplicode Explorer	155
Kafka	156
Создание через палитру	160
Постфикс send	165

Тесты	166
Docker	170
Создание Dockerfile	170
Создание Docker Compose файла	172
Kubernetes и Helm чарты	175
Создание React Admin модуля	182

Список терминов и сокращений

Термин или сокращение	Определение
ТЗ	Техническое задание
Инспектор	Часть боковой панели плагина, отображается в нижней части панели при переходе на вкладку Designer, позволяет редактировать свойства выделенного элемента кода.
Палитра	Часть боковой панели плагина, отображается в верхней части панели при переходе на вкладку Designer, позволяет добавлять в код новые компоненты. Набор доступных компонентов меняется в зависимости от того, в каком классе открыта панель.
Hibernate Envers	Популярный модуль, работающий в связке с JPA и Hibernate. Он позволяет отслеживать и сохранять изменения значений полей сущностей, а также работать со старыми состояниями этих сущностей (восстанавливать их значения из прошлых ревизий).
Hibernate Validator	Спецификация для валидации данных в приложениях
Lombok	Основанная на аннотациях библиотека Java, позволяющая сократить шаблонный код. В Lombok предлагаются различные аннотации, цель которых – заменить ненужный повторяющийся код, писать который утомительно.
Hibernate Types	Библиотека, используемая для сопоставления между типами SQL и примитивами Java / типом Object
MapStruct	Библиотека для Java, которая генерирует код для передачи данных между разными сущностями в программе.
Spring Data Audit	Модуль, позволяющий отслеживать и журналировать авторов и время изменения данных в проекте
Blaze Persistence Entity-View	Фреймворк, нацеленный на выделение слоя DTO. Позволяет определять DTO как интерфейсы и обеспечивает сопоставление модели JPA через аннотации.
DTO	(англ. Data Transfer Object) — один из шаблонов проектирования, используется для передачи данных между подсистемами приложения
Spring Data Mongo Repository	интерфейс фреймворка Spring Data, обеспечивающий интеграцию с документо-ориентированной СУБД MongoDB.
Инспекция	Подсказка от среды разработки IntelliJ IDEA или плагина для среды разработки для проверки качества и валидности кода. Инспекции помогают быстрее писать код в соответствии с самыми строгими стандартами качества.
REST Controller	Представляет аннотированные методы бина как HTTP точки выхода, используя методанные, предоставленные аннотацией @RequestMapping к каждому из методов
OAuth2	Открытый протокол (схема) авторизации, обеспечивающий предоставление третьей стороне ограниченного доступа к защищённым ресурсам пользователя без передачи ей (третьей стороне) логина и пароля.

LDAP	Протокол, использующий TCP/IP и позволяющий производить операции аутентификации (bind), поиска (search) и сравнения (compare), а также операции добавления, изменения или удаления записей.
JWT	Открытый стандарт для создания токенов доступа, основанный на формате JSON. Как правило, используется для передачи данных для аутентификации в клиент-серверных приложениях
REST API	(от англ. representational State Transfer — «передача репрезентативного состояния» или «передача „самоописываемого“ состояния») — архитектурный стиль взаимодействия компонентов распределённого приложения в сети.
HTTP Method	Последовательность из любых символов, кроме управляющих и разделителей, указывающая на основную операцию над ресурсом. Обычно метод представляет собой короткое английское слово, записанное заглавными буквами.
CORS	(от англ. cross-Origin Resource Sharing, «совместное использование ресурсов разных источников») — это стандарт, позволяющий предоставлять веб-страницам доступ к объектам сторонних интернет-ресурсов.
CSRF	(от англ. cross-site request forgery) или межсайтовая подделка запроса – это форма атаки на любой сайт или веб-приложение.
Kafka Producer	Брокер (или группа брокеров), который отвечает за производство и отправку сообщений Big Data остальным брокерам-получателям (broker consumer) в распределённом Kafka-кластере
Kafka Customer	Сервис (или группа брокеров), который отвечает за получение Big Data сообщений, созданных продюсером.
DDL	(англ. Data definition language) — семейство языков программирования, используемых для описания структуры базы данных.
JPA	(англ. Java Persistence API) — спецификация API Java EE, предоставляет возможность сохранять в удобном виде Java-объекты в базе данных.
Kotlin	Статически типизированный, объектно-ориентированный язык программирования, работающий поверх Java Virtual Machine и разрабатываемый компанией JetBrains
Maven	Фреймворк для автоматизации сборки проектов на основе описания их структуры в файлах на языке POM (англ. Project Object Model), являющемся подмножеством XML
Gradle	Система автоматической сборки, построенная на принципах Apache Ant и Apache Maven, но предоставляющая DSL на языках Groovy и Kotlin вместо традиционной XML-образной формы представления конфигурации проекта
Рефакторинг	(англ. refactoring), или перепроектирование кода, переработка кода, равносильное преобразование алгоритмов — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы

Установка

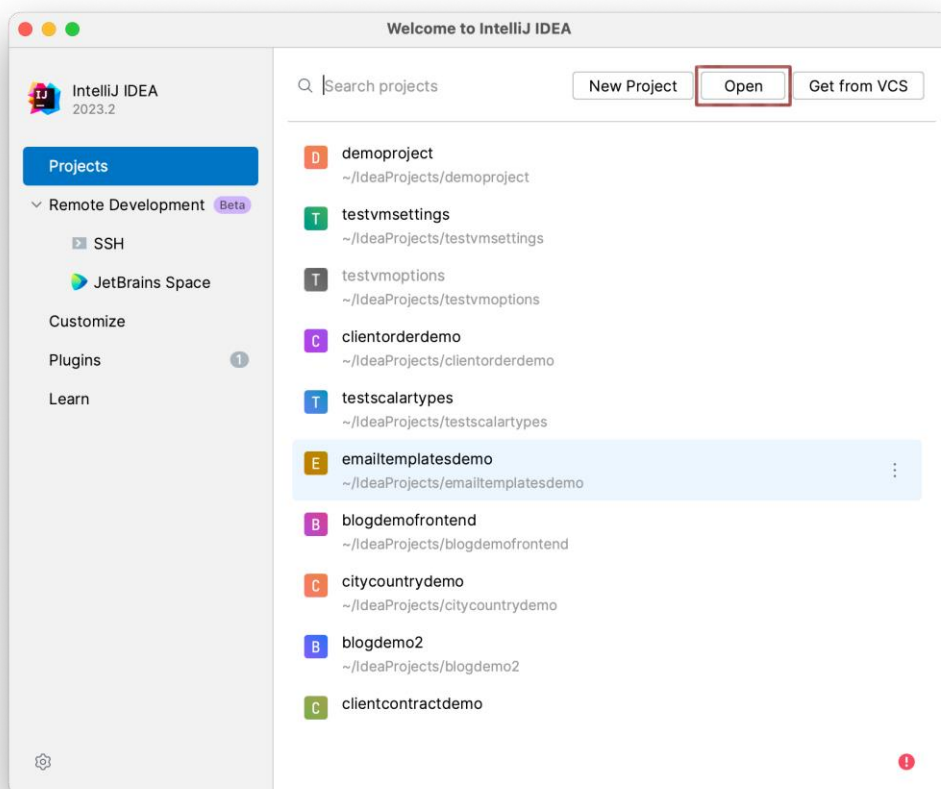
Для установки воспользуйтесь документом [“Инструкция по установке экземпляра программного обеспечения, предоставленного для проведения экспертной проверки”](#)

Работа с проектом

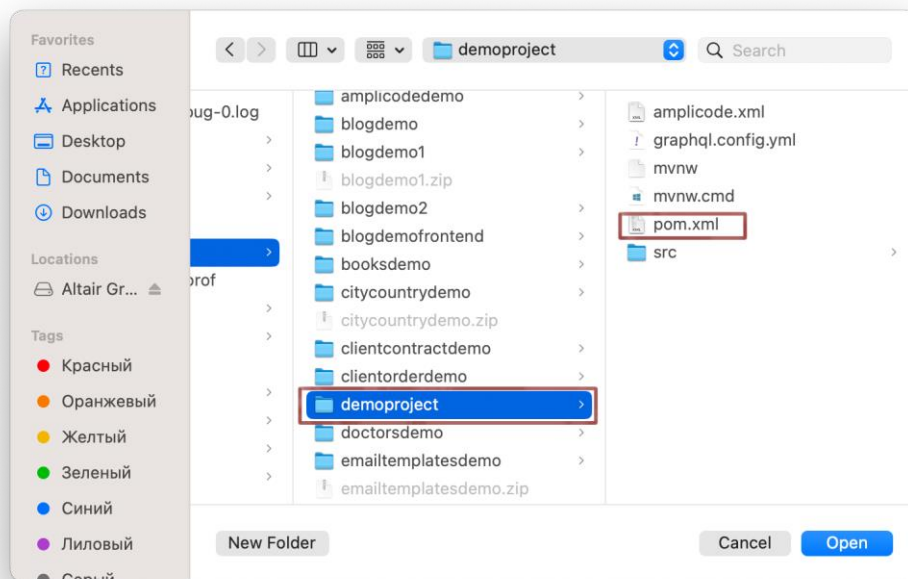
Открытие проекта

Для того, чтобы открыть существующий проект, повторите следующие шаги:

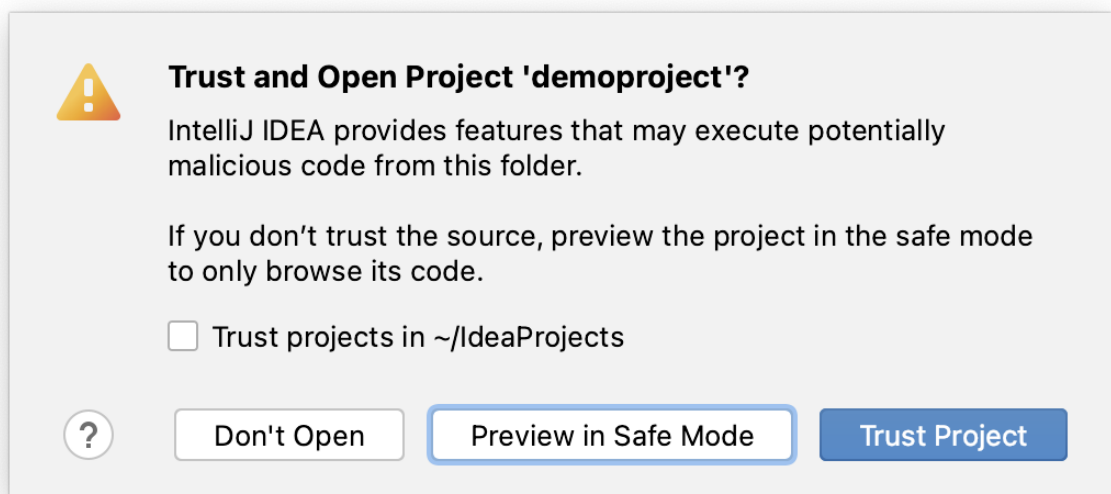
- Кликните Open на стартовой странице IDE. Если в IDE уже открыт другой проект, то выберите действие File → Open в основном меню IDE.



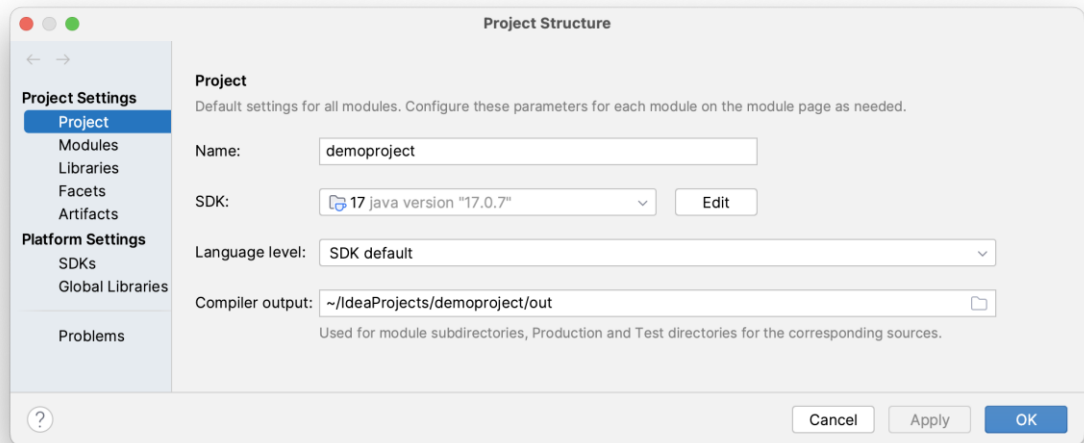
- В открывшемся диалоге файловой системы выберите корневую папку проекта, содержащую файл pom.xml или build.gradle и нажмите Open



- При первом открытии проекта появляется всплывающее окно с просьбой подтвердить, что мы доверяем этому проекту. Нажмите Trust Project для продолжения работы с проектом.



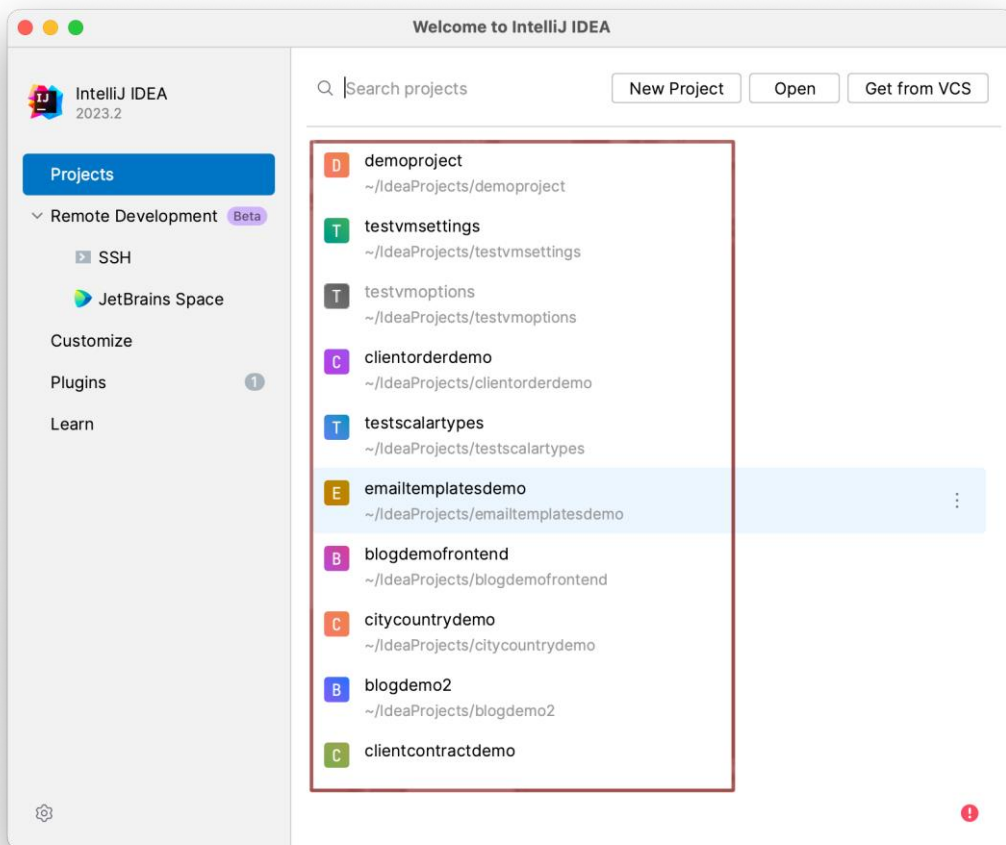
- Дождитесь окончания процесса синхронизации и инициализации проекта
- Если проект был открыт в первый раз, откройте File → Project Structure из основного меню



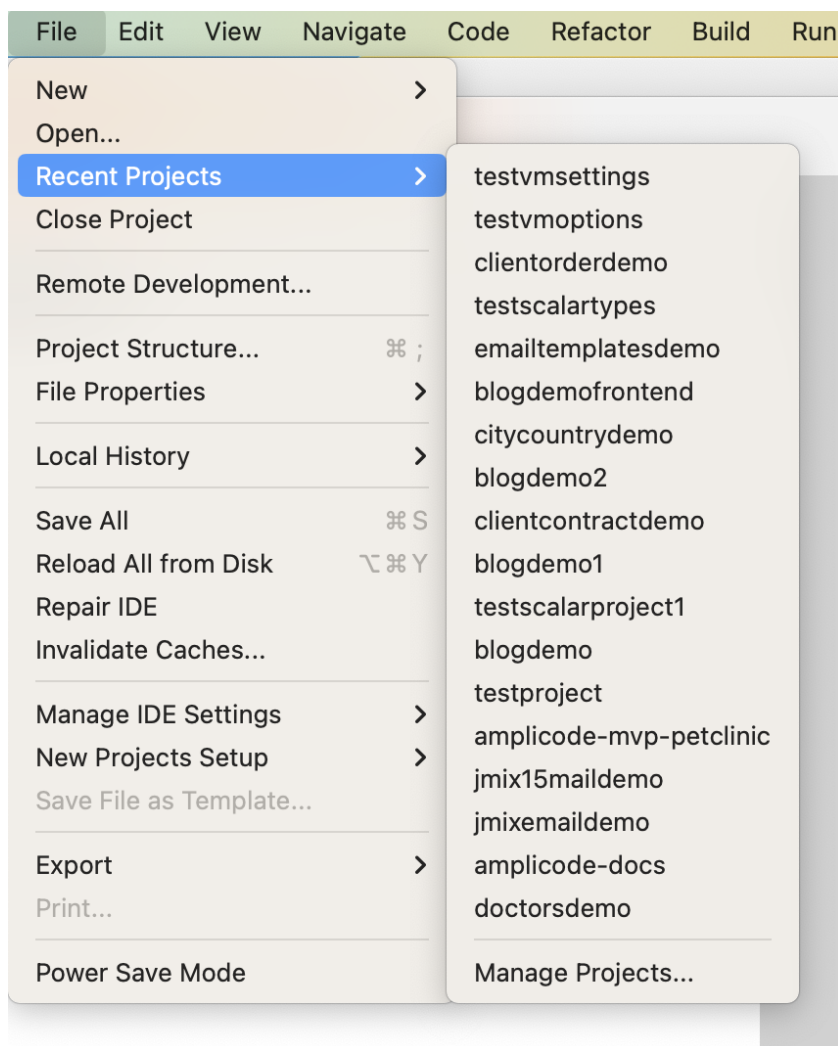
- Убедитесь, что поле Project SDK имеет значение, соответствующее JAVA_HOME, которое установлено в вашей среде. Если вы видите значение <No SDK>, выберите Add SDK из списка и выберите папку, в которой установлен JDK, например, C:\Program Files\BellSoft\LibericaJDK-17 в Windows или /Library/Java/JavaVirtualMachines/liberica-jdk-17.jdk/Contents/Home в macOS.
- Убедитесь, что поле Language level имеет значение, соответствующее версии JDK. Например, если JDK — 17, то и уровень языка должен быть 17 — Sealed types, always-strict floating-point semantics.

Также вы можете открыть проект из недавних проектов. Это можно сделать в стартовом окне IDE или с помощью действия File → Recent Projects из главного меню.

Стартовое окно:

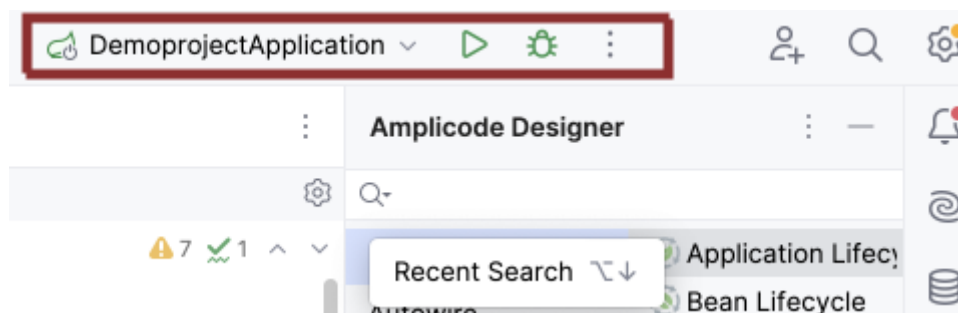


Меню:



Запуск проекта

Когда проект Amplicode импортируется, для него создается конфигурация запуска/отладки (Run/Debug Configuration). Вы можете использовать кнопки на главной панели инструментов для запуска и остановки вашего приложения.



Чтобы запустить приложение и подключиться к нему с помощью отладчика, просто нажмите кнопку отладки (debug) рядом с выбранной конфигурацией приложения Amplicode.

Статус можно отслеживать на вкладке Консоль окна инструмента отладки.

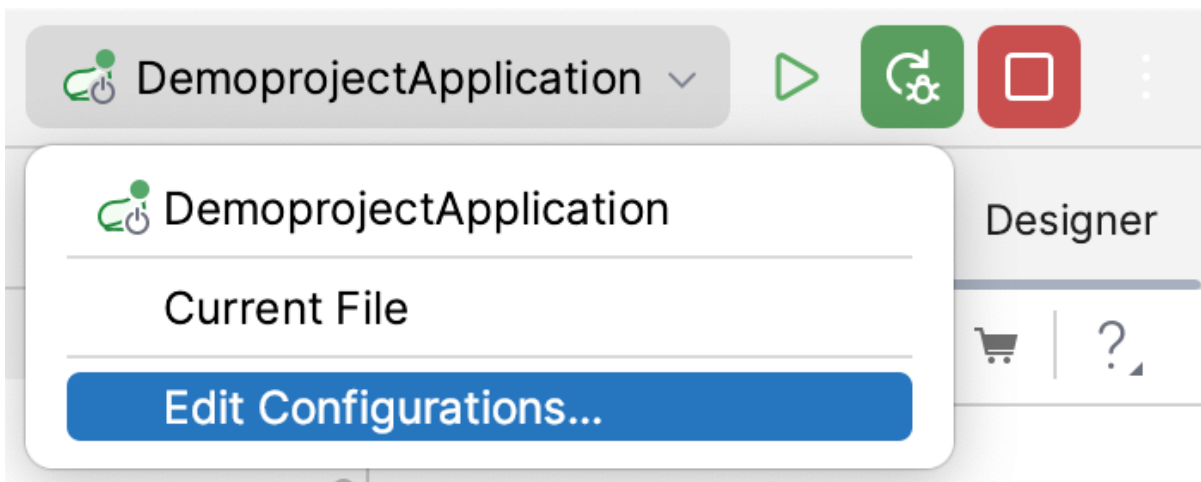


Чтобы остановить сервер приложений, нажмите кнопку «Остановить» на главной панели инструментов или в окне инструмента отладки.

Настройки конфигурации

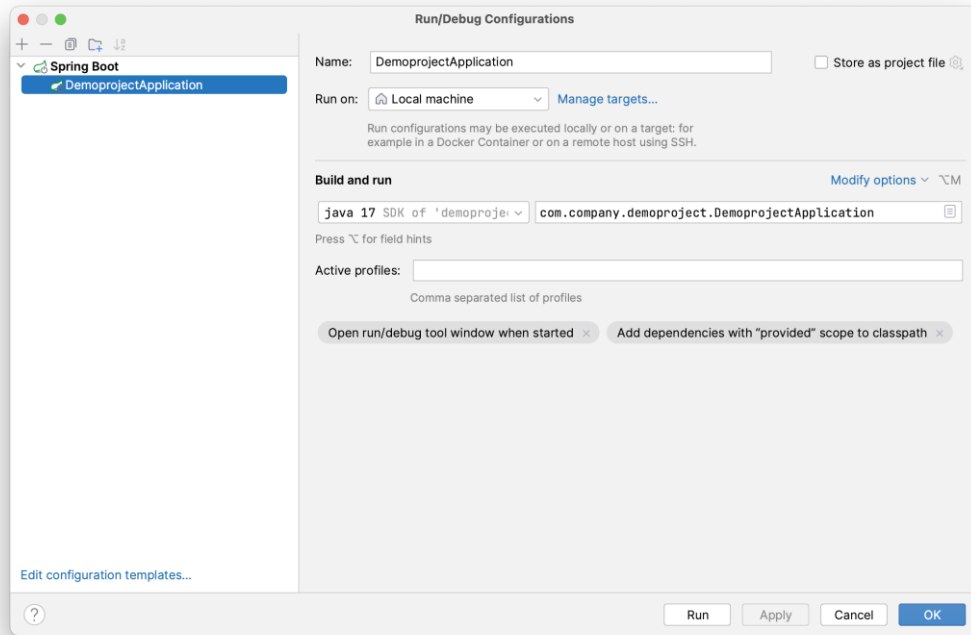
Вы можете настроить параметры приложения, запущенного из IDE, отредактировав Application Run/Debug Configuration.

Вы можете открыть диалоговое окно настроек, кликнув на элемент Amplicode Application на панели инструментов и выбрав Edit Configurations в контекстном меню.



Также конфигурацию можно открыть на редактирование с помощью действия из основного меню: Run → Edit Configurations

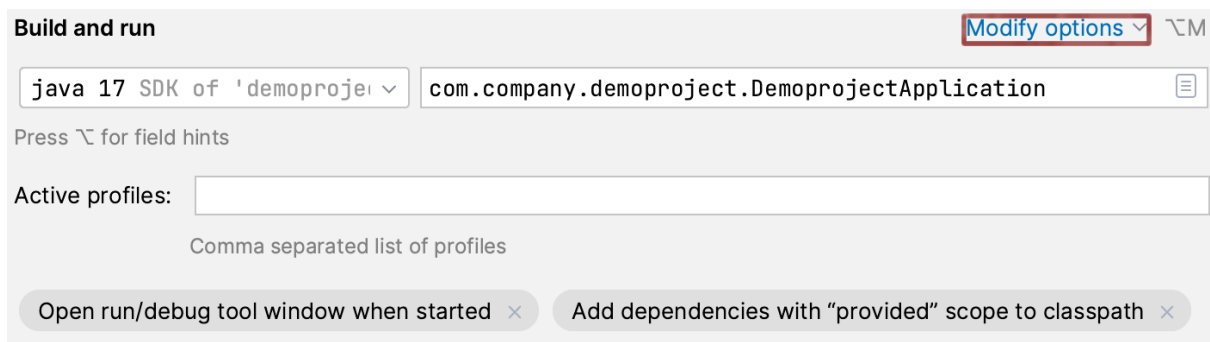
Появится диалоговое окно Run/Debug Configurations:

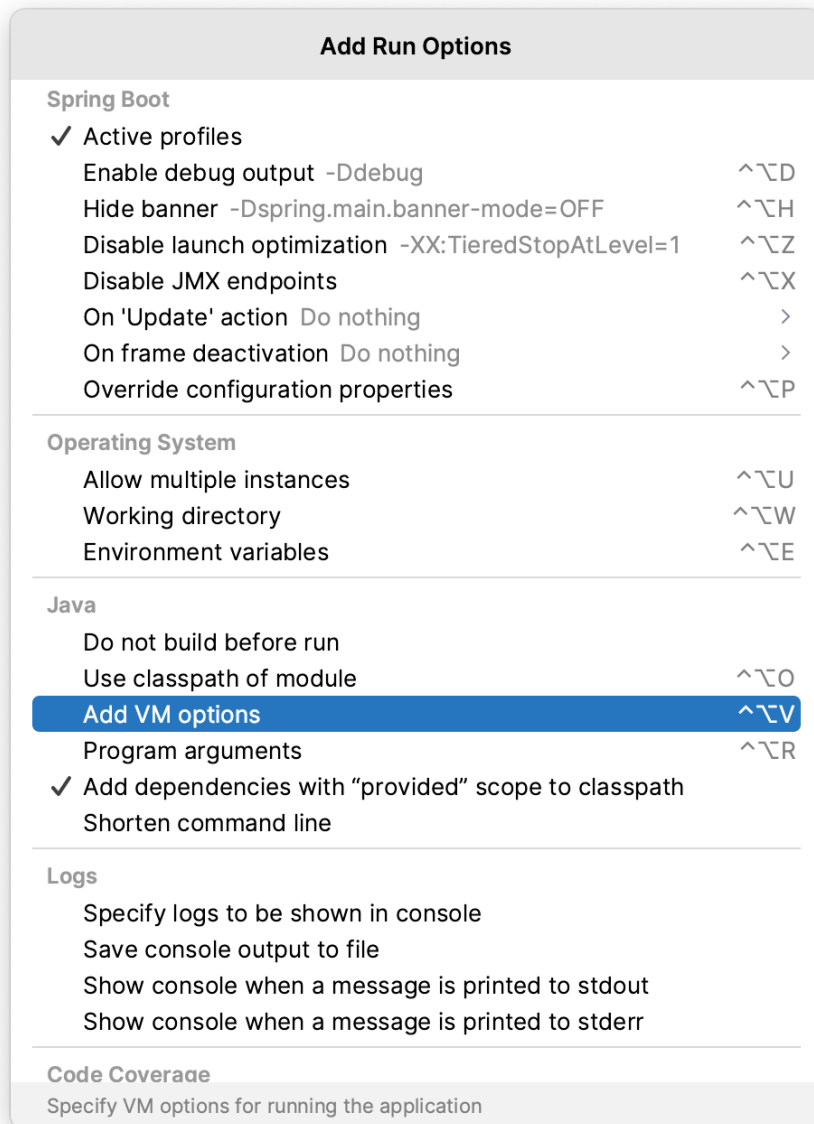


Вам может быть интересно изменить следующие настройки:

- Environment variables — переменные среды, которые должны быть доступны процессу Maven/Gradle и приложению.
- VM options — параметры JVM, передаваемые процессу Maven/Gradle.

Поле VM options по умолчанию не отображается. Чтобы иметь возможность редактировать его, вам нужно выбрать действие Modify options → Add VM options, см. изображения ниже.



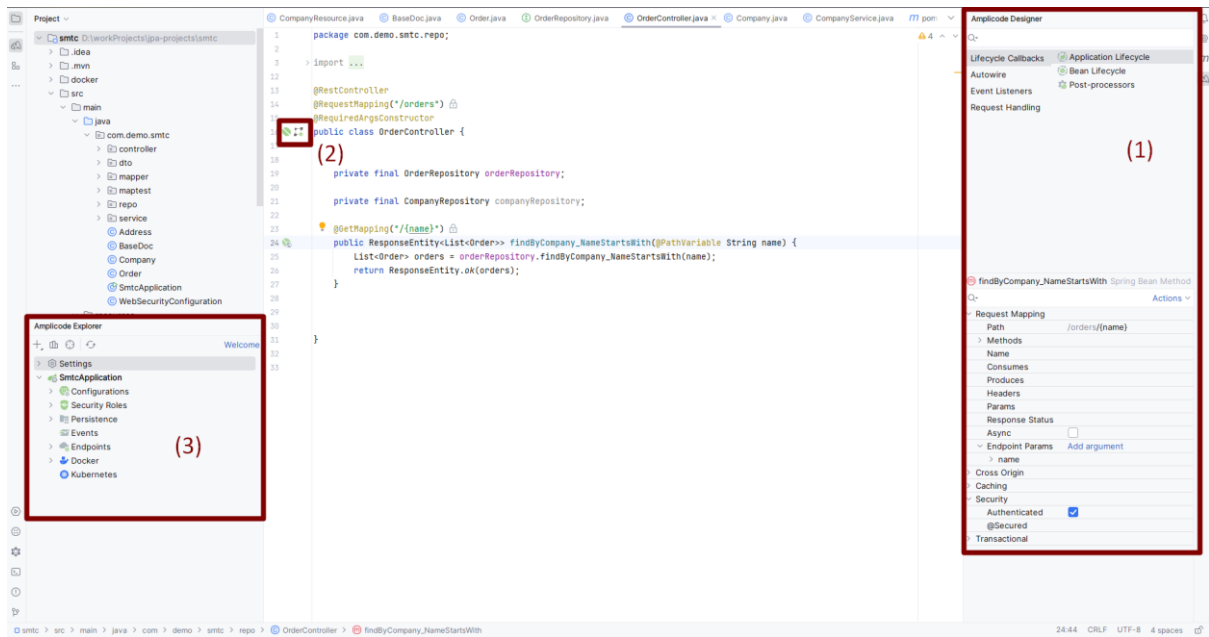


Основные функции плагина

В этом разделе описываются функции и элементы пользовательского интерфейса IDE, характерные для Amplicode.

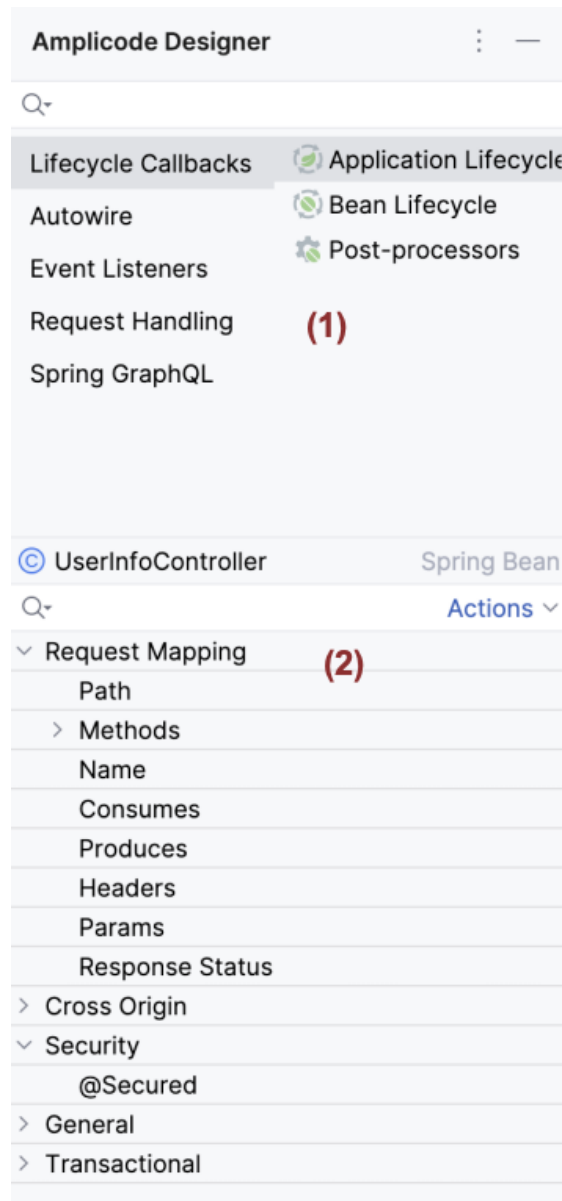
Панели инструментов

После установки Amplicode, загрузки существующего проекта и его открытия вы найдете панель Amplicode Designer (1) в правой части окна IDE. В части редактирования сущности/контроллера/сервиса доступны действия для работы с открытым классом (2). Вы также можете увидеть панель Amplicode Explorer (3) в нижнем левом углу окна IDE.



Amplicode предоставляет следующие панели в окне инструментов Amplicode Designer:

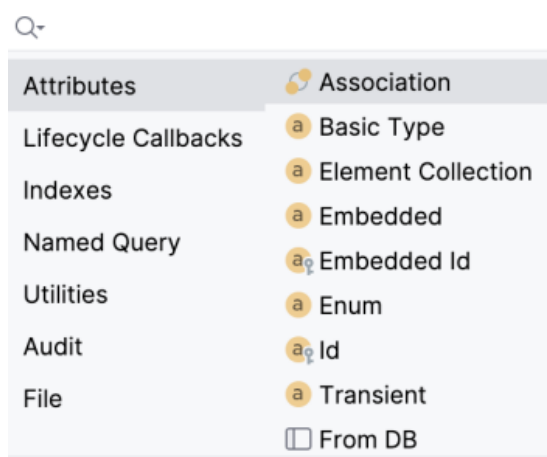
- Палитра (1)
- Инспектор (2)



Палитра

Палитра расположена в окне инструментов Amplicode Designer, в верхней его части. Она может выглядеть по-разному в разных контекстах.

Версия палитры для сущностей JPA показана ниже:



Палитра нацелена на генерацию кода, соответствующего текущему контексту. Для сущности это может быть атрибут или индекс; для репозитория - метод запроса и т. д. Отсюда разный вид Палитры в зависимости от типа редактируемого файла.

Инспектор

Инспектор предназначен для редактирования существующего кода: атрибутов, индексов, запросов и т. д. Все изменения, внесенные в Инспектор, немедленно отражаются в исходном коде. Аналогично, все изменения, внесенные вручную в исходный код, будут отражены в Инспекторе.

Щелкните любой элемент, который вам нужно настроить, и измените требуемые свойства. Обратите внимание, что Инспектор выглядит по-разному в зависимости от того, работаете ли вы с сущностями или с репозиториями. Ниже приведен пример Инспектора, который работает с атрибутами сущности.

```

@Length(min = 2)
@NotNull
@Column(name = "first_name", nullable = false)
private String firstName;

@Length(min = 2)
@NotNull
@Column(name = "last_name", nullable = false)
private String lastName;

@Enumerated(EnumType.STRING)
@Column(name = "specialty", nullable = false)
private Specialty specialty;

```

firstName String

Q- Actions ▾

General

Name	firstName
Type	String
Transient	<input type="checkbox"/>
Mutable	<input checked="" type="checkbox"/>
Fetch type	EAGER

Column

Length	255
Large object	<input type="checkbox"/>
Mandatory	<input checked="" type="checkbox"/>
Unique	<input type="checkbox"/>
Insertable	<input checked="" type="checkbox"/>
Updatable	<input checked="" type="checkbox"/>

Для репозитория инспектор приобретает совершенно другой вид:

DoctorRepository Doctor

Q- Actions ▾

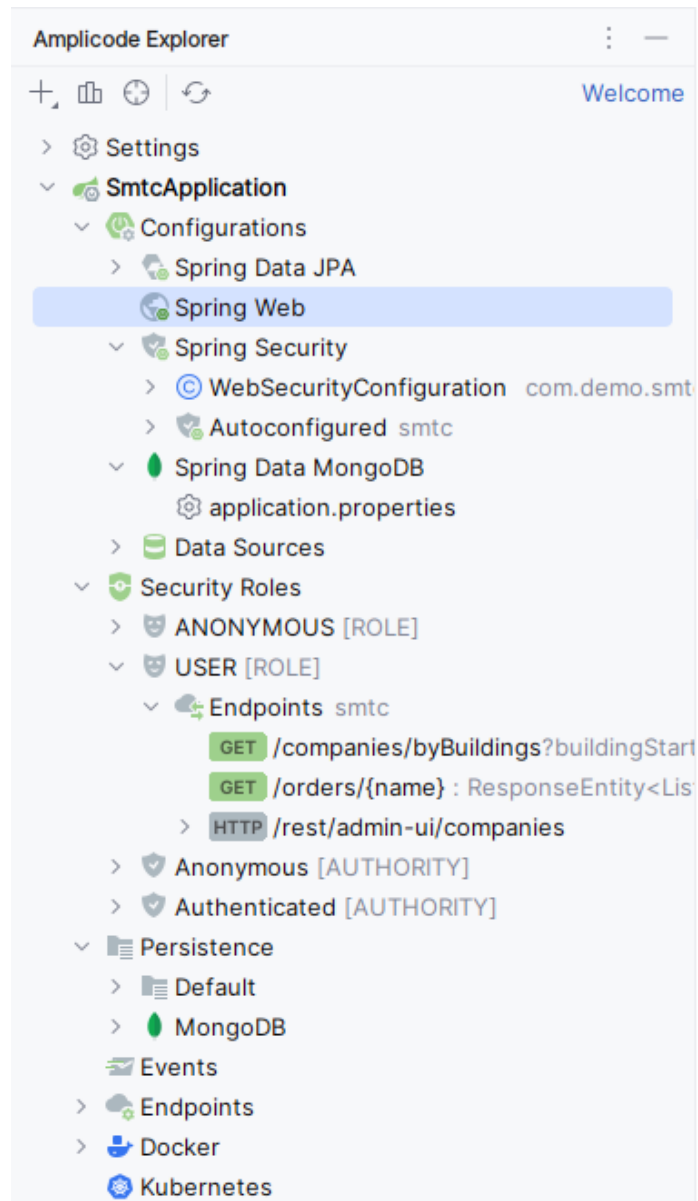
Package	com.company.doctorsdemo.doctor
Name	DoctorRepository
Parent	JpaRepository org.springframework.data.jpa.repository
Specification	<input checked="" type="checkbox"/>

Amplicode Explorer

Окно инструментов Amplicode Explorer предлагает нам комплексный, ориентированный на данные, вид проекта. Вы можете использовать его для различных целей:

- Пройти по модели данных и наблюдать за структурой сущностей, представленной иерархически. Эта функция позволяет легко переходить к сущностям, ссылающимся на текущую, а также к сущностям, на которые ссылается текущая сущность. Это чрезвычайно полезная функция, особенно для новичков в существующем проекте с большим графом сущностей или для рецензентов кода, у которых мало времени, чтобы понять, как разработана модель данных.
- Создать объекты, связанные с данными: сущности, конвертеры JPA, пользовательские типы Hibernate, репозитории Spring Data и журнал изменений Liquibase.
- Наблюдать за связанными репозиториями Spring Data, DTO и проекциями для каждой сущности.
- Редактировать дополнительные артефакты проекта, такие как соединения с базой данных, эндпоинты и многое другое.

Ниже вы можете видеть верхний и нижний фрагменты панели Amplicode Explorer рядом с частично развернутыми разделами.



Работа с JPA сущностями

Как мы уже упоминали, панель Amplicode Designer содержит две подпанели: Палитру и Инспектор.

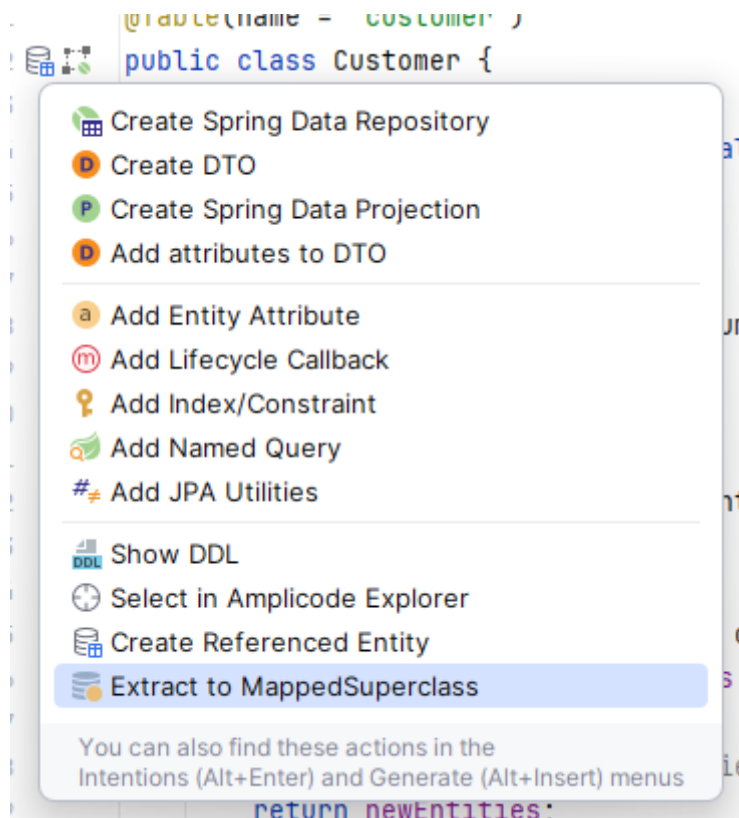
Палитра предоставляет действия генерации кода для следующих элементов:

- Attributes
- Lifecycle Callbacks
- Indexes
- Named Query
- Utilities – Equals/HashCode/ToString
- Audit

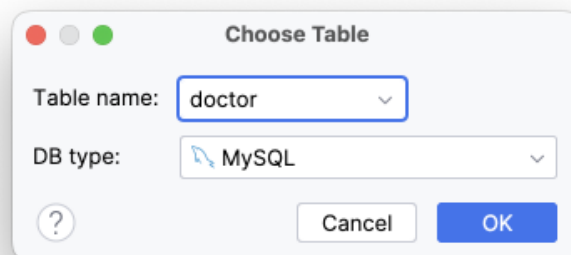
Мы рассмотрим различные примеры использования Палитры и Инспектора по мере того, как будем изучать, как работать с сущностями и их атрибутами.

Действия на боковой панели

На боковой панели доступны дополнительные действия для сущности



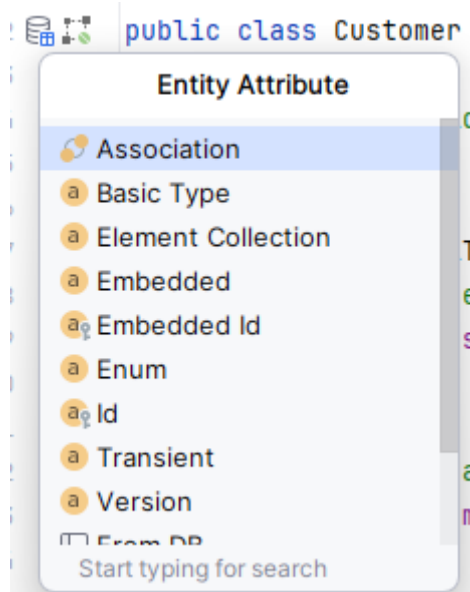
Show DDL отображает информацию, связанную с базой данных для текущей открытой сущности.



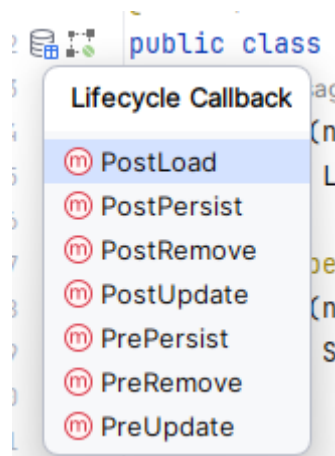
С помощью этого всплывающего окна вы также можете выбрать другую базу данных для сущности.

Функция Select in Amplicode Explorer автоматически отображает текущий выбранный объект на панели Amplicode Explorer.

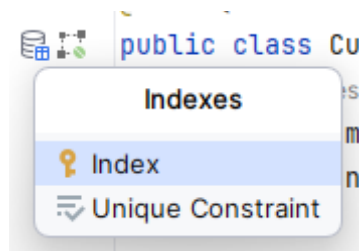
Add Entity Attribute, как следует из названия, предоставляет средства для создания атрибутов для текущей сущности (альтернатива палитре).



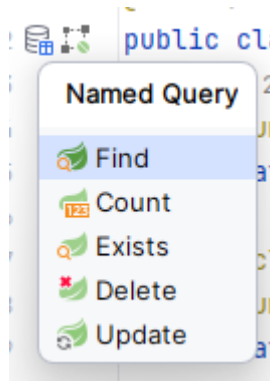
Add Lifecycle Callback Method, опять же, как очевидно из названия, предоставляет интерфейс для создания методов обратного вызова жизненного цикла.



С помощью функции Add Index/Constraint вы можете добавлять индексы или уникальные ограничения в базу данных.

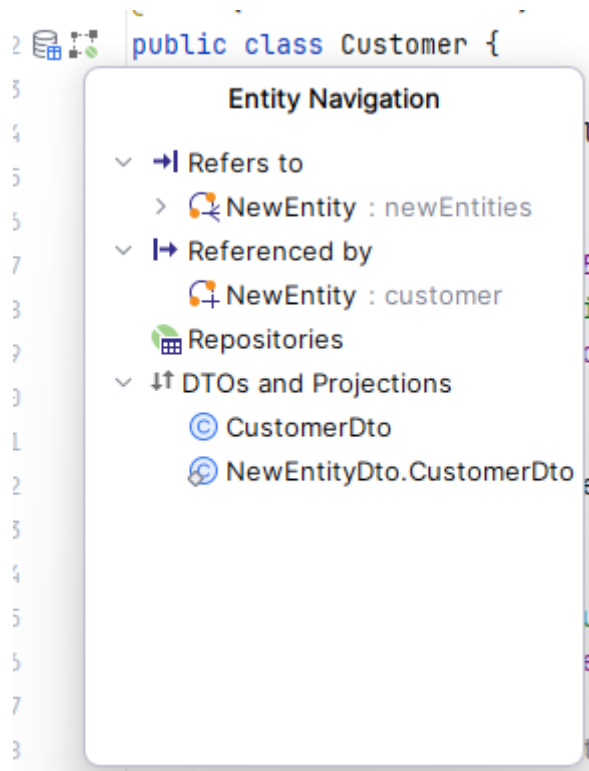


Действие Add Named Query предоставляет интерфейс для создания различных типов именованных запросов.



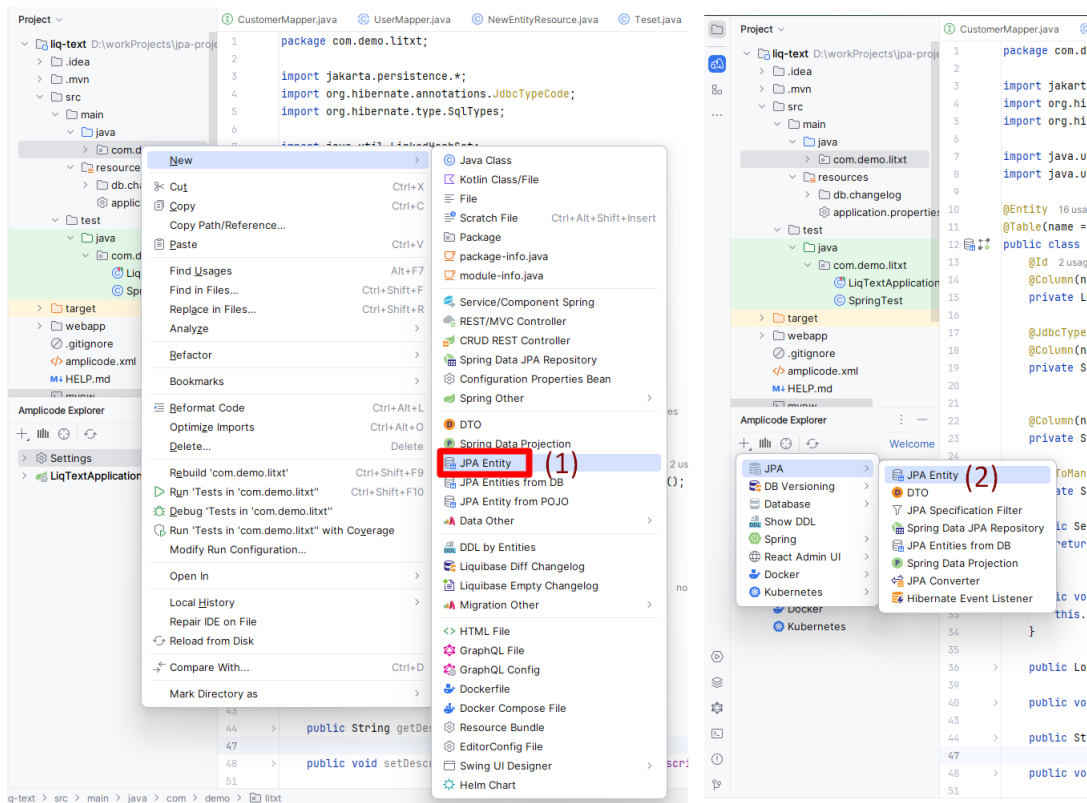
Действия под названием Create Spring Data Repository / Create DTO/ Create Spring Data Projection дает возможность создать больше репозиториях/DTO/проекций.

Значок References отображает ссылки на текущий выбранный объект в исходном коде.

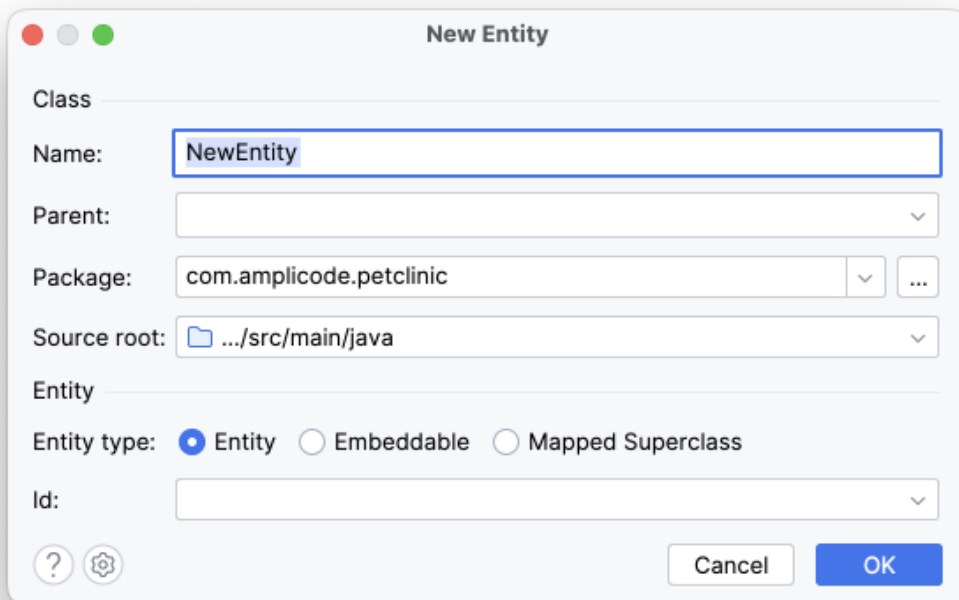


Создание сущности

Чтобы создать новую сущность JPA, щелкните правой кнопкой мыши по нужной папке и выберите New → JPA Entity (1). Также вы можете создать новую сущность из Amplicode Explorer (2):



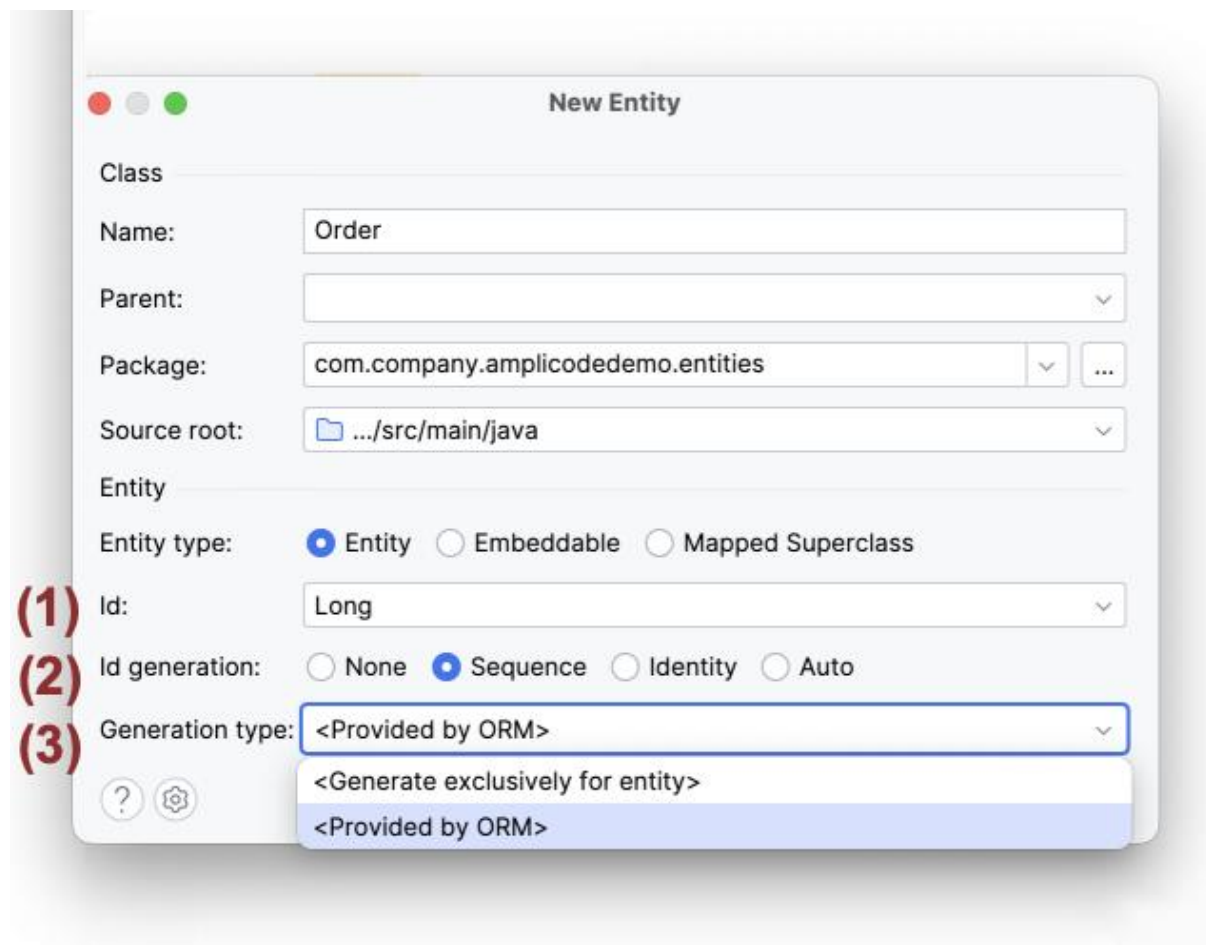
После этого появится окно New Entity:



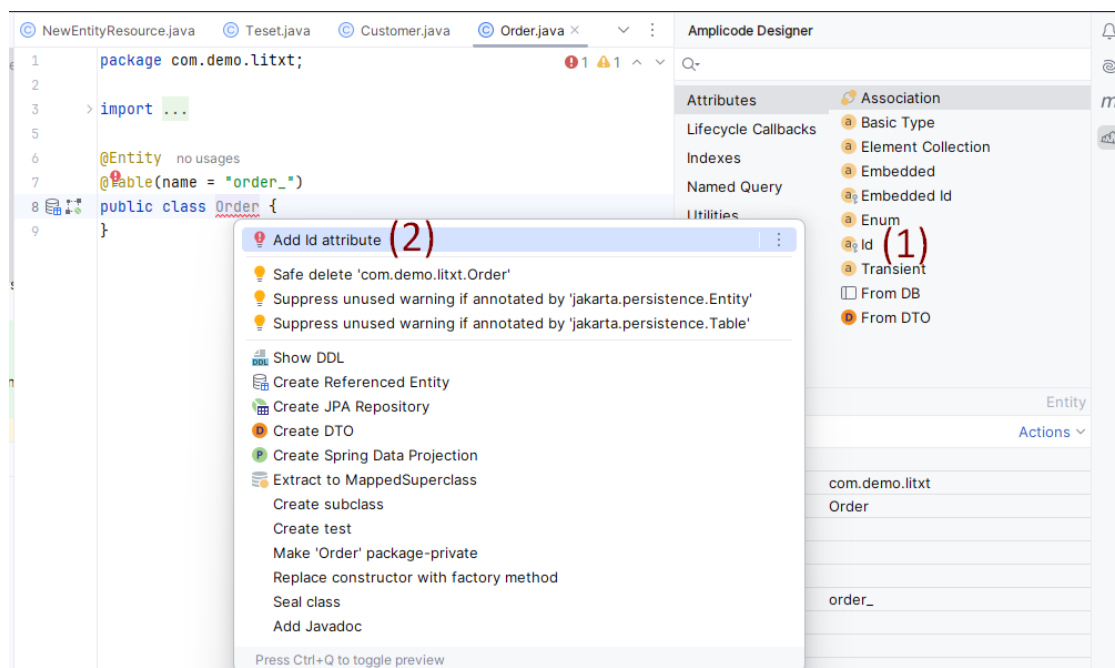
ID и выбор стратегии генерации

Согласно спецификации JPA, для определения сущности требуется атрибут ID. Amplicode позволяет сгенерировать этот атрибут и выбрать тип (1) и стратегию

генерации (2). Также можно указать, какую последовательность использовать для стратегии генерации последовательности (3).

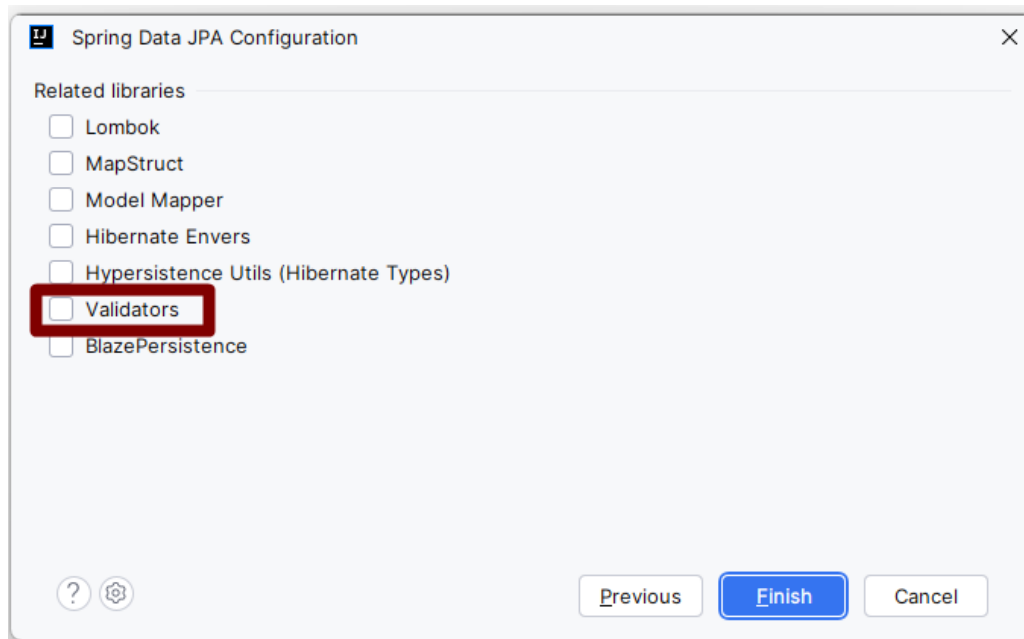


Кроме того, вы можете создать атрибут id с помощью палитры (1) или с помощью быстрого исправления (Alt+Enter/⇧⌘/значок лампочки) (2).



Валидация полей

Amplicode предоставляет возможность подключить в проект библиотеку Hibernate Validator. Она позволяет нам определять модель метаданных и API для проверки сущностей и методов. Для подключения откройте Amplicode Explorer, разверните дерево проекта, в разделе Configurations выберите Spring Data JPA, кликните правой кнопкой мыши и нажмите Configure, на этапе Related libraries выберите опцию Validators и нажмите Finish.



Инспектор содержит раздел с проверками, которые можно применить к атрибуту. Чтобы он появился, выберите атрибут в исходном коде:

```
@Length(min = 2)
@NotNull
@Column(name = "first_name", nullable = false)
private String firstName;

@Length(min = 2)
@NotNull
@Column(name = "last_name", nullable = false)
private String lastName;

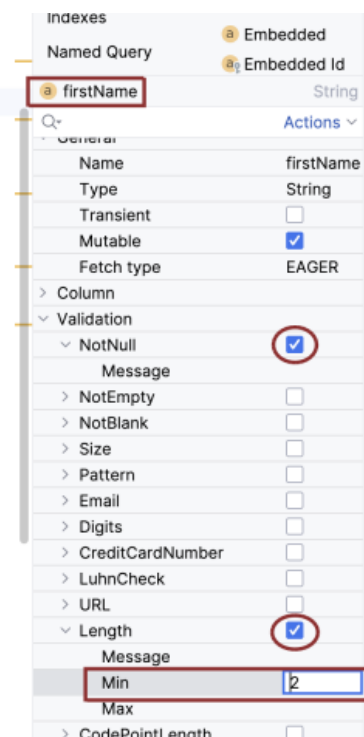
@Enumerated(EnumType.STRING)
@Column(name = "specialty", nullable = false)
private Specialty specialty;

public Specialty getSpecialty() {
    return specialty;
}

public void setSpecialty(Specialty specialty) {
    this.specialty = specialty;
}

public String getLastName() { return lastName; }

public void setLastName(String lastName) { this.lastName = lastName; }
```



Обратите внимание, что свойства и проверки, предлагаемые Инспектором, зависят от типа данных атрибута. Например, набор свойств и проверок, предлагаемых для атрибута Integer, будет отличаться от атрибута String и т. д.

Дополнительную информацию можно найти, перейдя по ссылкам ниже:

- [Hibernate Validator \(Hibernate Validator documentation\)](#)
- [Spring support for Bean Validation \(Spring documentation\)](#)

JPA конвертеры

Amplicode помогает вам сгенерировать пустой фрагмент кода для JPA Converter через Inspector:

Допустим, у нас есть атрибут isCompleted (логическое значение) в сущности Order.

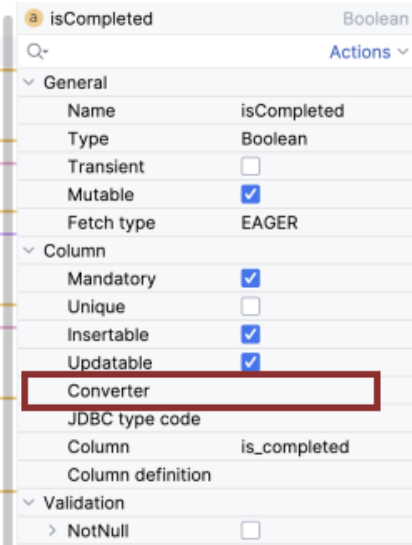
```
@Column(name = "is_completed", nullable = false)
private Boolean isCompleted = false;

public Boolean getIsCompleted() {
    return isCompleted;
}

public void setIsCompleted(Boolean isCompleted) {
    this.isCompleted = isCompleted;
}

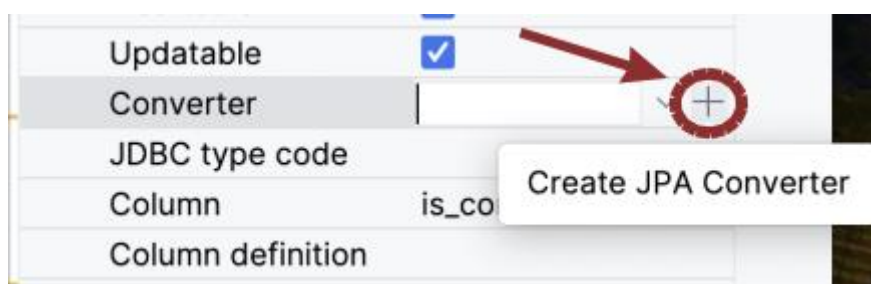
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}
```

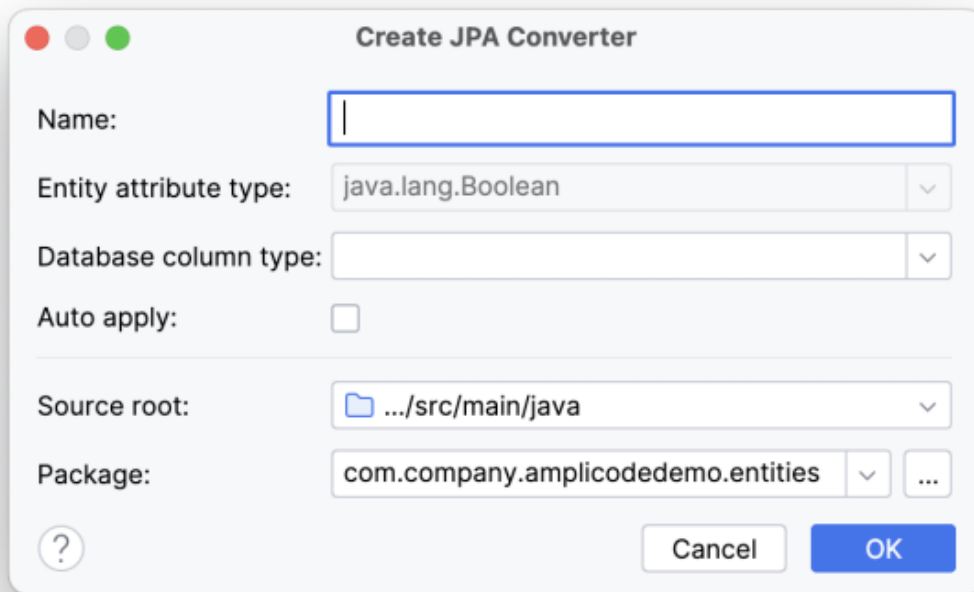


isCompleted		Boolean
Actions		
General		
Name	isCompleted	
Type	Boolean	
Transient	<input type="checkbox"/>	
Mutable	<input checked="" type="checkbox"/>	
Fetch type	EAGER	
Column		
Mandatory	<input checked="" type="checkbox"/>	
Unique	<input type="checkbox"/>	
Insertable	<input checked="" type="checkbox"/>	
Updatable	<input checked="" type="checkbox"/>	
Converter		
JDBC type code		
Column	is_completed	
Column definition		
Validation		
NotNull	<input type="checkbox"/>	

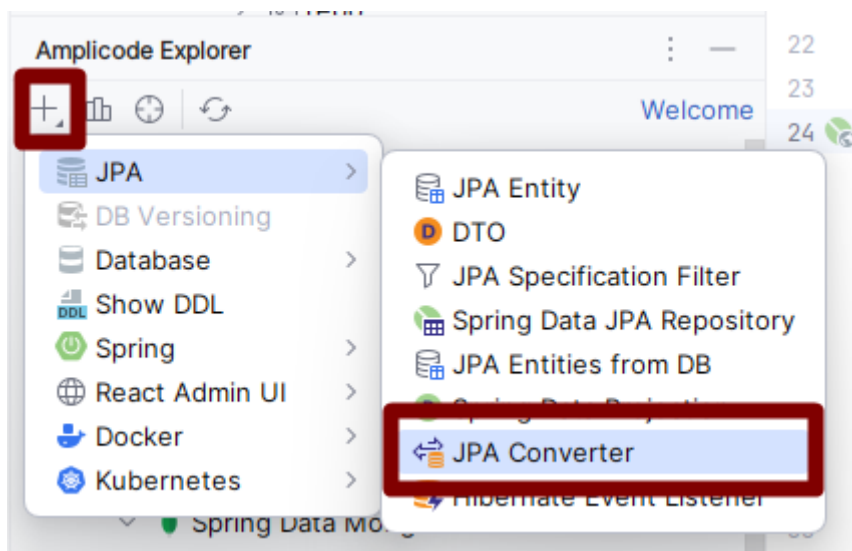
Наведите курсор на этот атрибут в коде, затем нажмите на пиктограмму «Плюс» в Инспекторе:



Откроется следующее окно:

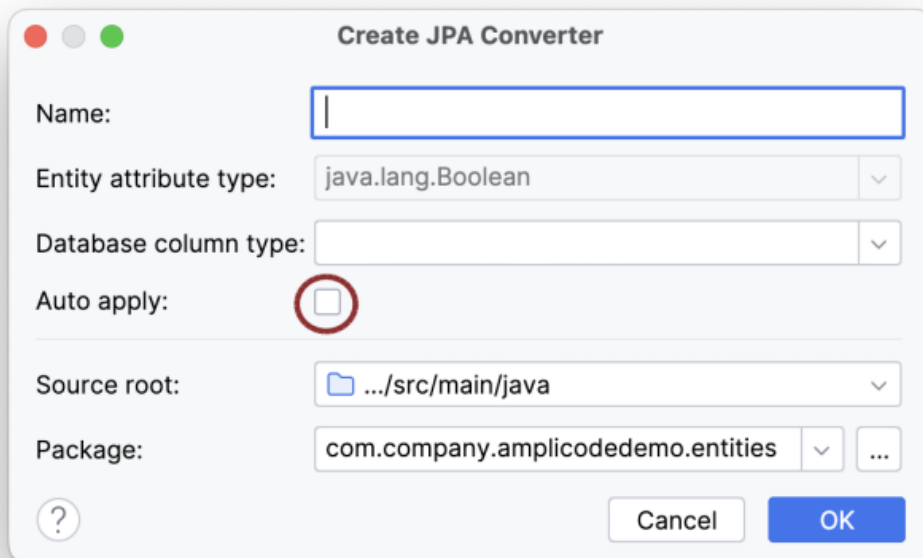


Вы также можете создать его через Amplified Explorer. Просто нажмите на кнопку Plus:



В окне Create JPA Converter можно настроить имя класса, тип атрибута сущности и тип столбца базы данных.

Вы также можете определить, будет ли JPA-конвертер автоматически применяться или нет.

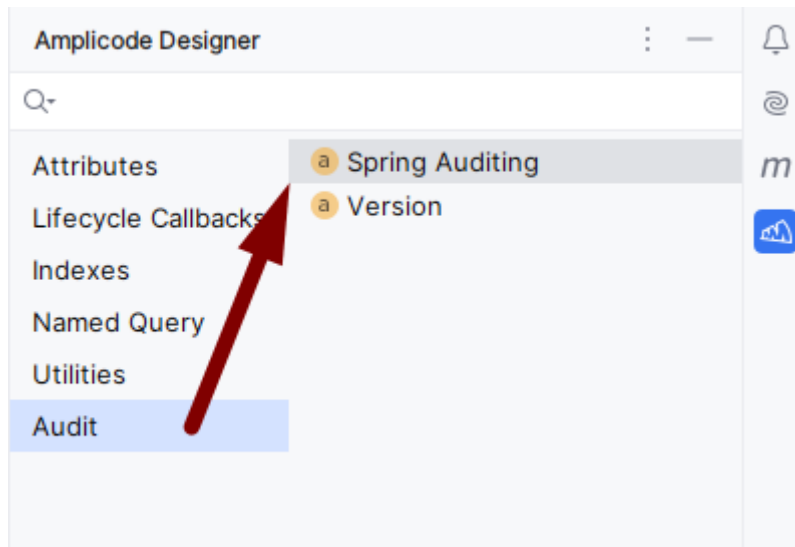


Поддержка аудита сущностей

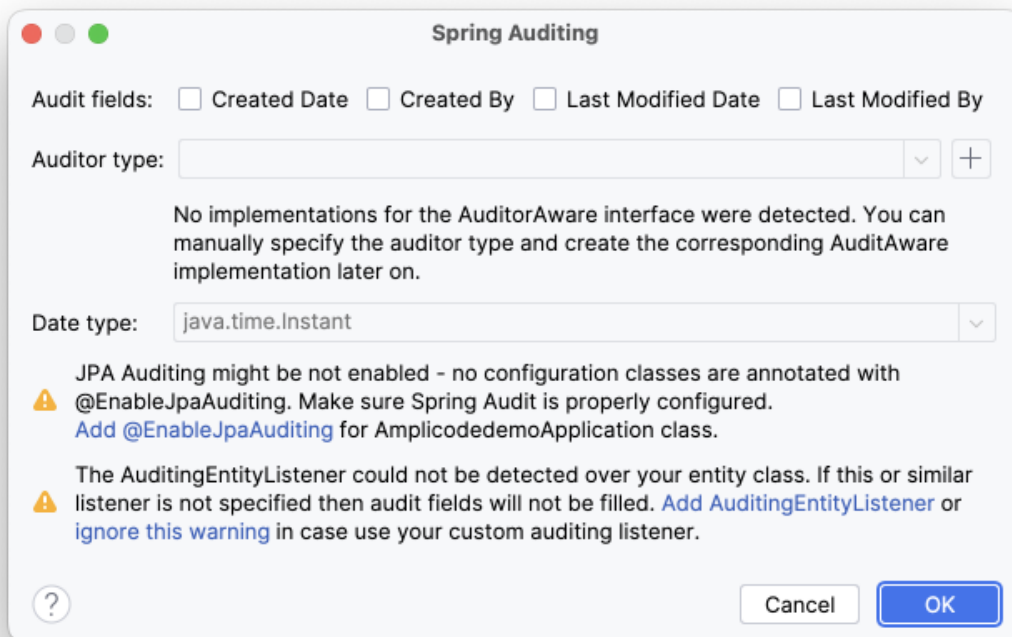
Наличие аудита в большом приложении является критически важным аспектом. С помощью AmpliCode вы можете без труда включать часто используемые поля аудита, используя аннотации, такие как `@CreatedBy`, `@CreatedDate`, `@LastModifiedBy` и `@LastModifiedDate`. Более того, AmpliCode уведомит вас, если вы забудете добавить аннотацию `@EnableJpaAuditing` в свою конфигурацию или если `AuditingEntityListener` не добавлен в текущую сущность.

Подробнее об аудите данных Spring можно прочитать в [документации Spring](#).

Для добавления аудита откройте дизайнер сущности и выберите действие Audit → Spring Auditing.



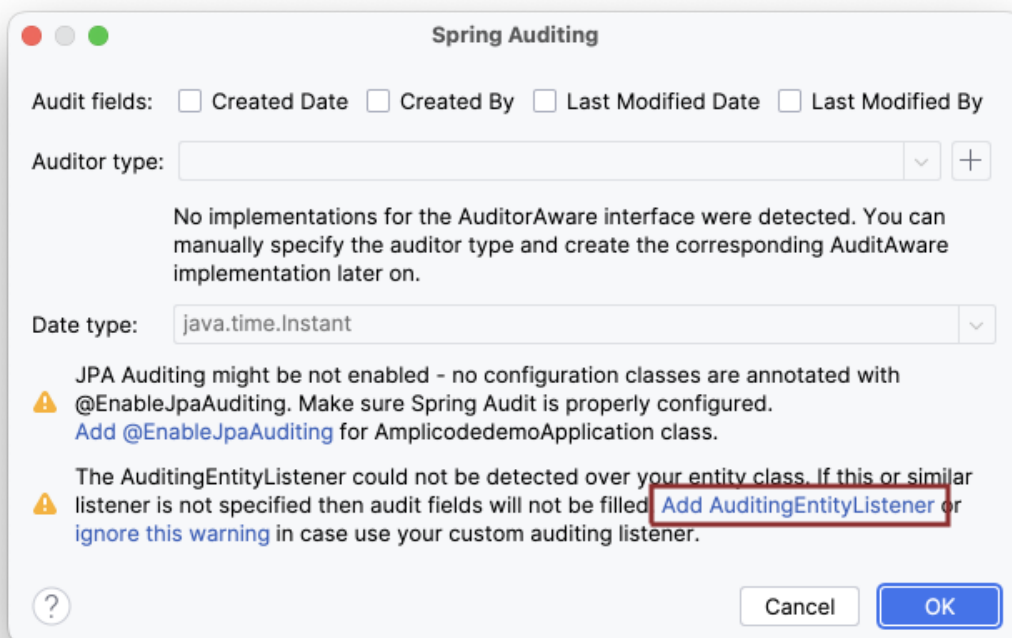
Мы получим предупреждение об отсутствующих аннотациях.



Кликните на предложение по добавлению аннотации и она автоматически добавится в главный класс приложения.

```
Order.java AmplicodedemoApplication.java x
package com.company.amplicodedemo;
import ...
@EnableJpaAuditing
@SpringBootApplication
@ConfigurationPropertiesScan
public class AmplicodedemoApplication {
    public static void main(String[] args)
}
```

Чтобы добавить `AuditingEntityListener`, достаточно нажать на ссылку, представленную в том же окне.



Работа с MongoDB документами

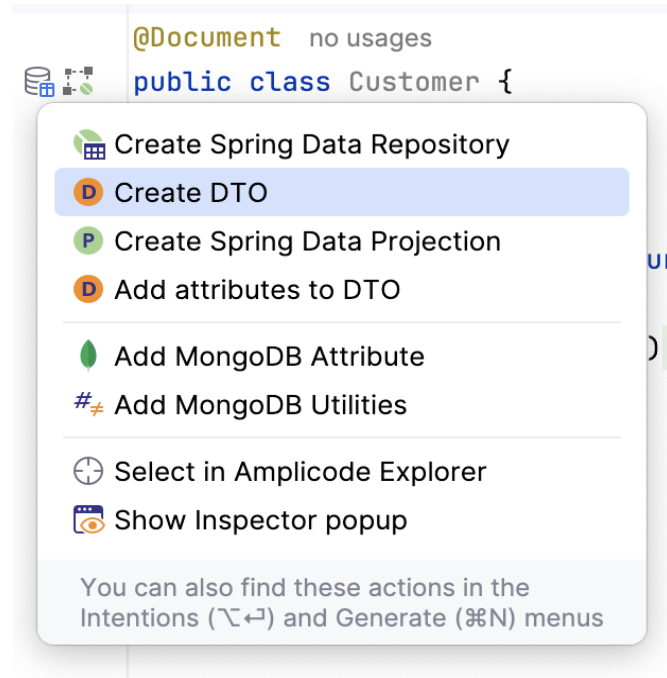
Палитра в MongoDB документе предоставляет действия генерации кода для следующих элементов:

- Attributes
- Utilities – Equals/HashCode/ToString
- Audit

Мы рассмотрим различные примеры использования Палитры и Инспектора по мере того, как будем изучать, как работать с сущностями и их атрибутами.

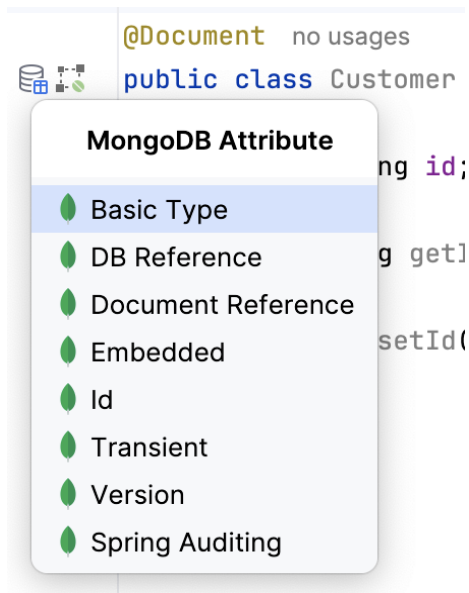
Действия на боковой панели

На боковой панели доступны дополнительные действия для MongoDB документа



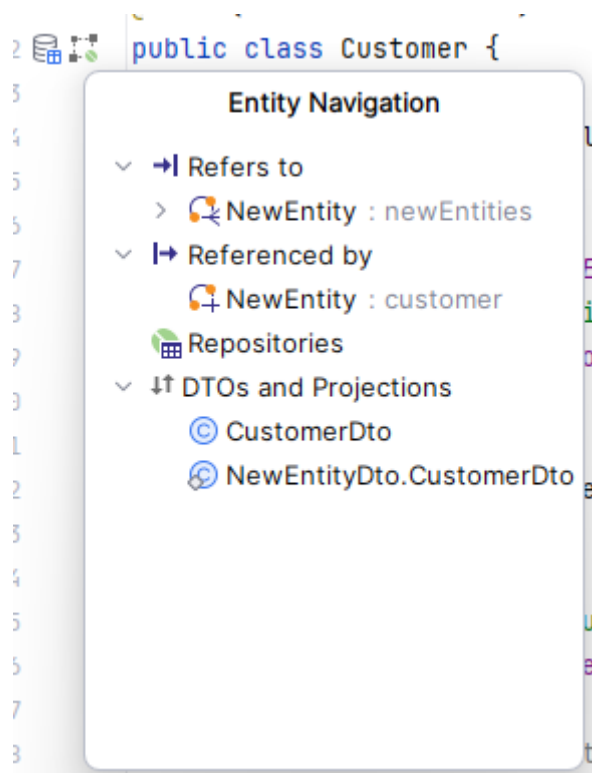
Функция Select in Amplicode Explorer автоматически отображает текущий выбранный объект на панели Amplicode Explorer.

Add MongoDB Attribute, как следует из названия, предоставляет средства для создания атрибутов для текущего MongoDB документа (альтернатива палитре).



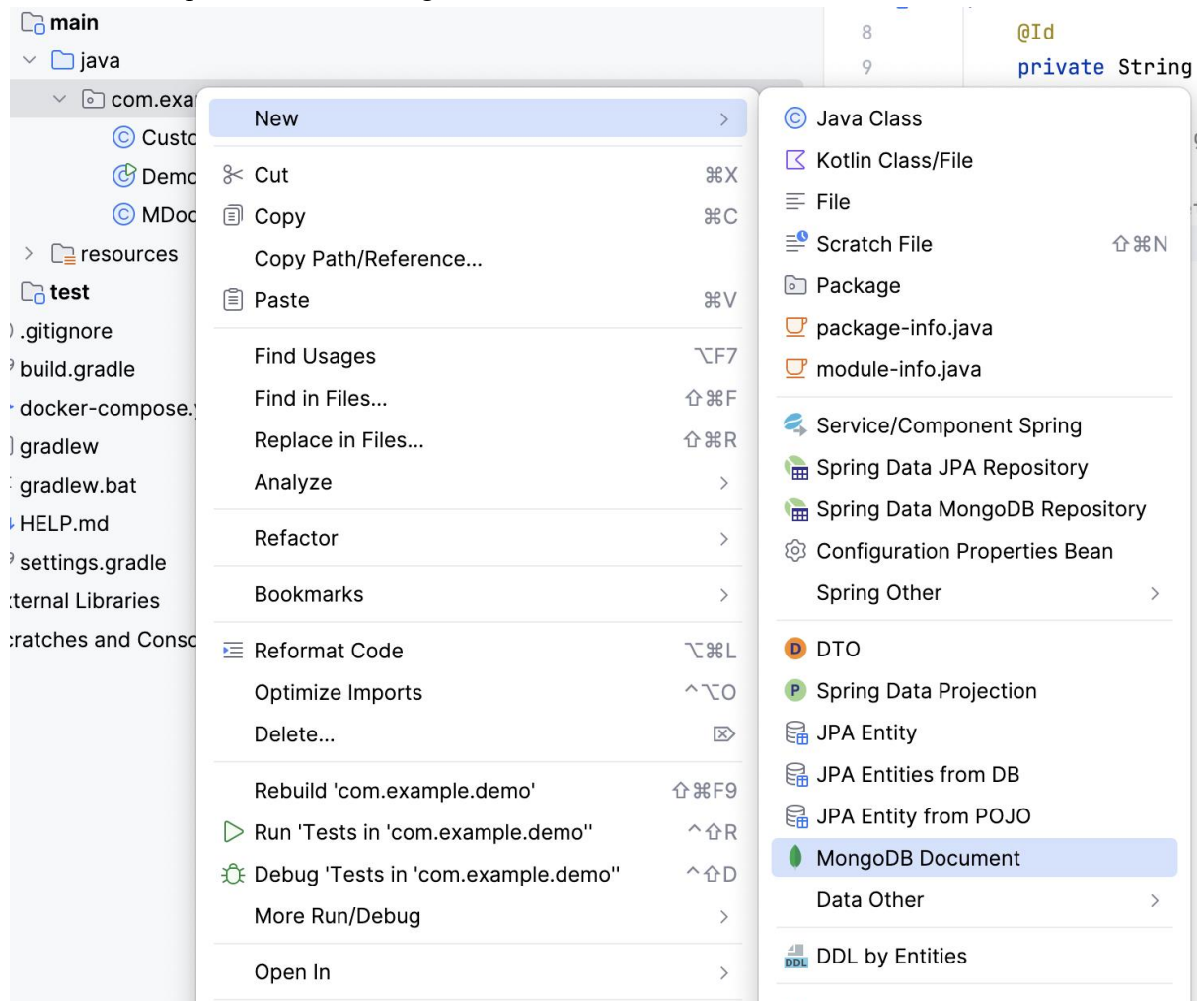
Действия под названием Create Spring Data Repository / Create DTO/ Create Spring Data Projection дают возможность создать репозитории/DTO/проекции.

Значок References отображает ссылки на текущий выбранный объект в исходном коде.

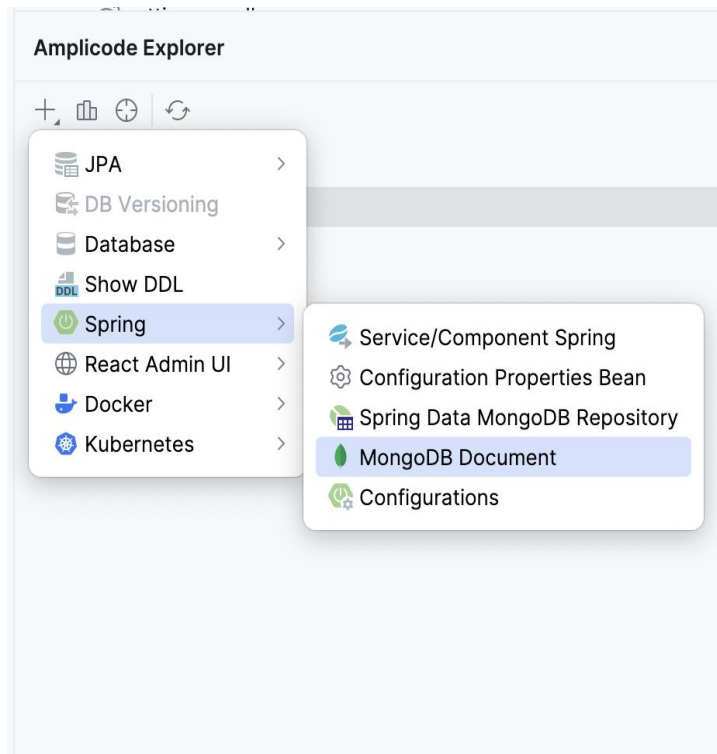


Создание MongoDB документа

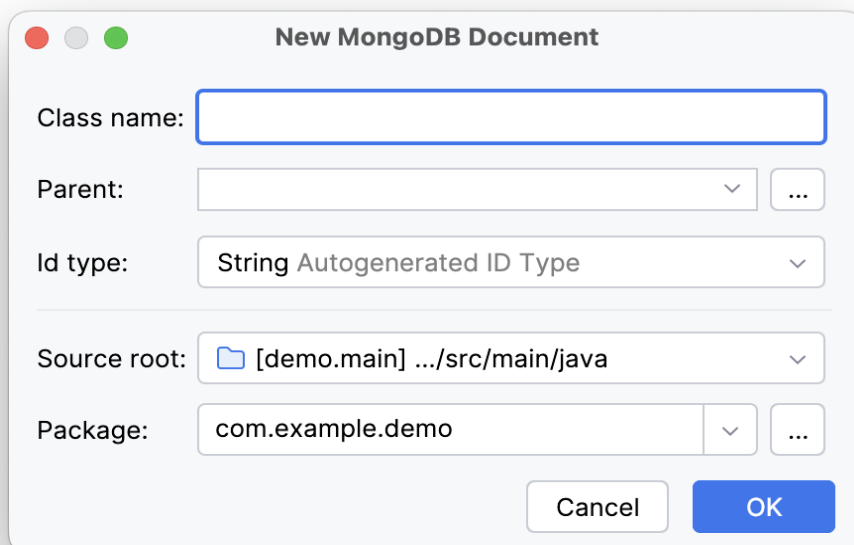
Чтобы создать новый MongoDB документ, щелкните правой кнопкой мыши по нужной папке и выберите New → MongoDB document.



Также вы можете создать новый документ из Amplicode Explorer:

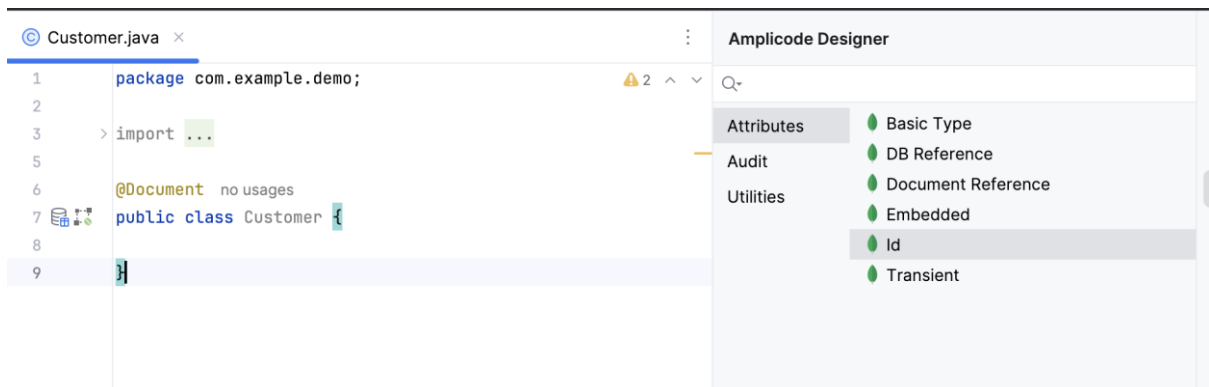


После этого появится окно New MongoDB Document



Для определения MongoDB документа требуется атрибут ID. Amplicode позволяет выбрать тип атрибута и возможность его автоматической генерации.

Кроме того, вы можете создать атрибут id с помощью палитры или через действие Add MongoDB Attribute описанного выше.



Поддержка аудита MongoDB документов

С помощью Amplicode вы можете без труда включать часто используемые поля аудита, используя аннотации, такие как `@CreatedBy`, `@CreatedDate`, `@LastModifiedBy` и `@LastModifiedDate` для MongoDB документов. Более того, Amplicode уведомит вас, если вы забудете добавить аннотацию `@EnableMongoAuditing` в свою конфигурацию.

Подробнее об аудите данных Spring можно прочитать в [документации Spring](#).

Настройка аудита для MongoDB документов аналогична настройке аудита для JPA сущностей и возможно обратиться к главе “Поддержка аудита сущностей” для JPA.

Spring Data репозитории

Spring Data JPA является частью более крупного семейства Spring Data, которое позволяет легко и быстро внедрять репозитории на основе JPA. Amplicode предоставляет понятный пользовательский интерфейс для создания репозитория, проекций, запросов и т. д. как для JPA сущностей, так и для MongoDB документов.

Больше информации о репозиториях можно найти в [документации Spring](#).

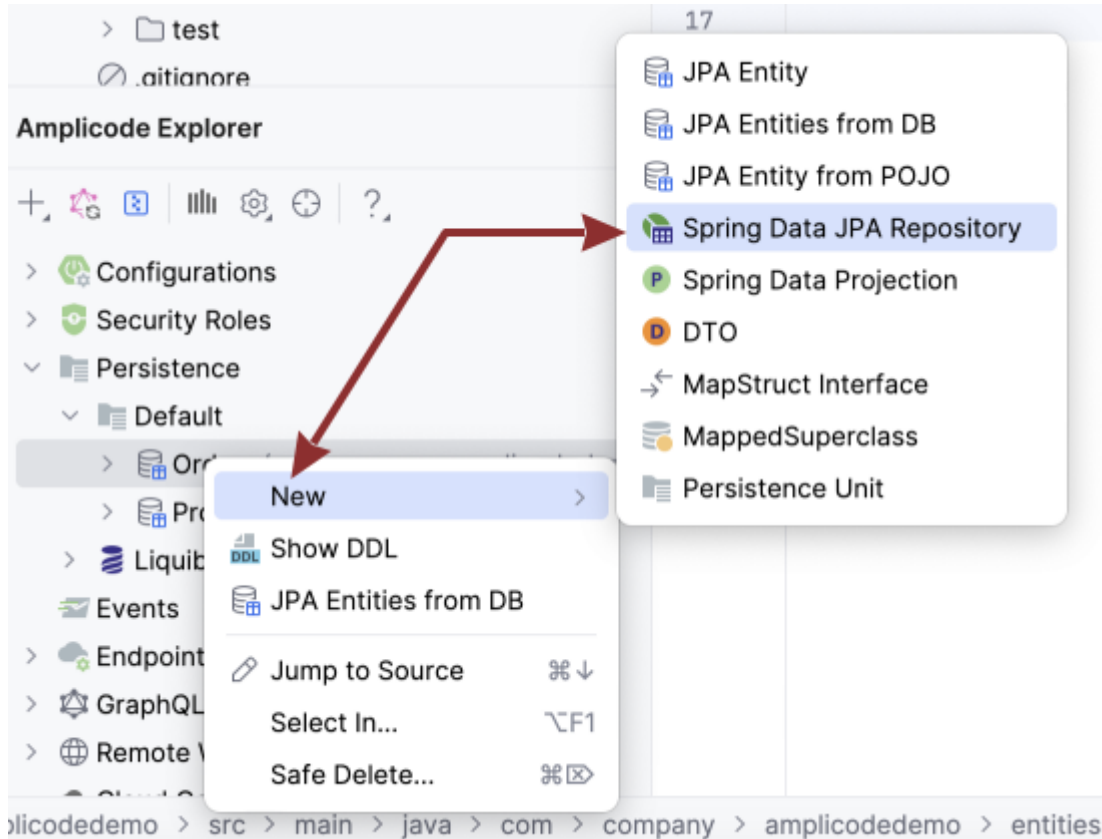
Создание репозитория

Amplicode предоставляет различные способы создания репозитория, чтобы сделать работу с объектами JPA и MongoDB документами более удобной для большинства пользователей.

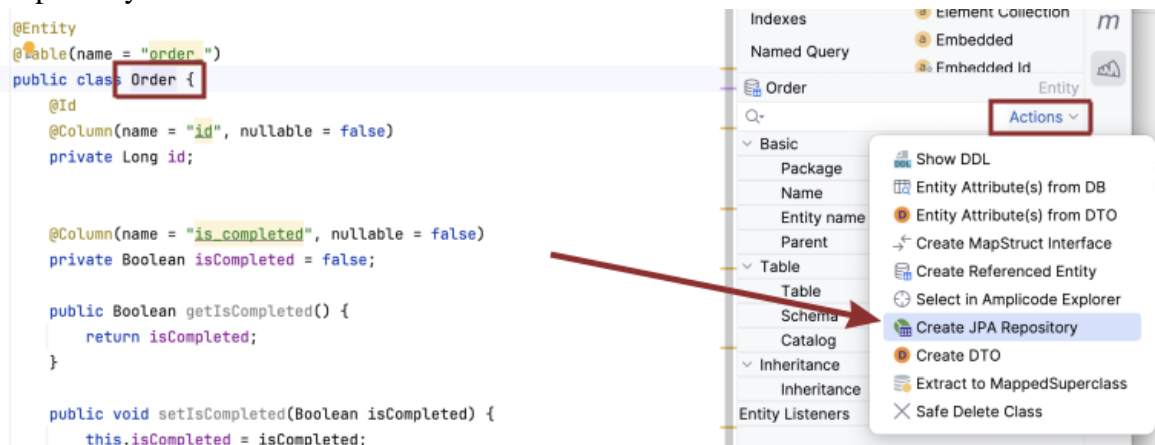
Нужно знать, что репозиторий Spring Data JPA можно создать только для сущности с полем, аннотированным `@Id`. Ниже перечислены все возможные способы (их четыре) создания репозитория в проекте.

- Щелкните правой кнопкой мыши сущность JPA в дереве проекта, затем выберите New → Spring Data JPA Repository. (скриншот)
- Используйте пиктограмму + на панели Amplicode Explorer, затем выберите JPA → Spring Data JPA Repository. (скриншот)

- Откройте Persistence в Amplicode Explorer, выберите сущность, щелкните ее правой кнопкой мыши и выберите New → Spring Data JPA Repository.

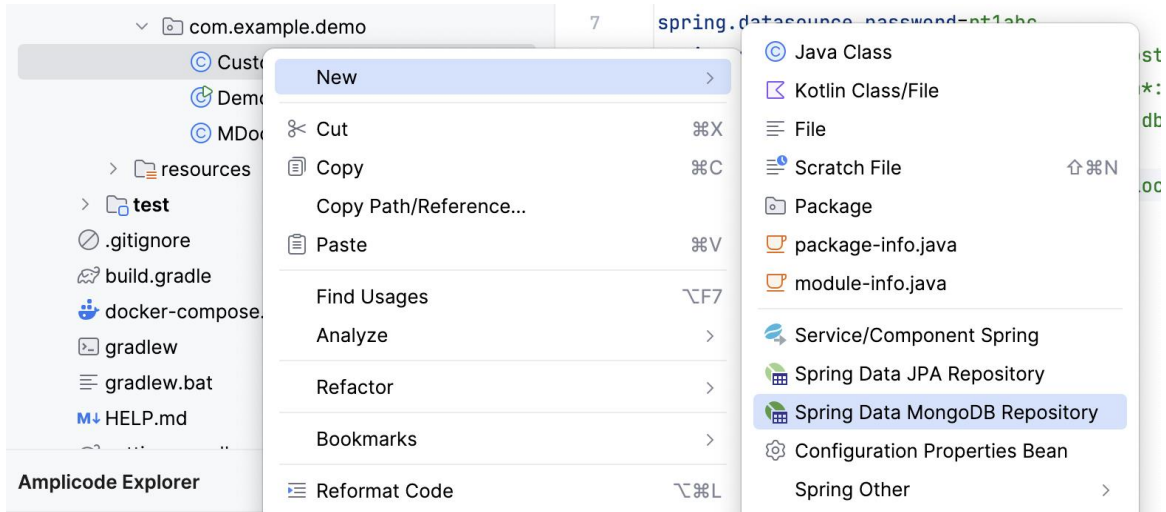


- Дважды щелкните сущность в дереве проекта, переключитесь на панель Amplicode Designer и в Инспекторе используйте раскрывающийся список Actions. Курсор должен быть на имени класса в окне Editor. Выберите Create JPA Repository.

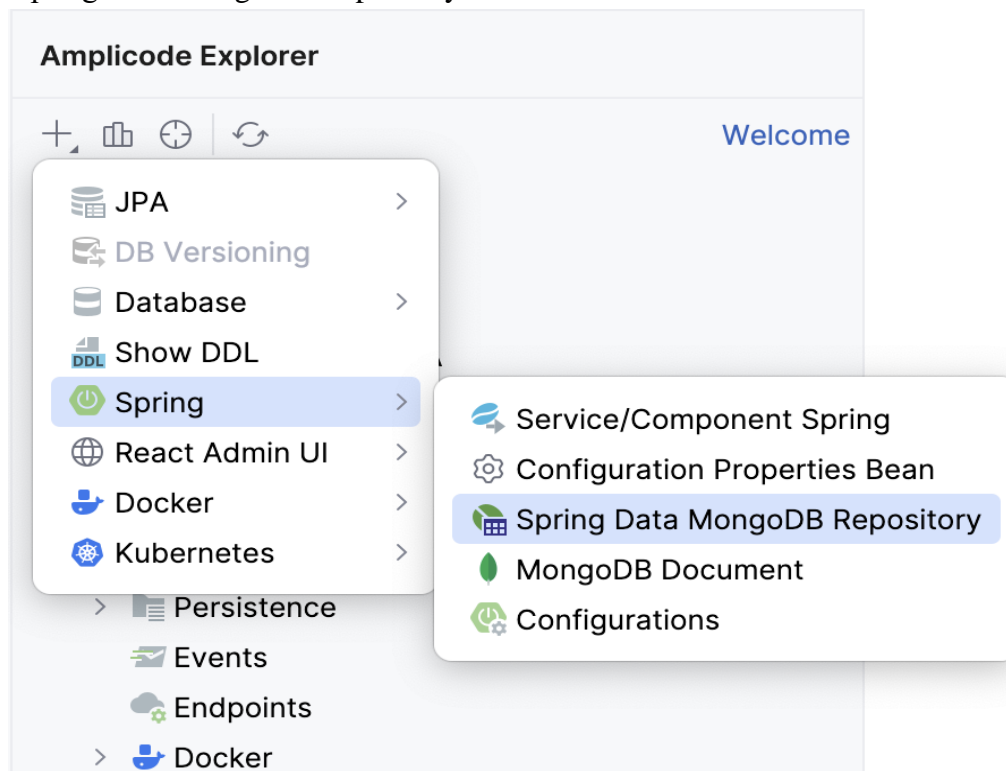


Аналогично, Spring Data MongoDB репозитории можно создать только для документов с полем, аннотированным @Id. Ниже перечислены все возможные способы (аналогично Spring Data JPA репозитория) создания Spring Data MongoDB репозитория в проекте.

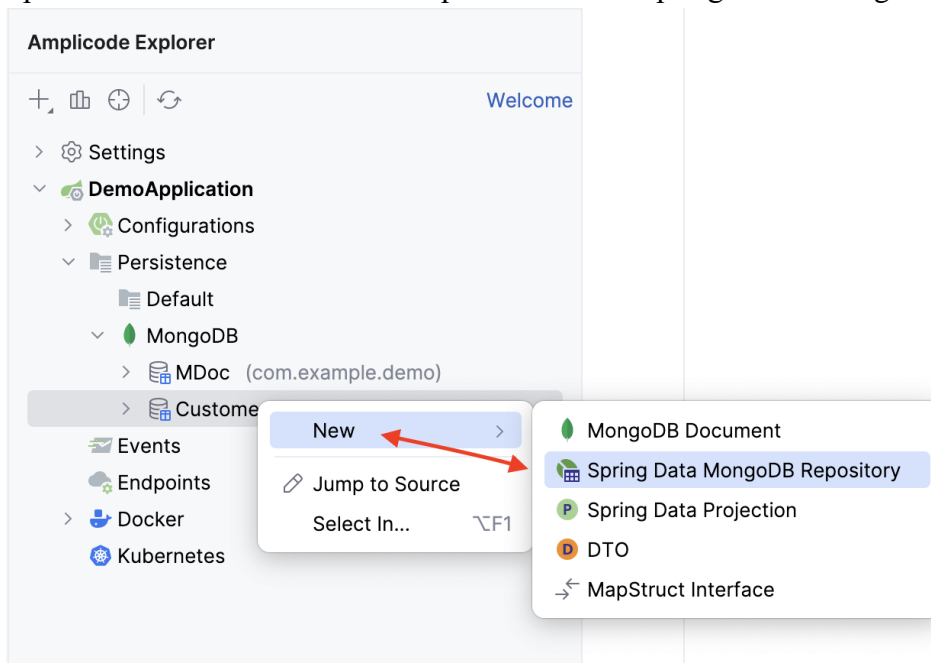
- Щелкните правой кнопкой мыши на MongoDB документ в дереве проекта, затем выберите **New** → **Spring Data MongoDB Repository**.



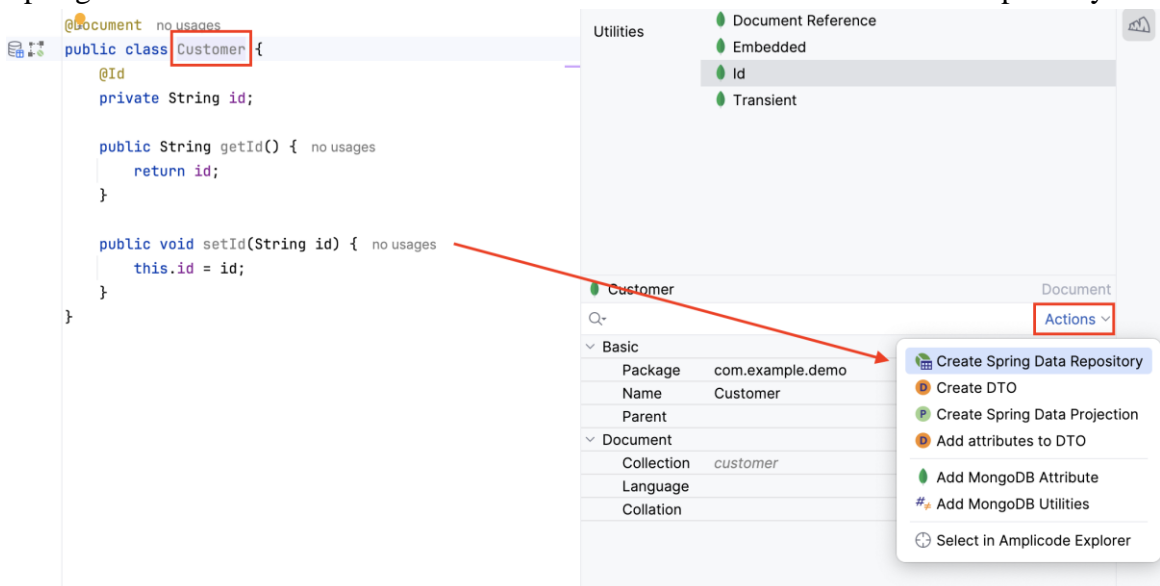
- Используйте пиктограмму + на панели Amplicode Explorer, затем выберите **Spring** → **Spring Data MongoDB Repository**.



- Откройте Persistence в Amplicode Explorer, выберите сущность, щелкните на ней правой кнопкой мыши и выберите New → Spring Data MongoDB Repository.



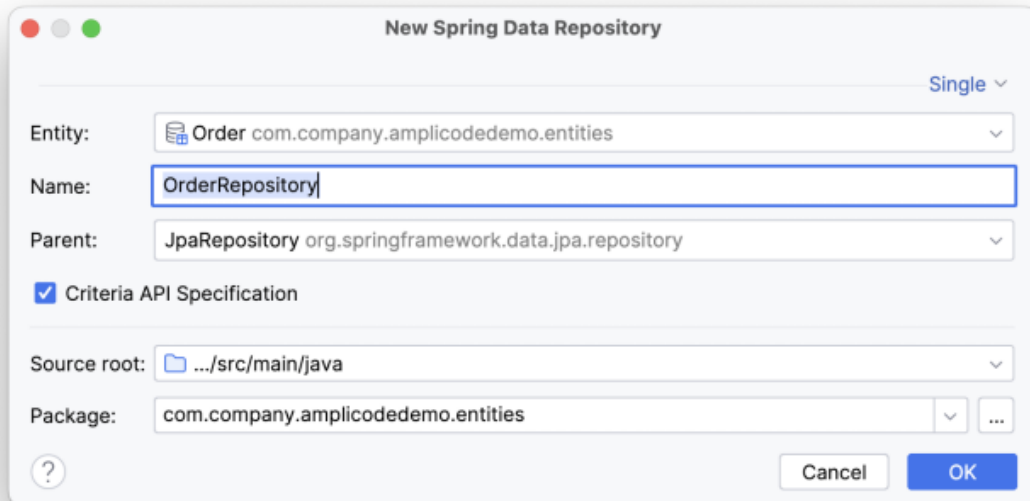
- Дважды щелкните сущность в дереве проекта, переключитесь на панель Amplicode Designer и в Инспекторе используйте раскрывающийся список Actions. Курсор должен быть на имени класса в окне Editor. Выберите Create Spring Data Repository.



В окне New Spring Data Repository вы можете задать:

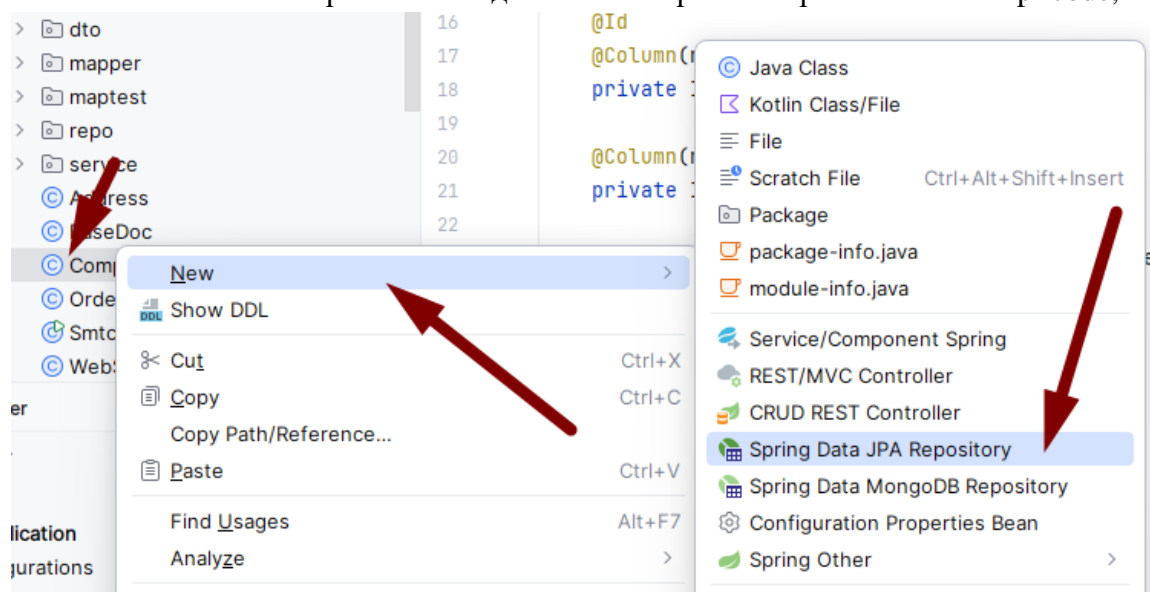
- Сущность/MongoDB документ, для которой будет создан репозиторий
- Имя класса
- Родительский класс для репозитория. Это может быть:
 - Репозиторий из пакета org.springframework.data.repository

- Резпозиторий из вашего проекта
- Будет ли репозиторий расширять JpaRepositoryExecutor или нет
- Исходный корень
- Пакет

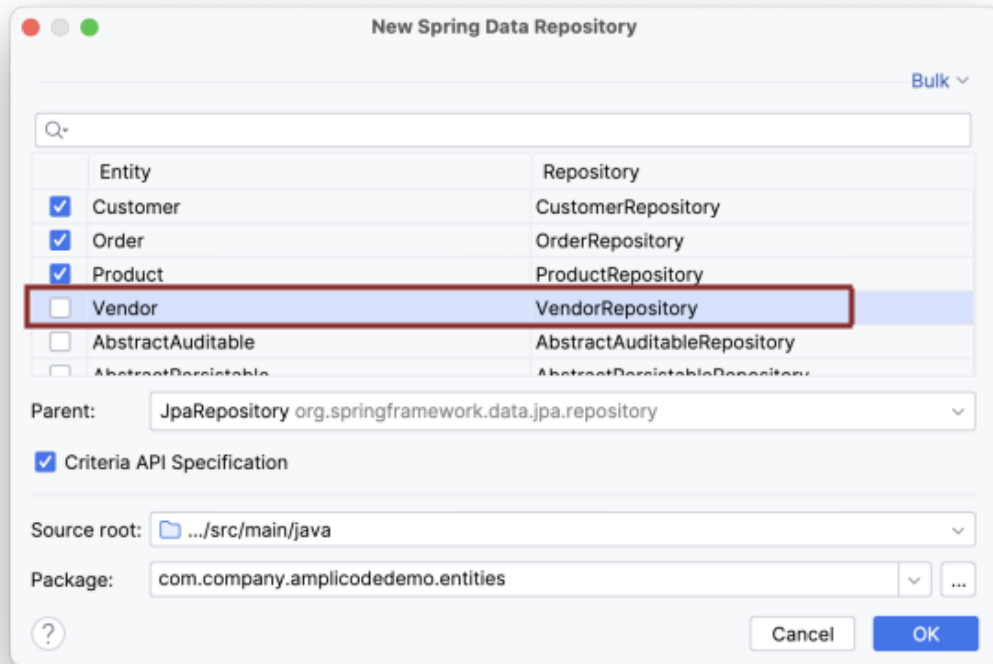


Когда у нас много сущностей, создание репозитория Spring Data по одному становится утомительной работой. С помощью Amplified вы можете ускорить этот процесс. Чтобы создать репозитории для сущностей JPA, вам нужно выполнить три шага:

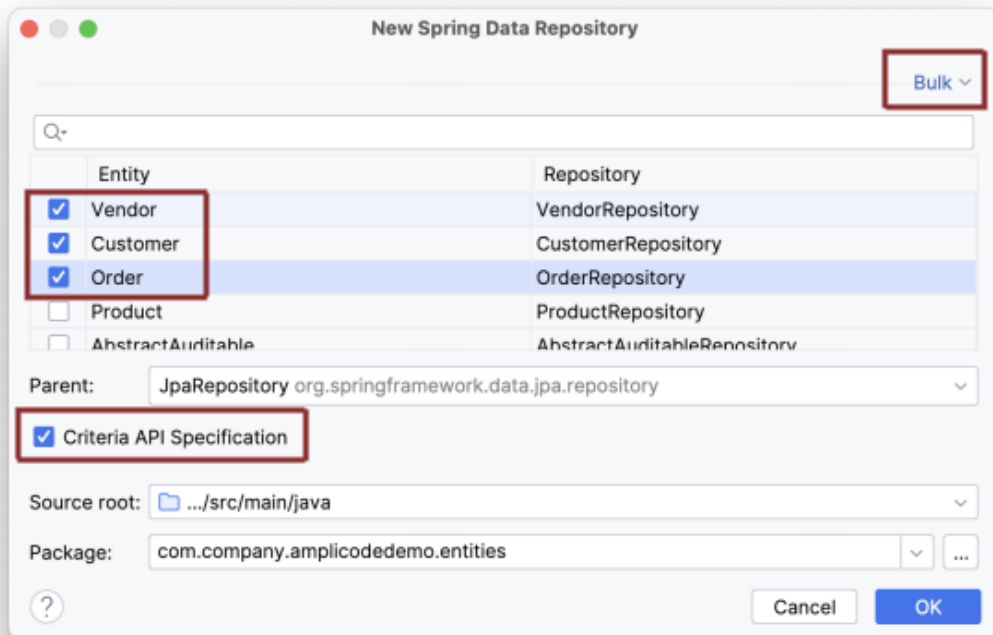
- Выберите сущности в дереве проекта
- Вызовите мастер создания репозитория Amplified,



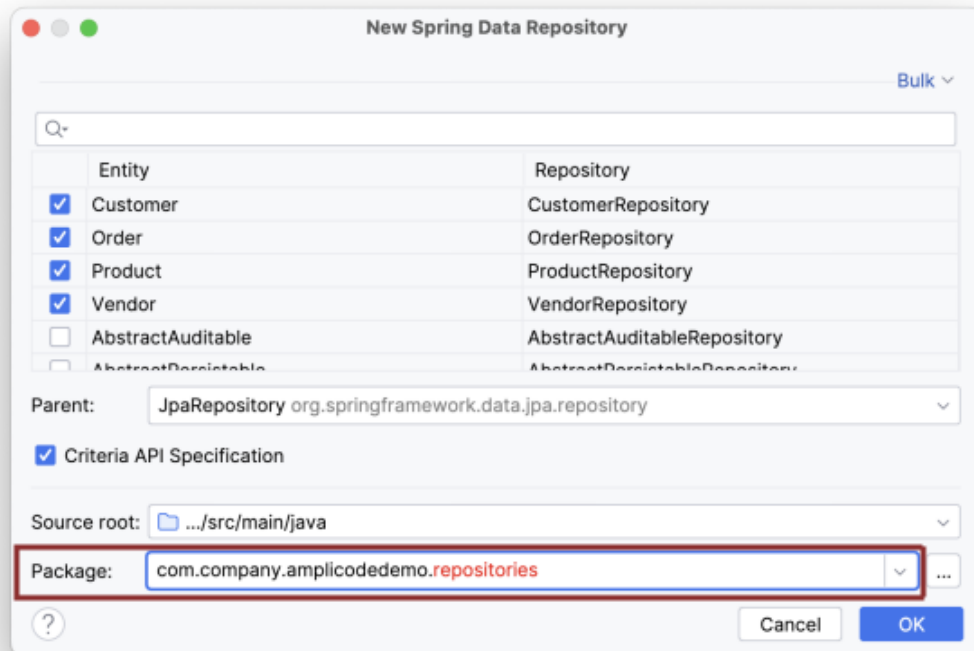
- Подтвердите свой выбор



Вы также можете переключиться с Single на Bulk непосредственно в окне New Spring Data Repository, а затем выбрать сущности.

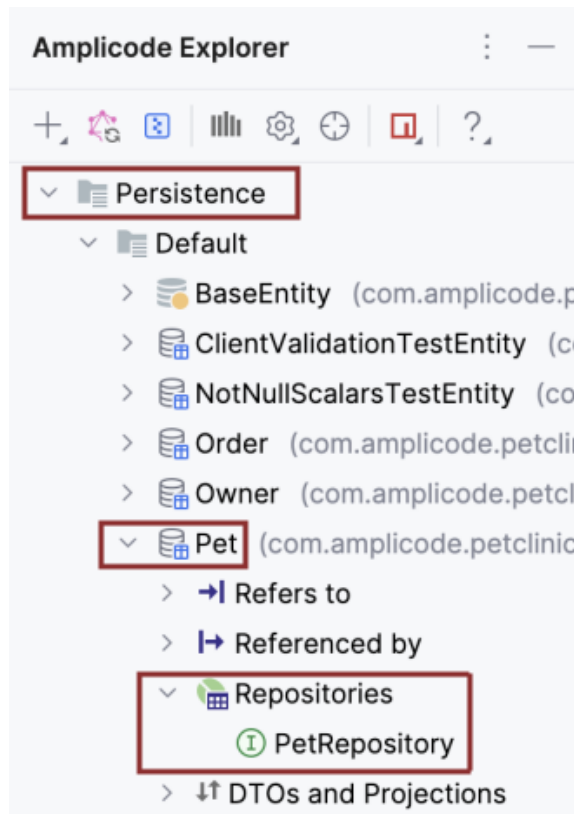


После создания репозитория вы можете перемещать их из одного пакета в другой в дереве проекта методом drag-n-drop. Либо создайте отдельный пакет только для репозитория и введите его имя в поле Package окна создания репозитория. Если каталог пакета не существует, он будет создан автоматически.



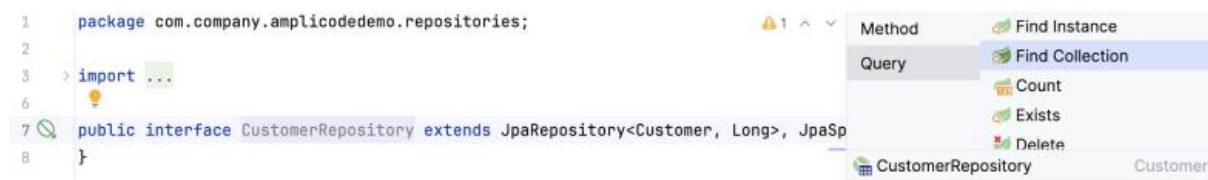
Репозитории на панели Amplicode Explorer

Для наиболее эффективной навигации по проекту Amplicode группирует все репозитории для каждой сущности/MongoDB документа. Неважно, находятся ли репозитории для сущности в разных или в одном пакете проекта. Все репозитории, связанные с сущностью, будут отображены в разделе Repositories. Отсюда можно быстро перейти к реализации репозитория или создать новый.



Создание запросов и методов

Spring Data предоставляет возможность определить запрос с помощью аннотации `@Query`. Вы можете использовать Палитру, чтобы писать их быстро и без опечаток.

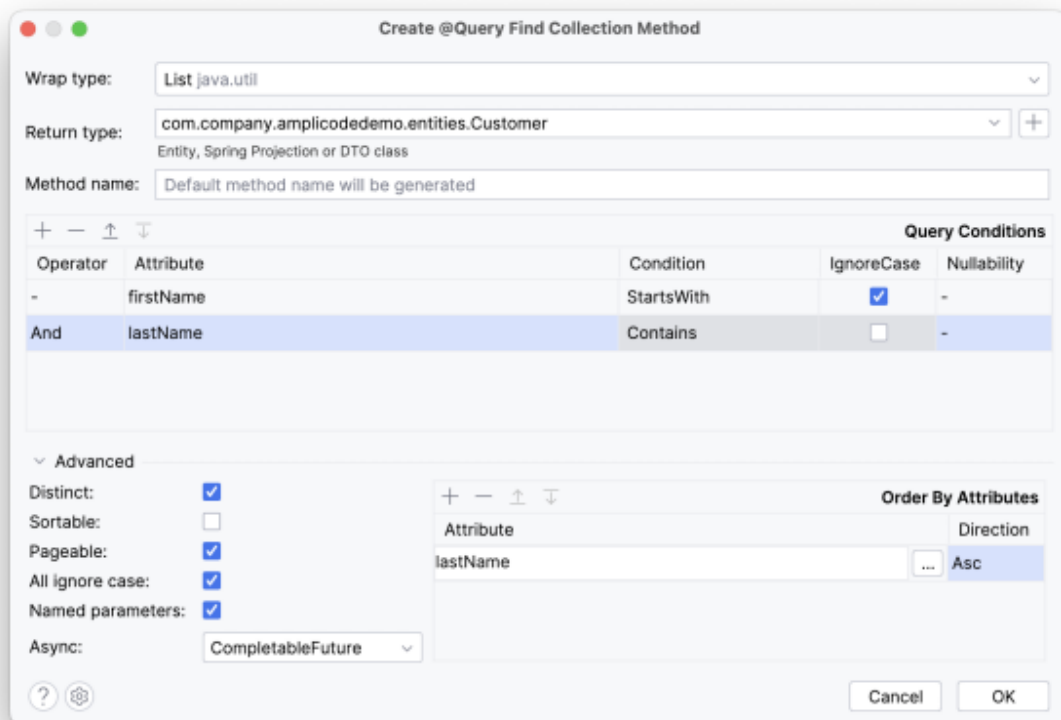


Выберите тип запроса и настройте его с помощью удобного пользовательского интерфейса.

Пример

Все, что мы рассмотрим в примере ниже, может быть сгенерировано как `@Query` или как производный метод запроса. Единственное отличие в том, что для производного метода запроса вы не можете указать его имя. Он будет сгенерирован автоматически в соответствии с Соглашением об именовании для запроса.

Давайте рассмотрим пример создания запроса Find Collection.



В верхней части окна можно определить тип обертки для коллекции и тип возвращаемого запроса. Более того, Amplicode позволяет вам генерировать новый интерфейс Projection или класс DTO непосредственно из этого окна (просто нажмите кнопку +).

Вы также можете указать имя метода. Тем не менее, если вы оставите его пустым, имя будет сгенерировано автоматически в соответствии со [стратегией именования для запроса](#).

В середине окна находится таблица для условий запроса.

В нижней части окна можно указать:

- будут ли параметры иметь имя или нет;
- будет ли использоваться параметр Pageable или нет;
- и т. д.

Наконец, вы можете указать поля, по которым будет упорядочен результат запроса.

Для указанной выше конфигурации будет сгенерирован следующий запрос:

```
public interface CustomerRepository extends JpaRepository<Customer, Long>,
JpaSpecificationExecutor<Customer> {
    @Query("""
        select distinct c from Customer c
        where upper(c.firstName) like upper(concat(:firstName, '%'))
        and upper(c.lastName) like upper(concat('%', :lastName, '%'))
    """)
```

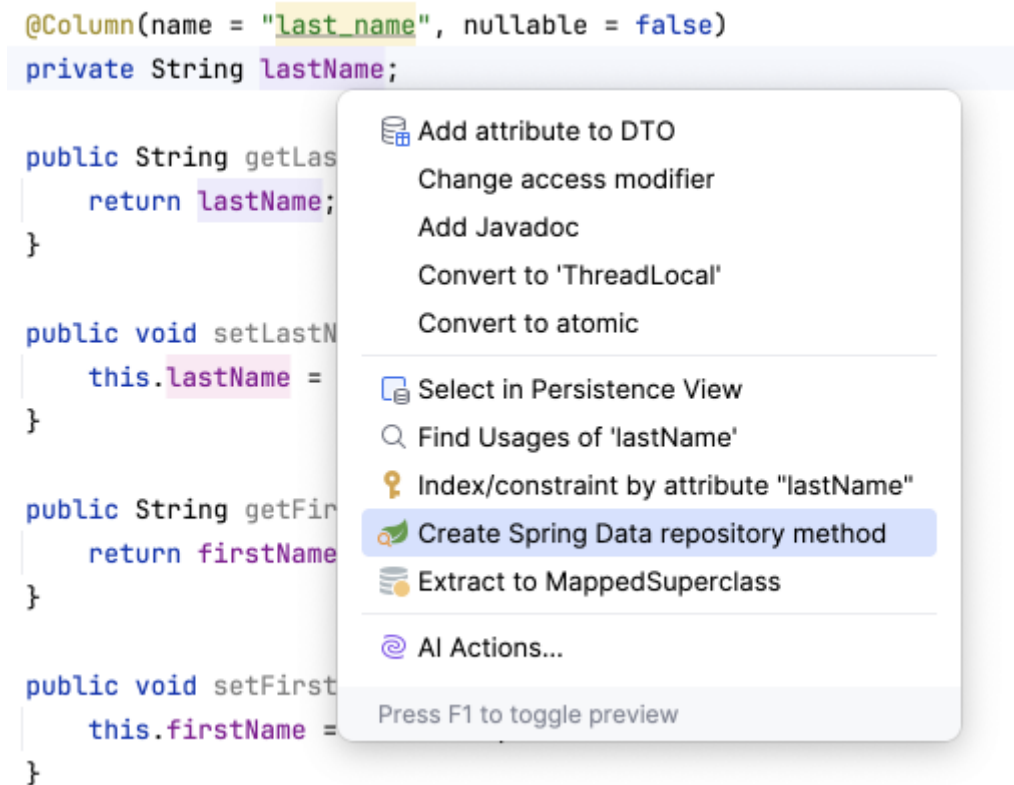
```

        order by c.lastName""")
    @Async
    CompletableFuture<List<Customer>>
findDistinctByFirstNameStartsWithIgnoreCaseAndLastNameContainsAllIgnoreCaseOrderByLastNameAsc(
    @Param("firstName") String firstName,
    @Param("lastName") String lastName,
    Pageable pageable);
}

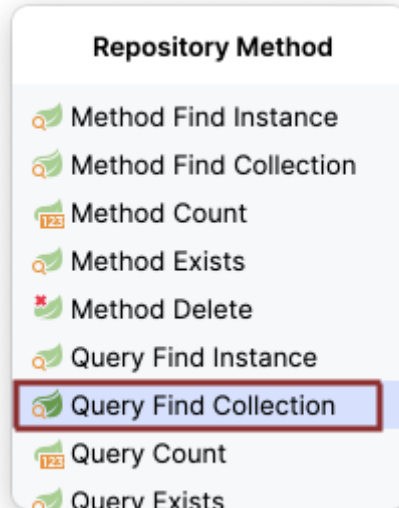
```

Entity Intention

Amplicode также предоставляет intention для атрибутов сущности, ведущих непосредственно к окну создания запроса/метода. Поместите курсор на нужный атрибут, нажмите Alt+Enter (или Opt+Enter на Mac) и выберите Create Spring Data repository.



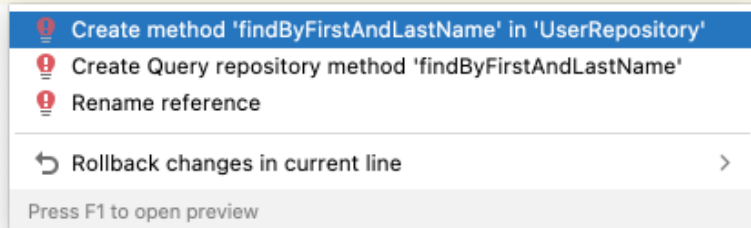
В открывшемся окне выберите необходимый тип запроса/метода.



Некорректные ссылки

Некоторые разработчики предпочитают сначала объявить вызов метода, который еще не существует, а затем реализовать его. Amplicode полностью совместим с этим стилем программирования. Просто напишите нужную сигнатуру и перейдите к мастеру создания запроса или метода с помощью специальных действий:

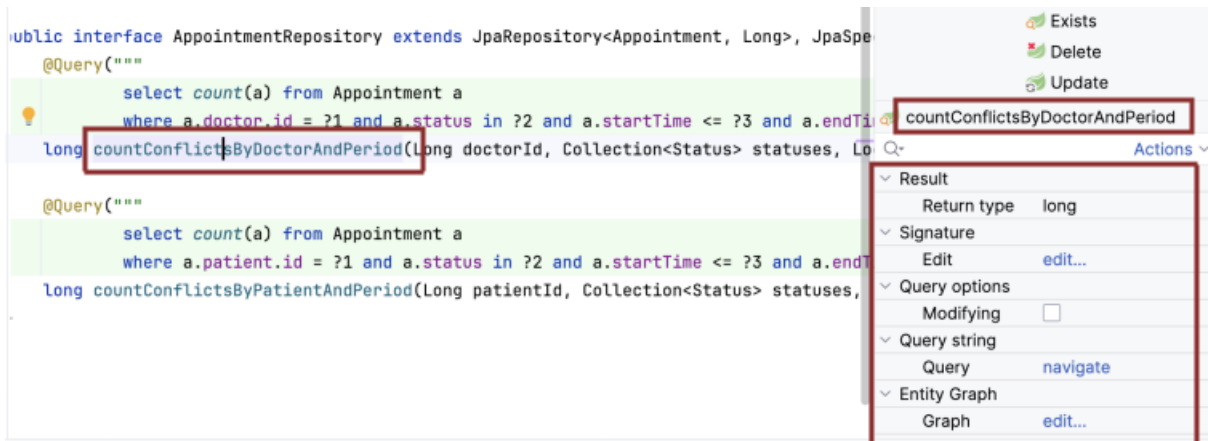
```
userRepository.findByFirstNameAndLastName(firstName, lastName);
```



Show Context Actions via $\backslash \leftrightarrow$ (Alt+ \leftrightarrow for Win/Linux)

Изменение существующих запросов и методов

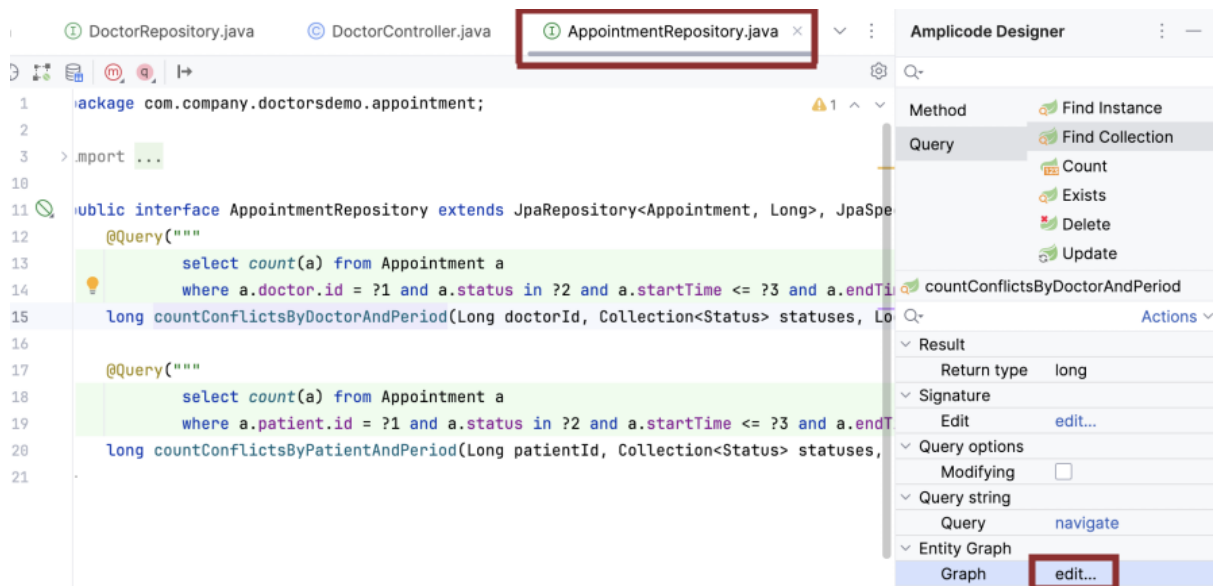
Чтобы настроить метод или запрос, поместите на него курсор и используйте Инспектор:



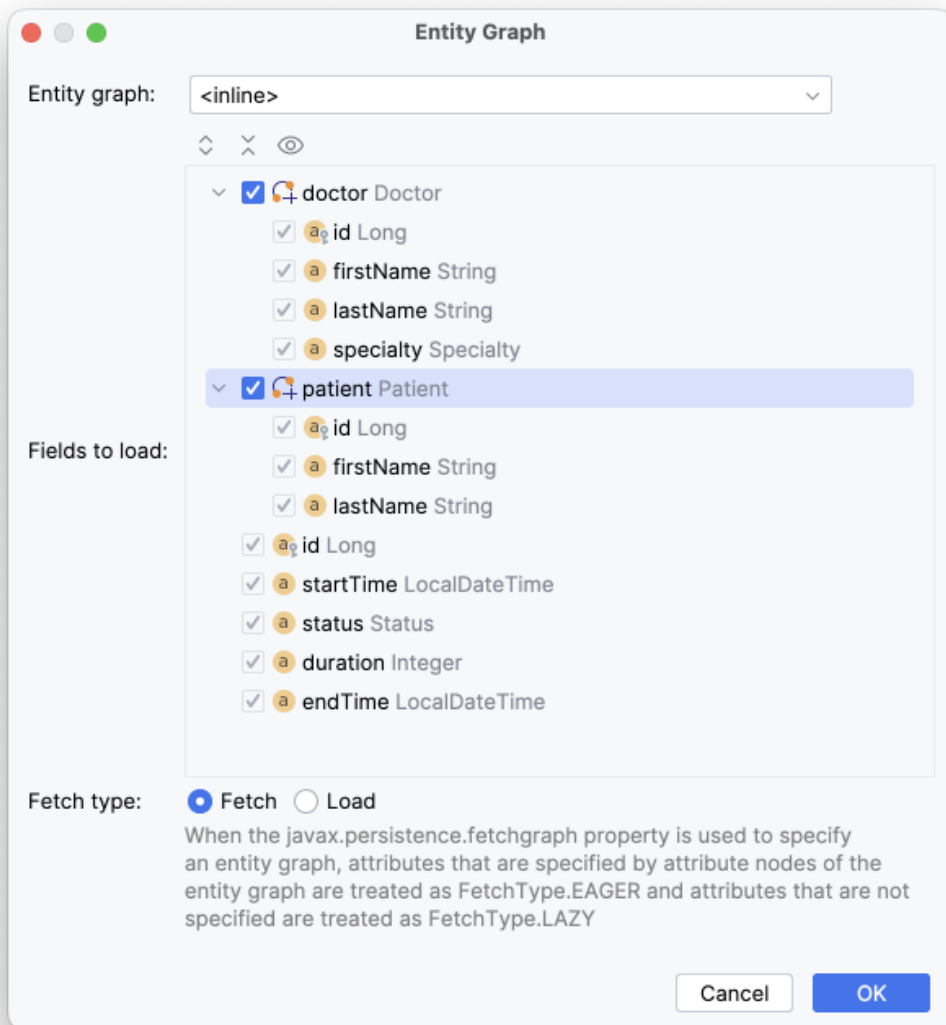
Поддержка EntityGraph

Функция EntityGraph всегда была одной из самых востребованных функций. Графы сущностей дают нам еще один уровень контроля над данными, которые необходимо извлечь. Amplicode поддерживает их, поэтому вы можете строить графики с помощью удобного мастера графического интерфейса.

Щелкните правой кнопкой мыши нужный файл репозитория в дереве проекта. В инспекторе найдите Graph и нажмите «Изменить».



В окне EntityGraph отметьте и снимите отметки с вложенных сущностей в соответствии с вашими требованиями.



Нажмите ОК. Пример полученного кода приведен ниже:

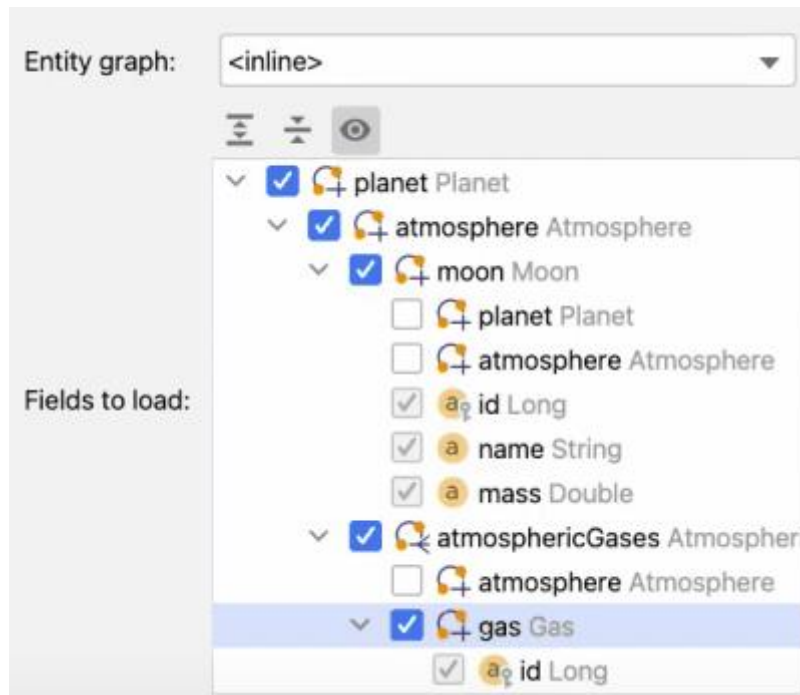
```
public interface AppointmentRepository extends JpaRepository<Appointment, Long>,
JpaSpecificationExecutor<Appointment> {
    @EntityGraph(attributePaths = {"doctor", "patient"})
    @Query("""
        select count(a) from Appointment a
        where a.doctor.id = ?1 and a.status in ?2 and a.startTime <= ?3 and a.endTime >= ?4""")
    long countConflictsByDoctorAndPeriod(Long doctorId, Collection<Status> statuses, LocalDateTime
maxStartTime, LocalDateTime minEndTime);

    @Query("""
        select count(a) from Appointment a
        where a.patient.id = ?1 and a.status in ?2 and a.startTime <= ?3 and a.endTime >= ?4""")
    long countConflictsByPatientAndPeriod(Long patientId, Collection<Status> statuses, LocalDateTime
maxStartTime, LocalDateTime minEndTime);
}
```

}

Вы можете увидеть добавленную аннотацию `@EntityGraph` в первом запросе.

Глубина вложенности может варьироваться, но функция `EntityGraph` по-прежнему будет поддерживать полный граф со всеми уровнями вложенности. Смотрите пример ниже:



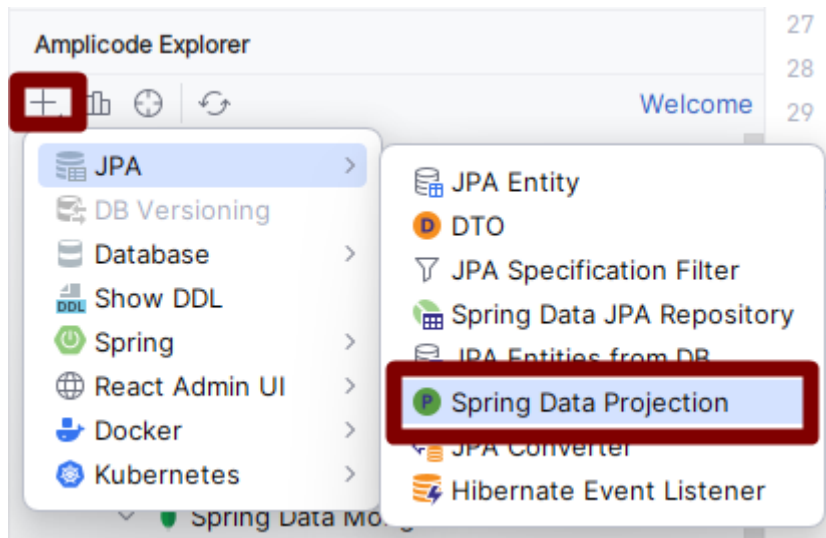
Дополнительная информация по теме:

- [Configuring Fetch- and LoadGraphs \(Spring documentation\)](#)
- [Dynamic fetching via Jakarta Persistence entity graph \(Hibernate user guide\)](#)

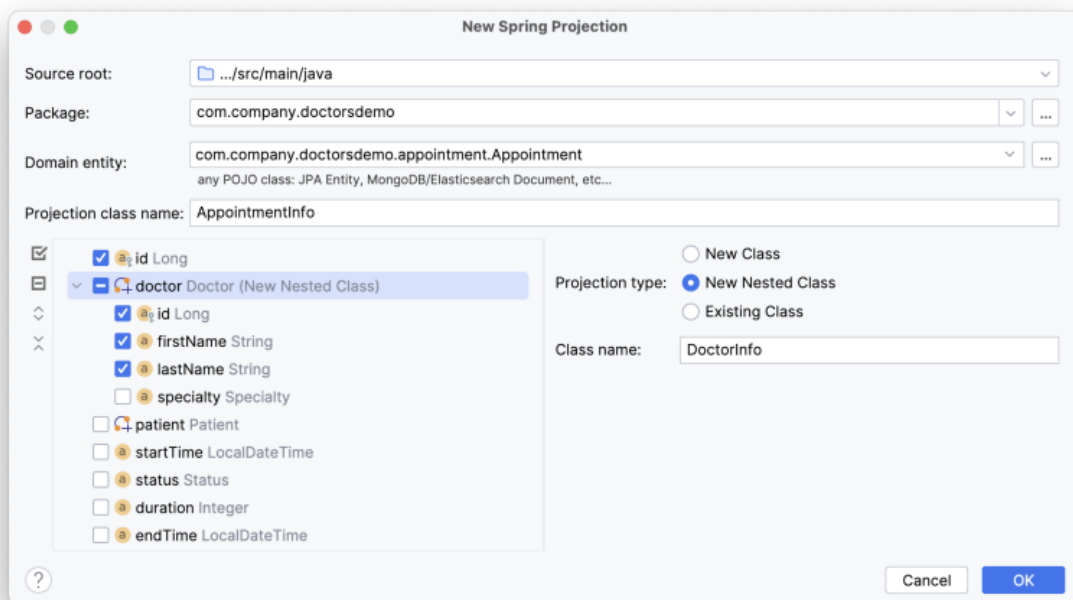
Проекции

Иногда вам нужно только подмножество столбцов из таблицы. В таких случаях Spring Data JPA Projections пригодятся, позволяя вам возвращать только требуемые поля из запросов. Больше информации о проекциях можно найти в [документации](#).

Чтобы создать проекцию, щелкните значок + на панели Amplicode Explorer и выберите JPA → Spring Data Projection.



Также Amplicode позволяет вам генерировать Проекции для указанных сущностей. Выберите связанную сущность, выберите Тип проекции и выберите необходимые поля.



Для указанной выше конфигурации будет создана следующая проекция:

```
public interface AppointmentInfo {
    Long getId();
    DoctorInfo getDoctor();

    /**
     * Projection for { @link com.company.doctorsdemo.doctor.Doctor }
     */
    interface DoctorInfo {
        Long getId();
    }
}
```

```
String getFirstName();

String getLastName();
}
}
```


Синхронизация проекции и сущности

Со временем сущности могут меняться, и вам нужно соответствующим образом менять проекции. Amplicode позволяет синхронизировать сущность с ее проекцией и наоборот. Подробнее об этой функции читайте в разделе Создание DTO.

Навигация между сущностью и проекциями

Amplicode сможет связать интерфейс Projection с сущностью:

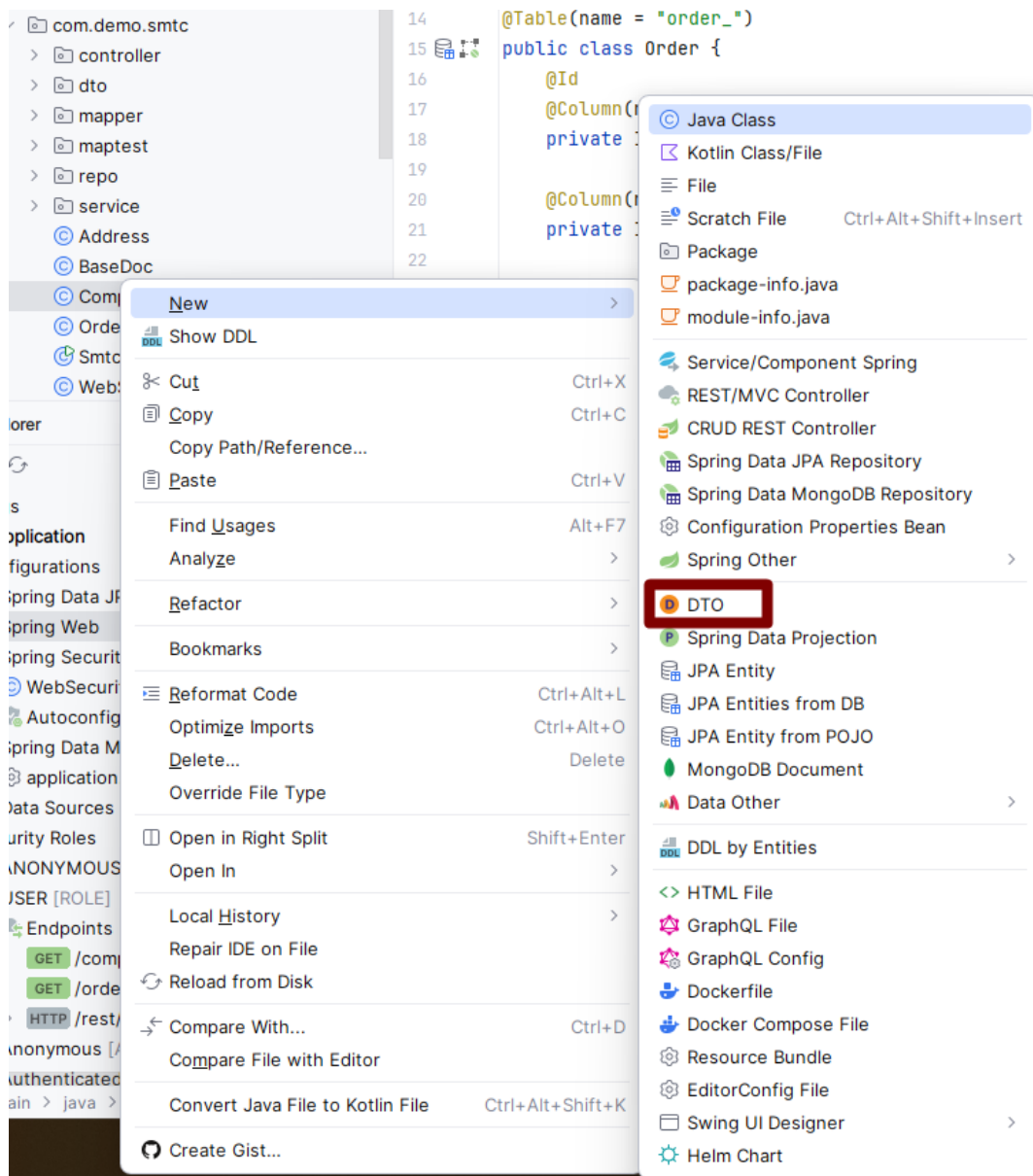
- Интерфейс Projection появится в разделе DTO и Projections на панели Amplicode Explorer и в навигаторе сущности (скриншот)
- Иконка навигации появится в проекции, чтобы облегчить навигацию по ее объекту.

```
14  interface DoctorInfo {
15     Long getId();
16
17     String getFirstName();
18
19     String getLastName();
20 }
21 }
```

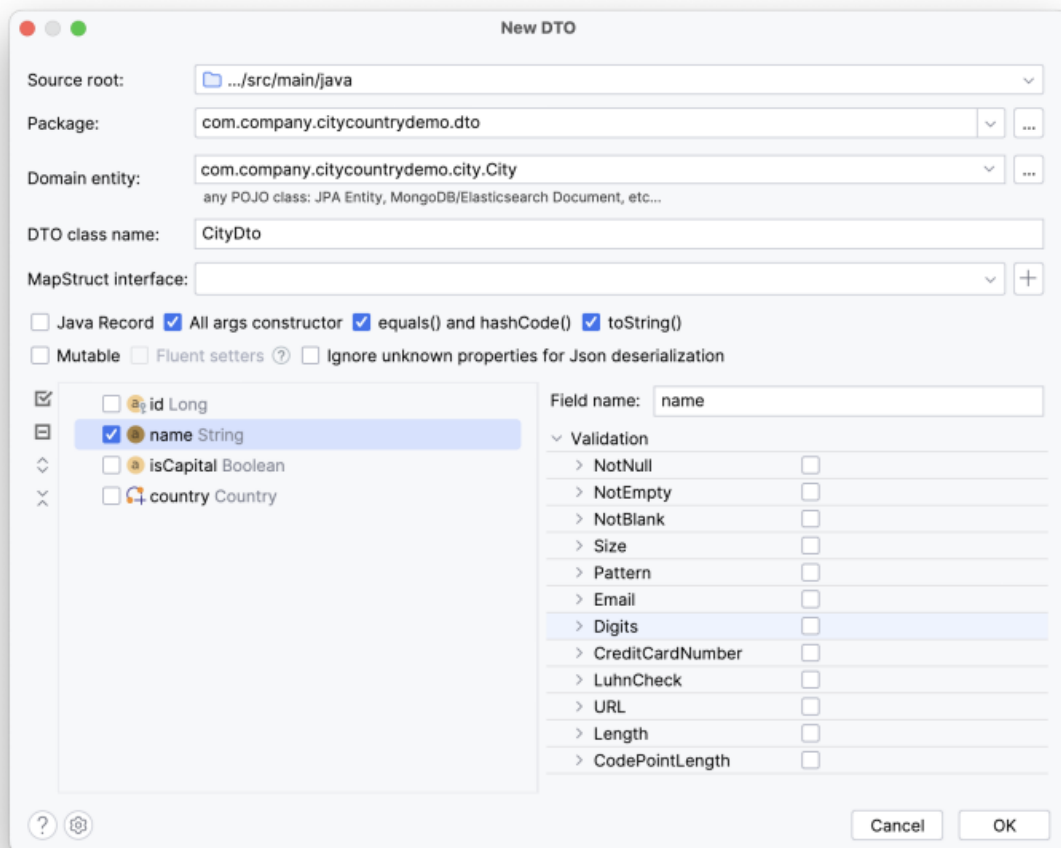
Работа с DTO

DTO (объект передачи данных) — это объект, который переносит данные между процессами. DTO для сущностей JPA обычно содержат подмножество атрибутов сущностей. По сути, DTO позволяют отделить уровень представления/бизнес-логики от уровня доступа к данным.

Amplicode предлагает генерацию DTO из сущностей JPA через визуальный конструктор. Щелкните правой кнопкой мыши класс сущности в дереве проекта, затем выберите New → DTO, как показано ниже:



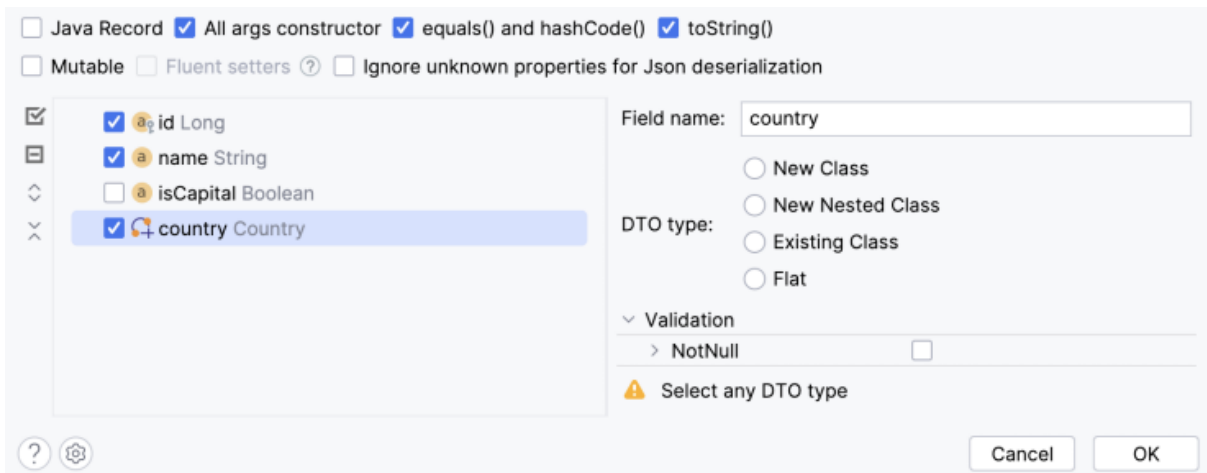
Появится следующее всплывающее окно:



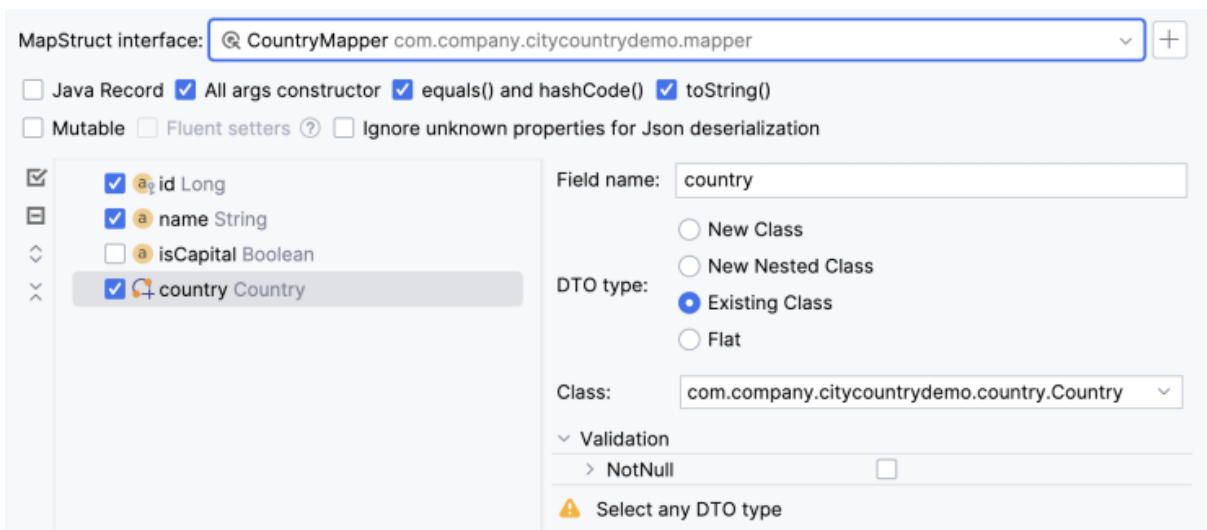
Нажатие на значок «Плюс» рядом с полем интерфейса MapStruct предоставит нам интерфейс для создания Mapper.

Amplicode предложит имя по умолчанию для Mapper в соответствии с соглашением об именовании, которое вы можете сохранить или изменить. Нажмите OK.

Следующий шаг — выбрать атрибуты, которые будут включены в DTO. При переключении между аргументами интерфейс в правой части окна изменяется, чтобы предоставить необходимые интерфейсы для добавления настроек. Например, для связанного атрибута он будет выглядеть следующим образом:



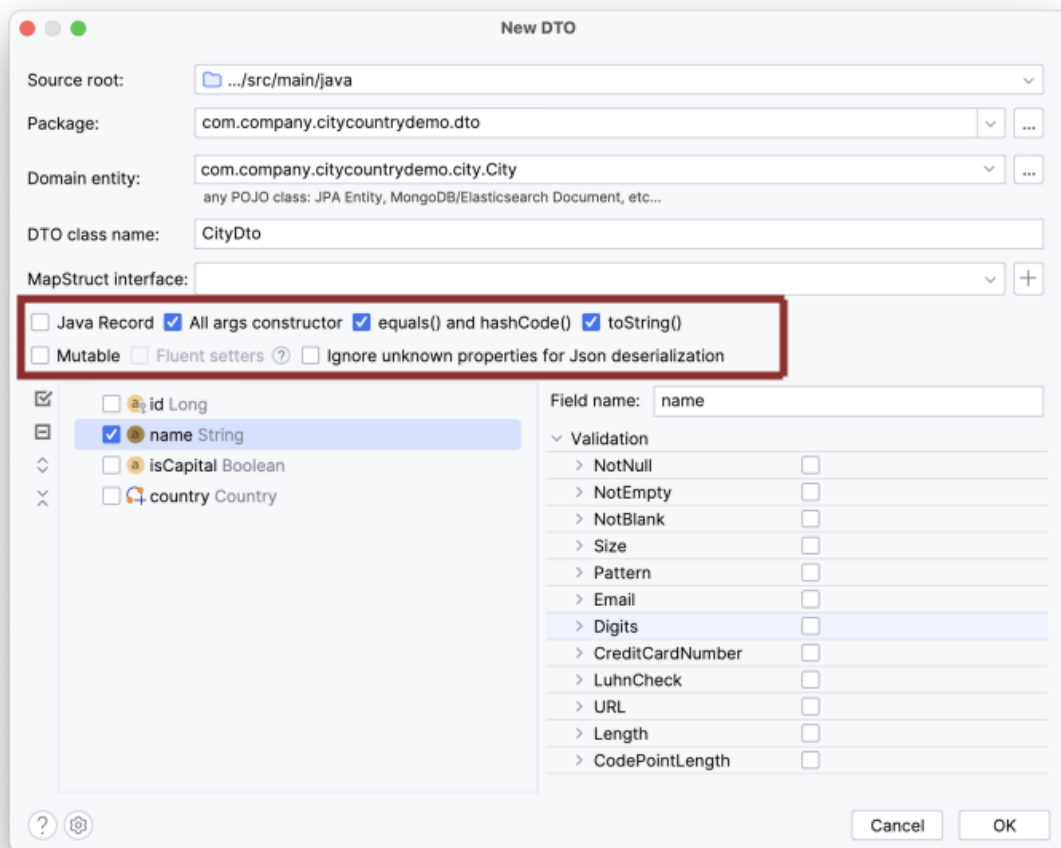
Для целей нашего проекта мы выбираем Existing Class для типа DTO и Country для класса.



Нажмите ОК. Будут сгенерированы два новых класса для DTO и для Mapper. Класс DTO будет содержать только те атрибуты, которые мы отметили в окне создания DTO.

Различные опции при создании DTO

При создании DTO Amplicode предоставляет на выбор следующие возможности:



- Java record — создает DTO как запись Java, предоставляя краткое и неизменяемое представление DTO с автоматическими реализациями equals(), hashCode() и toString()
- All args constructor — создает конструктор, который принимает аргументы для всех полей DTO
- equals() и hashCode() — создает методы equals() и hashCode() для DTO на основе его полей
- toString() — создает метод toString() для DTO, предоставляя строковое представление его полей
- Mutable — по умолчанию сгенерированные DTO являются неизменяемыми с окончательными полями и без сеттеров. Если вам нужны изменяемые DTO с закрытыми полями и сеттерами, вы можете отметить эту опцию
- Fluent setters — эта опция доступна, если выбрана опция Mutable. Она позволяет сгенерированным сеттерам возвращать this вместо void, что позволяет объединять методы в цепочку для нескольких вызовов сеттеров

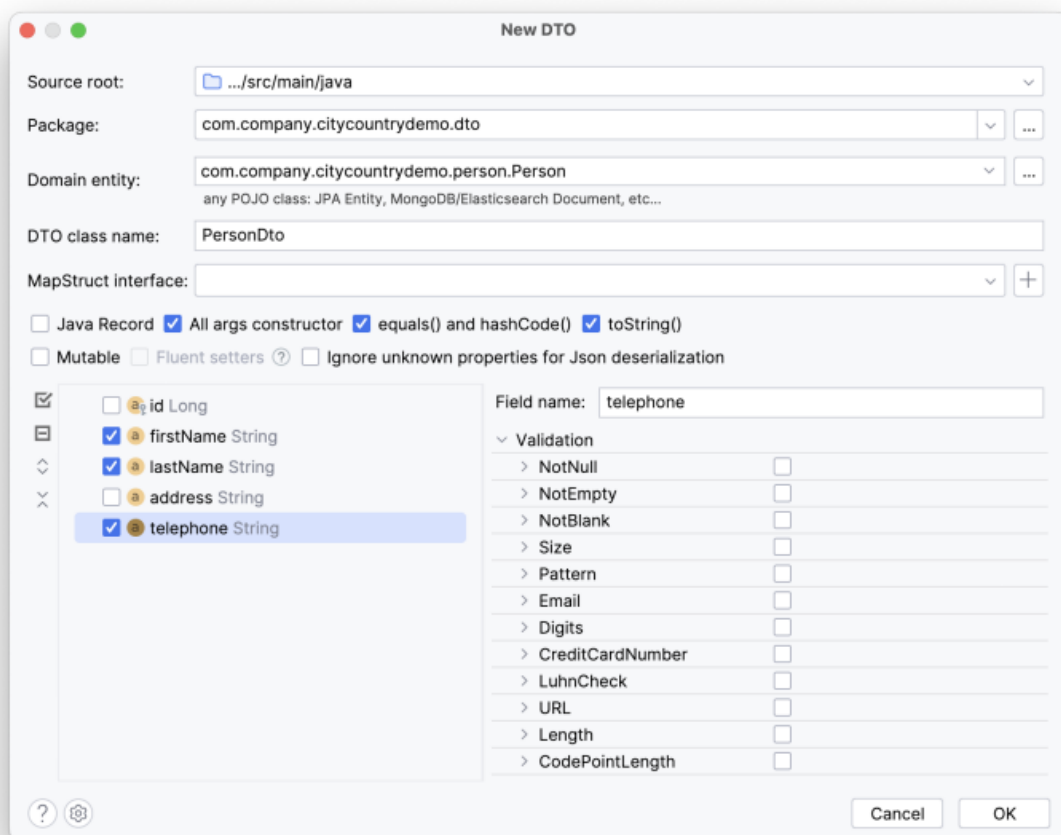
Java Record All args constructor equals() and hashCode() toString()
 Mutable Fluent setters ? Ignore unknown properties for Json deserialization

- Ignore unknown properties for json deserialization – применяет аннотацию `@JsonIgnoreProperties(ignoreUnknown = true)` к DTO, позволяя ему игнорировать любые неизвестные свойства во время десериализации JSON. Доступно только в том случае, если в проект включена зависимость Jackson Annotations.

Поддержка Lombok

Amplicode упрощает генерацию DTO, предоставляя поддержку Lombok наиболее оптимальным способом.

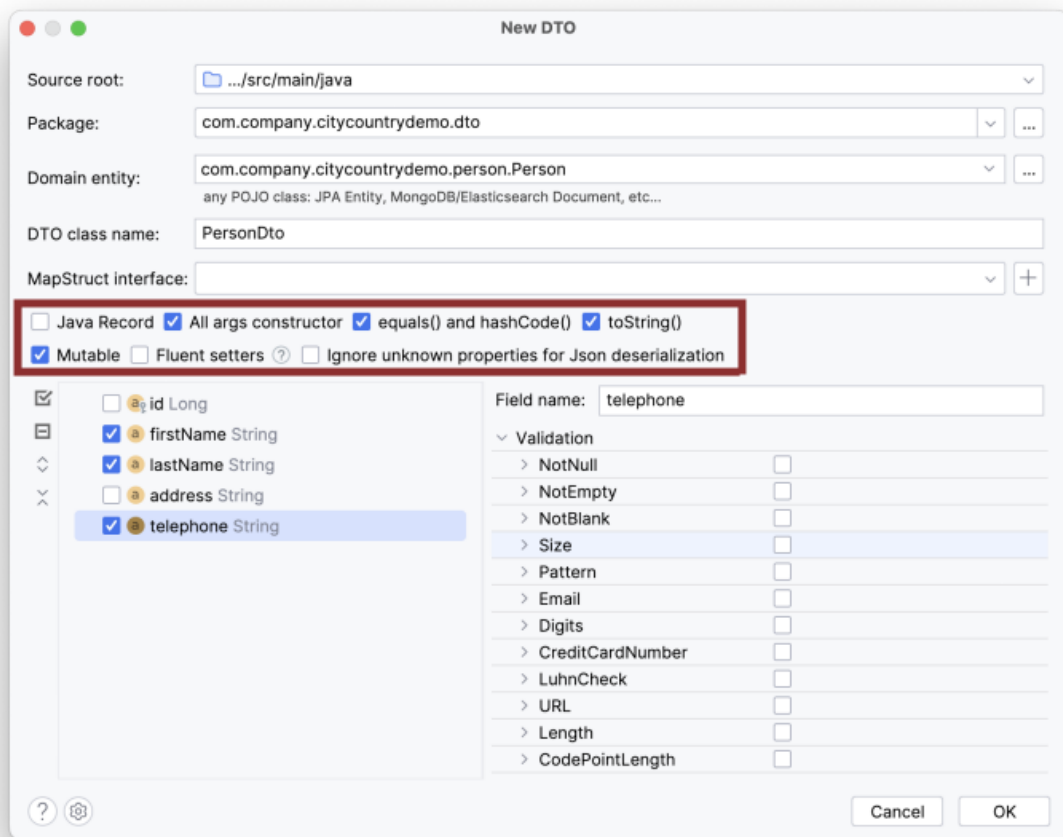
Например, когда вы отмечаете флажки All args constructor, equals() и hashCode() и toString() в мастере генератора DTO, Amplicode применяет `@Value` к сгенерированному DTO, отбрасывая избыточные модификаторы доступа, чтобы сохранить чистоту кода.



@Value

```
public class PersonDto implements Serializable {
    String firstName;
    String lastName;
    String telephone;
}
```

Если вам нужен изменяемый DTO с теми же параметрами, что и в примере выше, Amplicode добавит вместо этого аннотации `@Data`, `@AllArgsConstructor` и `@NoArgsConstructor`.



@Data

@AllArgsConstructor

@NoArgsConstructor

```
public class PersonDto1 implements Serializable {
    private String firstName;
    private String lastName;
    private String telephone;
}
```

Чтобы использовать эту функцию, обязательно добавьте [зависимости Lombok](#) в свой проект и включите ее в настройках декларации DTO (см. соответствующий раздел документации).

Вложенные DTO для атрибутов ассоциации

Доменные объекты могут ссылаться на другие доменные объекты через ассоциации, и Amplicode позволяет генерировать для них DTO из того же окна. Просто выберите поле доменной сущности в дереве и затем выберите тип DTO.

Всего для выбора доступны следующие типы DTO:

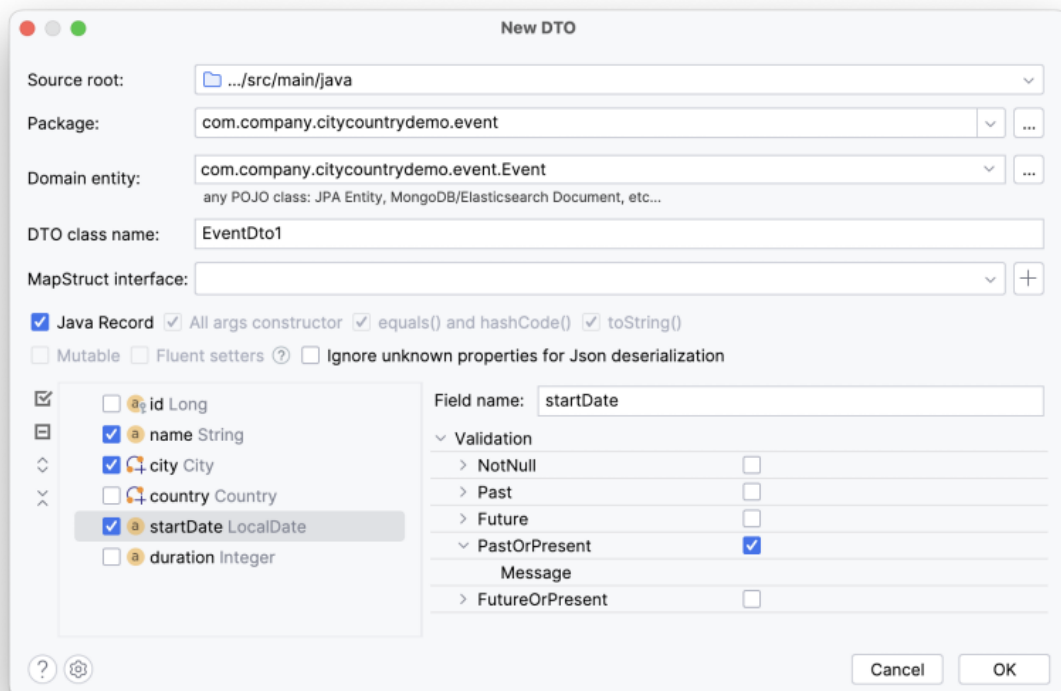
- New Class - будет создан новый класс в отдельном файле;
- New Nested Class - будет создан новый общедоступный статический вложенный класс;
- Existing Class - вы можете выбрать существующий класс DTO, который уже существует в проекте;

- Flat - все поля вложенного класса будут преобразованы в поля единственного сгенерированного DTO. Имена сгенерированных полей будут формироваться путем объединения имени внутреннего класса с именами его полей.

Поддержка Java Records

Если вы используете в своем проекте SDK версии 16 и выше, то Amplicode предоставит дополнительный флажок Java Record в мастере New DTO. Чтобы проверить версию SDK проекта, выберите File → Project Structure.

Если вы хотите создать иммутабельные(неизменяемые) DTO, то Java Record — отличный вариант.



Пример

Для этого примера мы создаем отдельный пакет с именем dtos и устанавливаем флажок Java Record в окне New DTO. Мы также выбираем опцию New Class для ассоциированных элементов. Для атрибута startDate мы добавляем проверку (PastOrPresent) для генерации аннотации.

```
public record EventDto1(@PastOrPresent LocalDate startDate, CityDto city, String name) imple
```

Поскольку сущность и DTO будут находиться в разных пакетах, необходим следующий комментарий, который поможет им найти друг друга:

```
/**
 * DTO for { @link com.company.citycountrydemo.event.Event }
 */
```

Amplicode сгенерирует его автоматически. Появится значок, который поможет нам перемещаться между DTO и сущностью. Если исходная сущность имеет аннотации валидации, они будут скопированы в DTO.

Создание JPA сущностей из DTO

Amplicode предоставляет функцию "Create Entity from POJO", которая помогает создать JPA сущность из любого Java/Kotlin класса. Эта возможность может быть полезной, если вы разрабатываете свое приложение, следуя подходу API-first: сначала определяете DTO для API, а затем реализуете модель данных.

Диалог создания JPA сущностей из DTO можно условно разделить на несколько частей. В верхней части можно настроить базовые параметры:

- Source root;
- Имя пакета;
- POJO класс;
- Имя, которое будет установлено JPA сущности;
- Родительская JPA сущность;
- Помимо этого, существует возможность выбрать существующий или создать новый MapStruct интерфейс для преобразования JPA сущности в DTO и обратно. Данная функциональность доступна только при наличии MapStruct зависимости.
- Наконец, Amplicode предлагает создать новый атрибут, который будет выступать в роли уникального идентификатора для JPA сущности. Альтернативно, можно выбрать один или несколько атрибутов для id из уже существующих.

Нижняя часть визуального дизайнера представляет собой master-detail панель выбора атрибутов, которые будут включены в JPA сущность. При переключении между аргументами интерфейс справа от списка атрибутов изменяется, предоставляя необходимые параметры. Например, в случае наличия библиотеки валидации, отобразится список доступных валидаций. А в случае выбора ассоциативного атрибута, появится возможность настройки ассоциативных связей.

Настройка ассоциативных атрибутов

Amplicode предоставляет широкие возможности настройки ассоциативных атрибутов. Есть возможность выбрать тип атрибута:

- Transient – в сущности будет сгенерировано поле с тем же типом, что и в POJO классе, аннотированное аннотацией @Transient;
- Create new Entity Class – в визуальном дизайнера появятся параметры необходимые для [новой JPA сущности](создание JPA сущности);
- Select Existing Entity Class – в визуальном дизайнера появится возможность выбора существующей сущности из выпадающего списка.

Помимо этого, при выборе "Create new Entity Class" и "Select Existing Entity Class", необходимо будет настроить [параметры ассоциативной связи](создание ассоциации), которая будет сгенерирована в новой JPA сущности.

Поддержка MapStruct

Описанная ниже функциональность доступна только при наличии подключенной к проекту библиотеки MapStruct.

Amplicode позволяет сгенерировать MapStruct интерфейс через:

- Amplicode Explorer -> Persistence -> Entity -> Правая кнопка мыши -> MapStruct Interface;
- Project -> Правая кнопка мыши / (Cmd+N/Alt+Insert) -> New -> Other -> MapStruct Interface;
- Диалог создания DTO -> MapStruct interface;
- Диалог создания доменных объектов из DTO -> MapStruct interface;
- Диалог делегирования методов -> Method Parameters / Return Options.

В диалоге создания MapStruct интерфейса можно настроить следующие параметры:

- Доменный объект;
- DTO класс;
- Имя для интерфейса который будет сгенерирован;
- Имя пакета.

По умолчанию, в MapStruct интерфейсе будут сгенерированы сигнатуры методов для преобразования DTO в доменный объект и обратно, а также для частичного обновления.

Генерация методов

Amplicode предоставляет возможности создания отдельных методов для существующего MapStruct интерфейса. Диалог создания MapStruct методов можно вызвать, открыв "Generate Menu" (Cmd+N/Alt+Insert) и выбрав пункт "Mapper Methods...".

Использование дженерик MapStruct интерфейса

MapStruct позволяет объявлять дженерик интерфейсы. Например:

```
public interface EntityMapper<D, E> {  
    E toEntity(D dto);  
    D toDto(E entity);  
    List<E> toEntity(List<D> dtoList);  
    List<D> toDto(List<E> entityList);  
}
```

Такой MapStruct интерфейса удобно использовать в качестве родительского для всех остальных интерфейсов, чтобы сохранить их лаконичность. В случае обнаружения такого дженерик интерфейса в проекте Amplicode расширит окно создания MapStruct интерфейса новой опцией – "Parent interface".

В результате Amplicode сгенерирует MapStruct интерфейс с ключевым словом extends и указанием выбранных типов для преобразования. Например:

```
@Mapper(componentModel = "spring")  
public interface UserMapper extends EntityMapper<UserDTO, User> {
```

```
}
```

Если же для преобразования будут требоваться дополнительные методы, Amplicode не забудет их сгенерировать. Например, метод для связывания дочерних объектов:

```
@Mapper(unmappedTargetPolicy = ReportingPolicy.IGNORE, componentModel = "spring")
public interface ProjectMapper extends EntityMapper<ProjectDTO, Project> {
    @AfterMapping
    default void linkTasks(@MappingTarget Project project) {
        project.getTasks().forEach(task -> task.setProject(project));
    }
}
```

Синхронизация DTO и доменного объекта

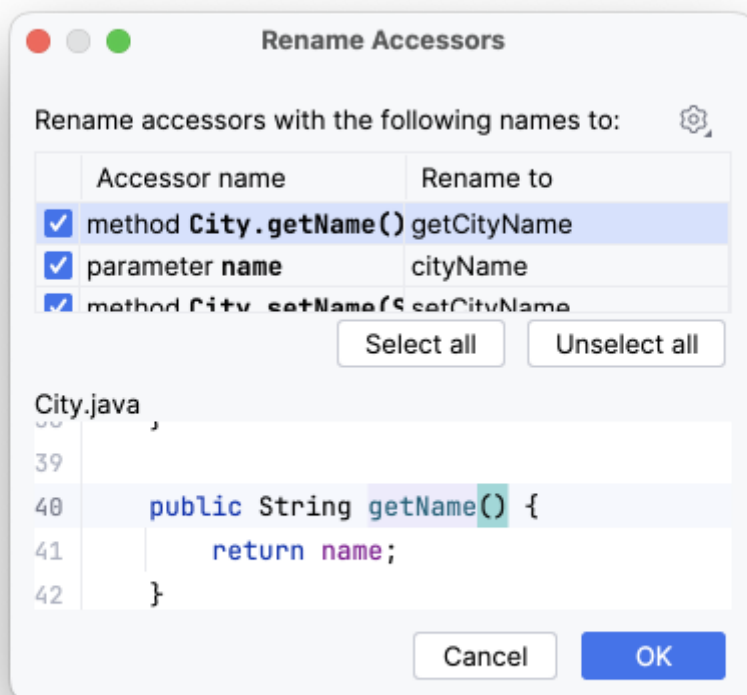
Изменение существующих атрибутов

DTO обычно используются на уровне контроллера API для определения только полей, требуемых клиентом. Вот почему DTO почти копируют структуру своих сущностей. С Amplicode вы можете использовать библиотеку MapStruct для сопоставления сущностей с DTO и наоборот. Библиотека автоматически сопоставляет одноименные свойства. Однако изменение имени свойства в сущности часто приводит к повреждению логики сопоставления. Amplicode помогает разработчикам рефакторить свойства сущности вместе с их связанными полями в DTO:

Например, предположим, что мы используем опцию Refactor → Rename, чтобы переименовать атрибут name класса City в cityName.

```
@Column(name = "name", nullable = false)
private String cityName // ≡ ;
```

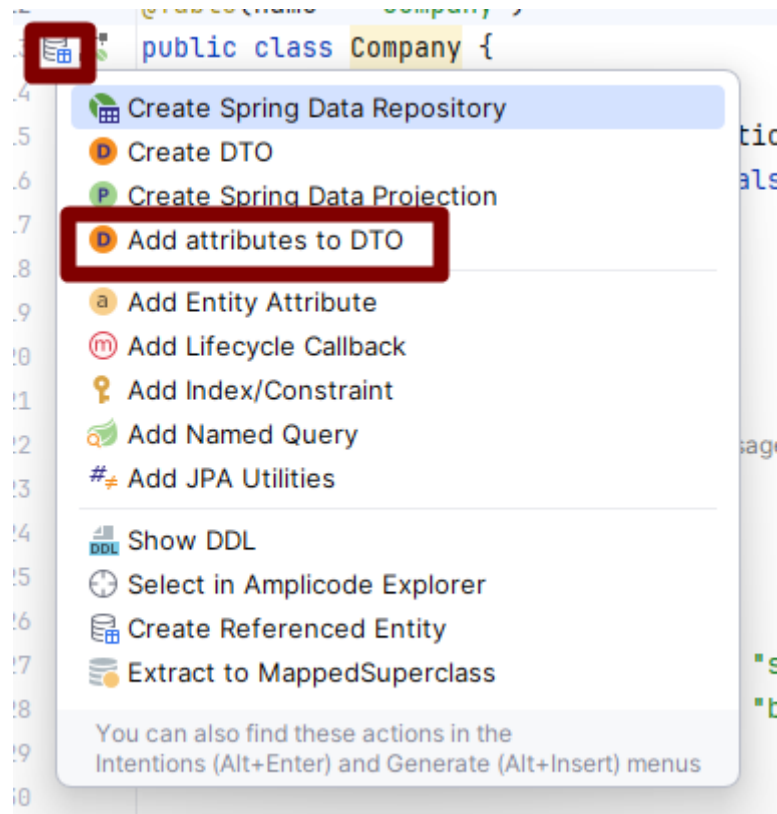
Появится следующее окно:



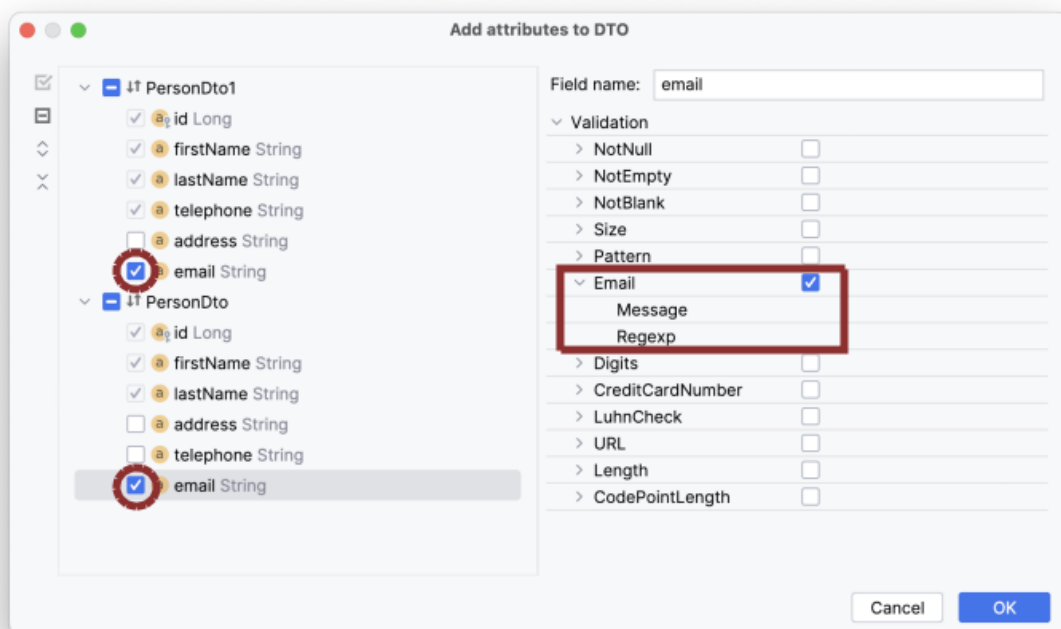
Окно предложит все необходимые изменения в названиях методов, компонентов и параметров, включая те, которые находятся в соответствующем классе DTO. Нам просто нужно решить, какие элементы в этом списке следует переименовать, снять галочки с остальных и нажать ОК.

Добавление атрибутов

Если вы добавили новый атрибут к сущности, соответствующие DTO также могут потребовать обновления с этим новым полем. Amplicode позволяет вам добавлять новое поле во все требуемые DTO одновременно. Для этой цели используйте значок Show Entity actions.



Выберите в меню пункт Add Attributes to DTO, затем во всплывающем окне отметьте новый атрибут во всех DTO, где он требуется.

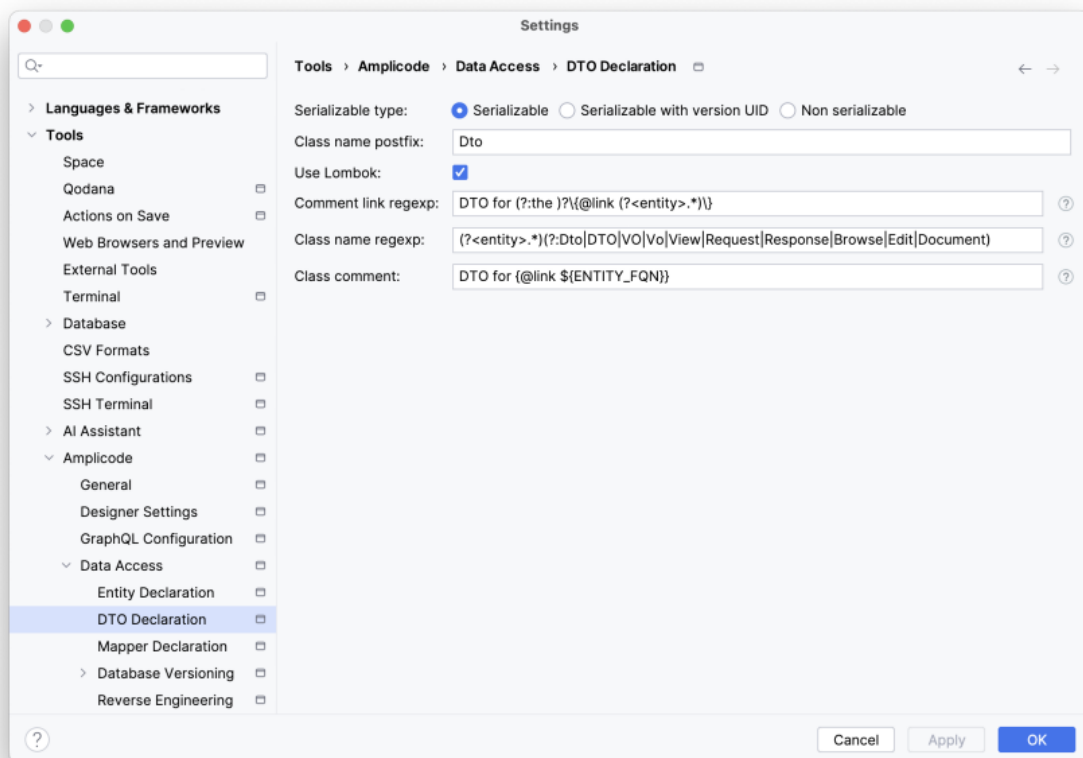


Более того, если вы предпочитаете вводить код вручную, а не использовать мастера, Amplicode может помочь вам и в этом. Просто начните вводить имя поля, которого нет в вашем DTO, и оно будет правильно добавлено в класс.


```
public class PersonDto1 implements Serializable {
    private final Long id;
    private final String firstName;
    private final String lastName;
    private final String telephone;
    add
    address from Person String telepho
    Press ^Space to see non-imported classes Next Tip
```

Работает также с ассоциациями. Эта функция работает с любой сущностью домена (любыми классами Java/Kotlin), а не только с сущностями JPA.

Настройки создания DTO



Каждый проект может следовать собственным соглашениям по написанию кода. В Tools → Amplicode → Data Access → DTO Declaration вы можете настроить:

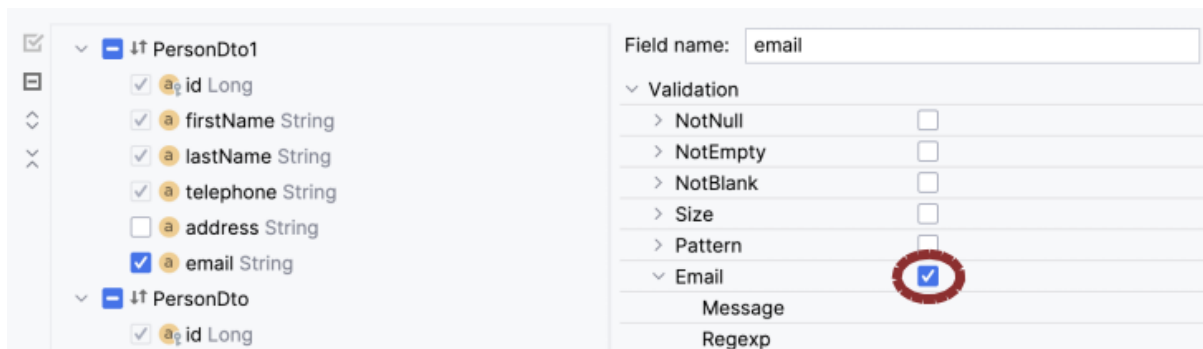
- Сериализуемый тип;
- Постфикс имени класса, который будет использоваться при генерации DTO;
- Использовать ли аннотации из библиотеки Lombok при генерации DTO;
- Регулярное выражение для комментариев, который будут использованы для связывания доменного объекта с его DTO. Для указания паттерна доменного объекта (Fully Qualified Name, FQN) в комментарии, используйте шаблон (?<entity>.). Например, если в данном поле указано значение DTO for (?the)?(@link (?<entity>.*))

)?{\@link (?<entity>.*)\}, будет распознан следующий комментарий: // DTO for {\@link io.jpabuddy.demo.entities.Project} entity. Описанная логика не работает, если поле пустое.

- Регулярное выражение для имени класса. Эта опция полезна, если вы следуете конвенции именования для DTO. Данная настройка позволяет Amplicode ассоциировать DTO с соответствующим доменным объектом только по имени DTO. Вы можете указать паттерн для простого имени класса доменного объекта, используя шаблон (?). Например, паттерн (?).Dto означает, что класс MyEntityDto будет считаться DTO для MyEntity. Описанная логика не работает, если поле пустое.
- Комментарий к классу. Задаёт комментарий, который будет генерироваться над DTO при его создании.

Правила валидации

Amplicode предлагает бесшовную настройку been validation ограничений для полей DTO в своем специальном мастере генерации DTO. Помимо определения проверок с нуля, вы можете автоматически переносить проверки из соответствующих сущностей и управлять ими в том же мастере.

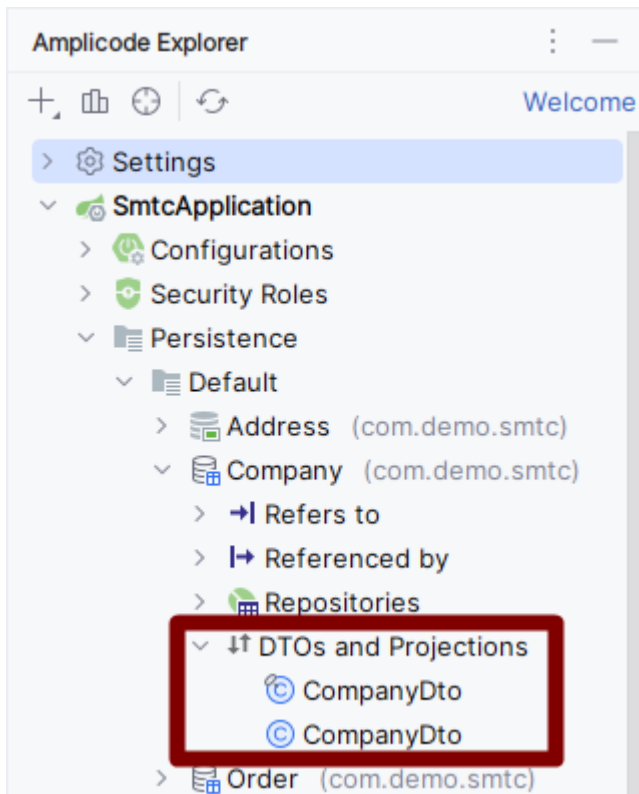


Благодаря возможности включения/отключения каждого ограничения и настраиваемым сообщениям проверки эта функция позволяет вам полностью управлять ограничениями проверки компонентов для ваших полей DTO.

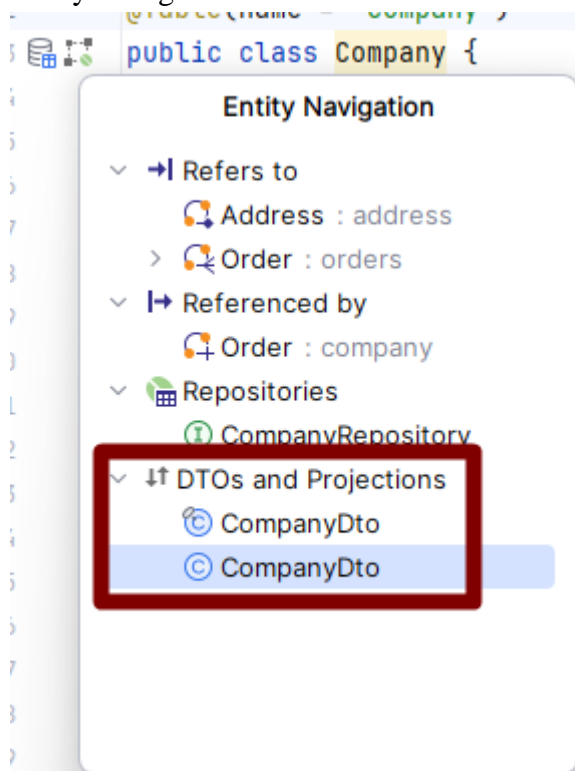
Навигация между сущностью и DTO

После того, как Amplicode свяжет класс DTO с соответствующей ему сущностью:

- Класс DTO появится в разделе Dto & Projections в дереве Amplicode Explorer и в окне Entity Navigation.
Amplicode Explorer:



Entity Navigation:



- В классе DTO появится значок, обеспечивающий удобный способ навигации к связанной с ним сущности.

Соединение с базой данных

Панель Amplicode Explorer отвечает за все, что связано с конфигурациями БД. Чтобы создать новое подключение к БД, нажмите кнопку «Плюс» и выберите Database -> DB Connection. Чтобы использовать функции Reverse Engineering и Database Versioning, первое, что вам нужно сделать, это создать подключение к базе данных.

На данный момент Amplicode поддерживает следующие базы данных:

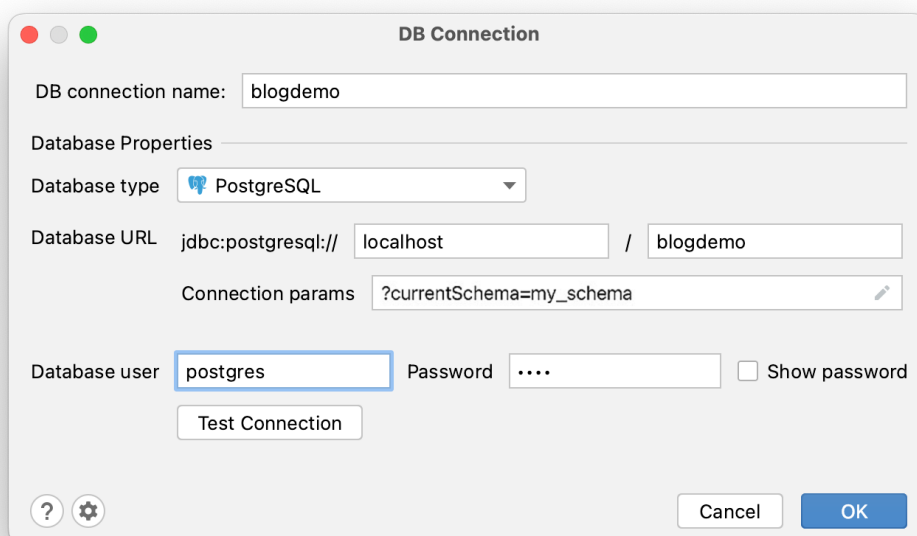
- PostgreSQL
- MSSQL
- MySQL
- MariaDB
- Oracle
- HSQLDB
- H2
- IBM Db2

Подключение к нестандартной схеме базы данных

Некоторые СУРБД, поддерживаемые Amplicode, предоставляют возможность создания нестандартных схем, но не все из них хорошо работают с JDBC. Вот почему вы можете столкнуться с некоторыми известными проблемами при генерации диффов или во время reverse engineering. На данный момент эти проблемы можно решить только с помощью определенных обходных путей. Ниже приведены примеры подключения к нестандартным схемам для всех баз данных, поддерживаемых Amplicode.

PostgreSQL

Схема PostgreSQL по умолчанию — public. Для других схем необходимо указать желаемое имя схемы в поле Connection params через параметр currentSchema:



Microsoft SQL Server

Схема Microsoft SQL Server по умолчанию — dbo. Чтобы подключиться к схеме, отличной от схемы по умолчанию в Microsoft SQL Server, необходимо выполнить следующие шаги:

1. Создайте ЛОГИН:
`create login JohnDoe with password='saPassword1'`
2. Создайте пользователя со схемой по умолчанию, из которой вы хотите создать сущность:
`create user JohnDoe for login JohnDoe with default_schema = my_schema`
3. Дайте пользователю права владельца:
`exec sp_addrolemember 'db_owner', 'JohnDoe'`
4. Создайте новое подключение с учетными данными только что созданного пользователя и добавьте имя схемы в поле URL базы данных

Для JDBC настройка подключения будет выглядеть следующим образом:

DB Connection

DB connection name:

Database Properties

Database type:

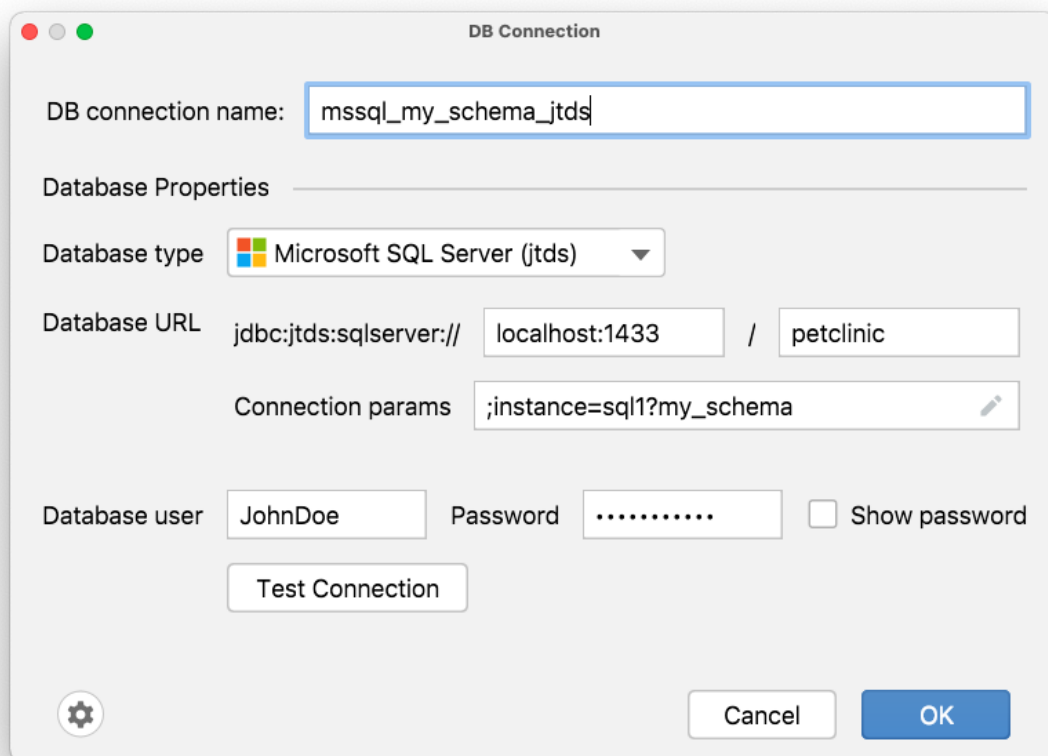
Database URL:

Connection params:

Integrated Security

Database user: Password: Show password

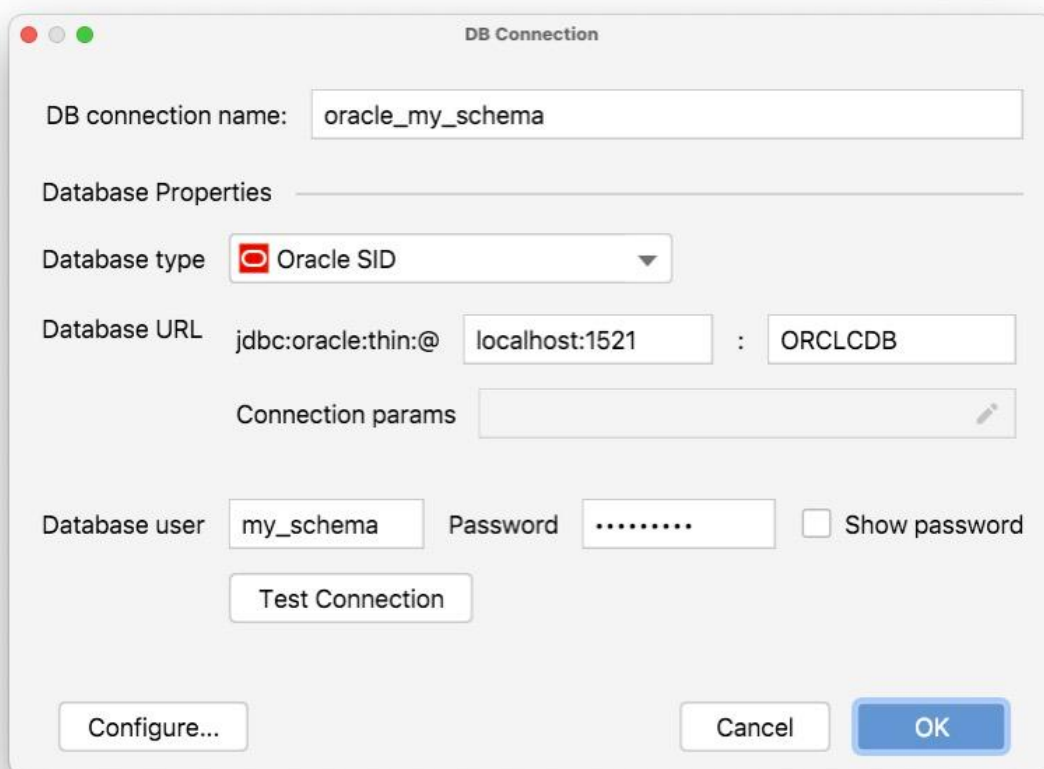
А для [JTDS](#), вот так:



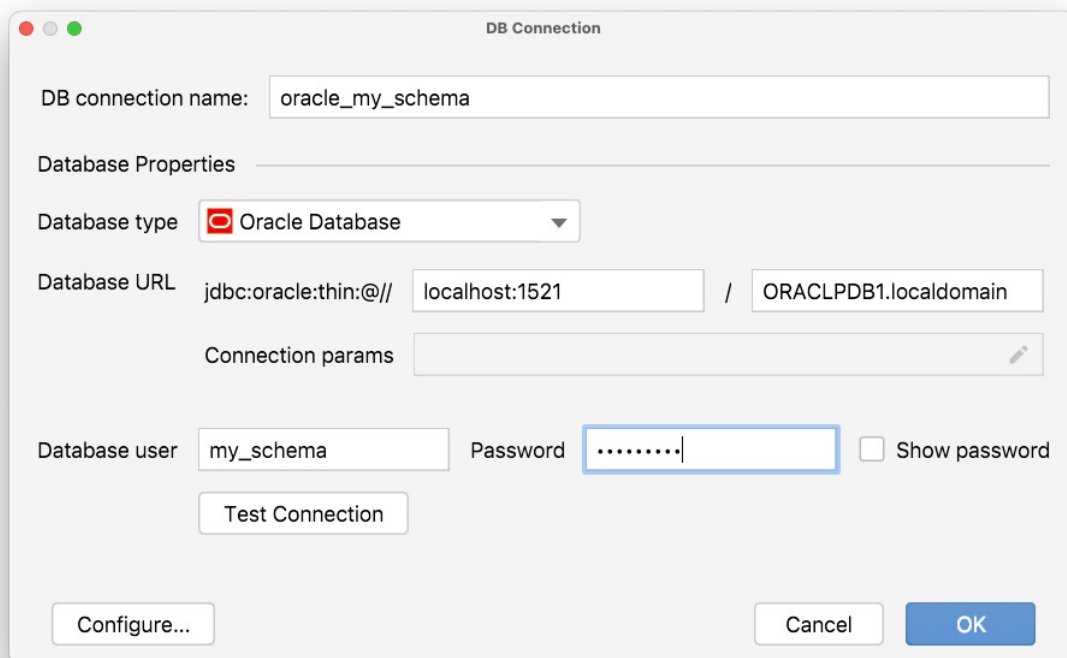
Oracle

В Oracle схема, пользователь и база данных — это одно и то же. Поэтому для подключения к схеме, отличной от схемы по умолчанию, необходимо указать ее имя в поле пользователя.

Для подключения через SID настройка будет выглядеть так:



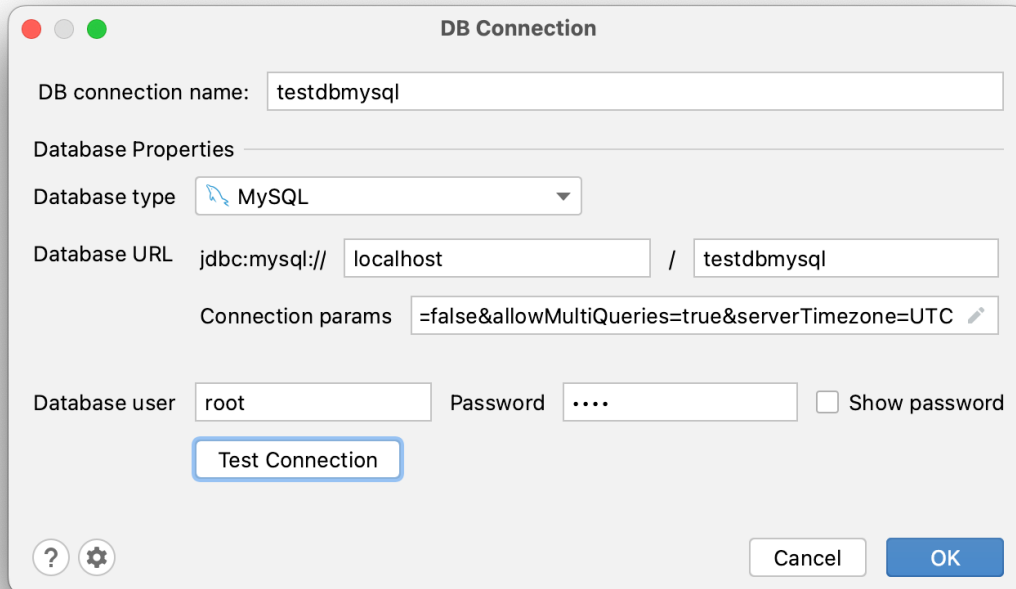
А для подключения через имя сервиса вот так:



Обратите внимание, что Reverse engineering не работает для системных таблиц, расположенных в схеме «SYS».

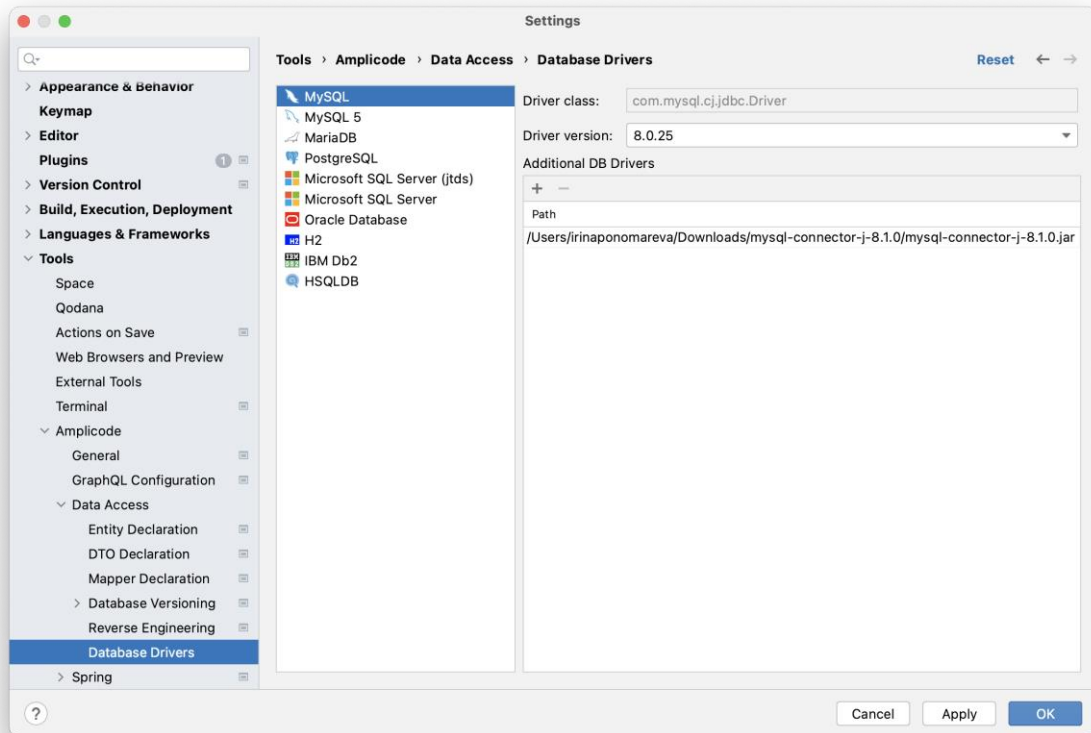
MySQL & MariaDB

Для подключения к схеме, отличной от схемы по умолчанию, необходимо указать имя схемы в поле URL базы данных:



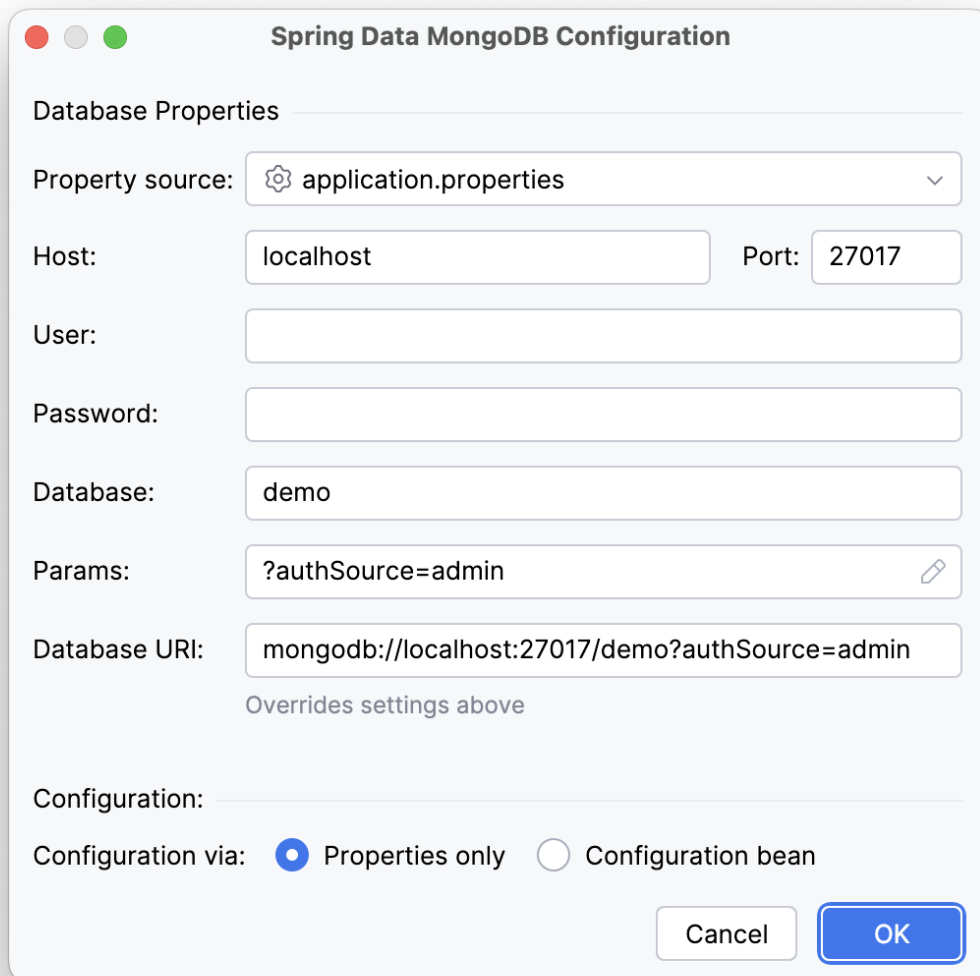
Драйверы баз данных

Поскольку IntelliJ IDEA Community Edition не позволяет настраивать драйверы баз данных, Amplicode предлагает вам альтернативу. Откройте Settings → Tools → Amplicode → Data Access → Database Drivers. Здесь вы можете настроить драйверы для каждой поддерживаемой СУБД, выбрав одну из предложенных версий драйверов и добавив дополнительные файлы с локальной машины.



Соединение с MongoDB

Панель Amplicode Explorer позволяет сконфигурировать использование MongoDB в проекте. Для использования MongoDB в проекте, нажмите кнопку «Плюс», выберите Spring -> Configurations -> Spring Data MongoDB Configuration и укажите параметры соединения в группе Database Properties.



The image shows a 'Spring Data MongoDB Configuration' dialog box. It has a title bar with standard macOS window controls (red, yellow, green buttons) and the title 'Spring Data MongoDB Configuration'. The dialog is divided into two sections: 'Database Properties' and 'Configuration'.
In the 'Database Properties' section, there are several fields:

- 'Property source': A dropdown menu with a gear icon and the text 'application.properties'.
- 'Host': A text input field containing 'localhost'.
- 'Port': A text input field containing '27017'.
- 'User': An empty text input field.
- 'Password': An empty text input field.
- 'Database': A text input field containing 'demo'.
- 'Params': A text input field containing '?authSource=admin' with a pencil icon on the right.
- 'Database URI': A text input field containing 'mongodb://localhost:27017/demo?authSource=admin'.

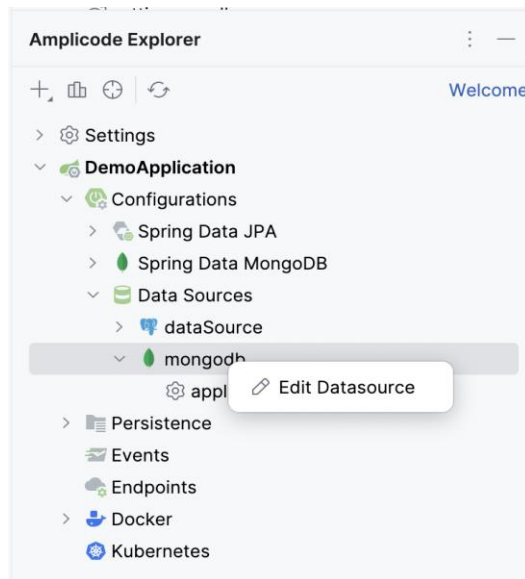
Below the 'Database URI' field, there is a note: 'Overrides settings above'.
In the 'Configuration' section, there is a label 'Configuration:' followed by a radio button selection:

- 'Configuration via:': A label followed by two radio buttons. The first is selected (filled blue circle) and labeled 'Properties only'. The second is unselected (empty circle) and labeled 'Configuration bean'.

At the bottom right of the dialog, there are two buttons: 'Cancel' and 'OK'.

По нажатию ОК проект настроится по требованиям Spring Data MongoDB и возможно будет начать работу с MongoDB документами и Spring Data MongoDB репозиториями.

В дальнейшем параметры соединения с Mongo возможно изменить через Amplicode Explorer найдя в дереве элемент Configurations -> Datasources -> mongoDB и выполнив действие Edit Datasource.



Версионирование баз данных

Чтобы гарантировать, что модель данных в исходном коде соответствовала относящимся к приложению базам данных, чрезвычайно важно поддерживать их в синхронизированном состоянии. Для того, чтобы справиться с этой задачей, существуют два распространенных подхода.

Начинаем с базы данных

Когда используется этот подход, база данных получает приоритет над классами модели данных (POJO или JPA сущностями), которые генерируются из схемы базы данных через процесс генерации кода, известный как Database reverse engineering. Важно избегать каких-либо изменений в сгенерированных классах, поскольку они могут быть регенерированы в любое время, и все изменения, сделанные в коде, пропадут. Однако, этот подход не устраняет необходимость генерации скриптов миграции, поскольку они нужны для обновления существующих инсталляций до последней версии.

Начинаем с исходного кода

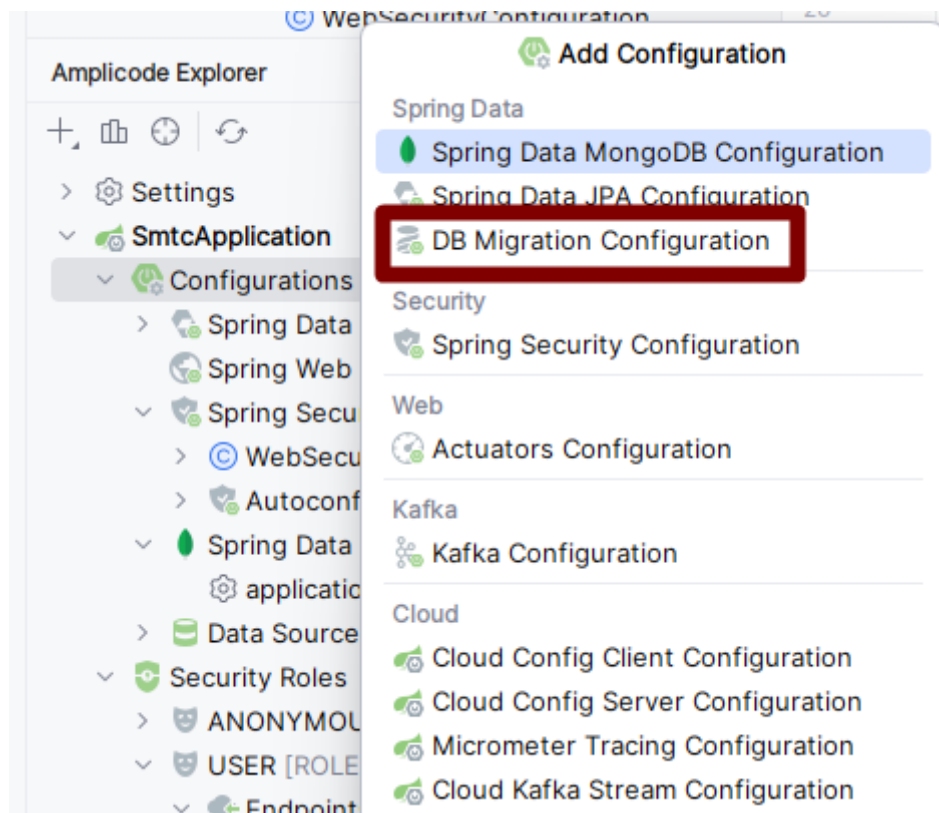
Это обратный подход; здесь классы модели данных являются единственным источником истины. Следовательно, база данных модифицируется в зависимости от изменений, сделанных в модели данных. Чтобы обновить базу данных, скрипт миграции должен отражать изменения, отличающие устаревшее состояние базы данных от текущего состояния классов модели данных в любом формате, например, в формате ченджлогов Liquibase. Amplicode предоставляет удобные инструменты для помощи разработчикам в работе по любому из этих двух сценариев. Данный гайд показывает, как Amplicode может сэкономить вам кучу времени для генерации скриптов обновления по изменениям.

Первое, что вам необходимо сделать, чтобы использовать функционал версионирования баз данных — это создать подключение к БД. В предыдущей главе описано, как это сделать правильно и упоминаются все возможные проблемы, с которыми вы можете столкнуться.

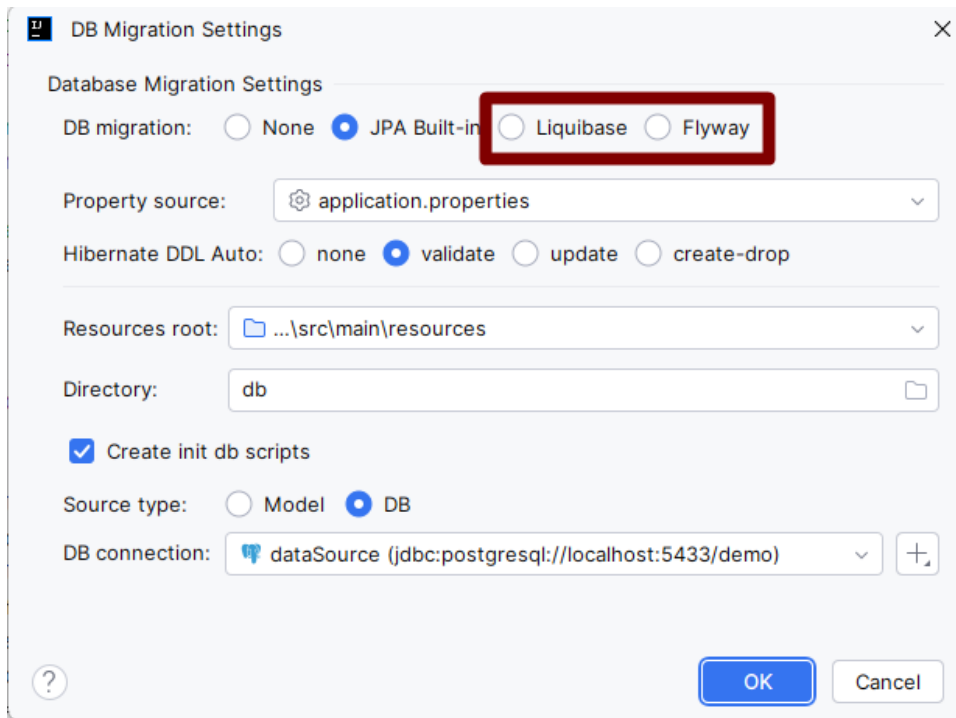
Поддержка различных библиотек версионирования баз данных

Amplicode поддерживает два популярных решения, который часто используются в приложениях на Java вместе с JPA: [Flyway](#) и [Liquibase](#). Однако, присутствует также опция для получения DDL скриптов для ваших JPA сущностей, даже если ни одно из этих решений не используется в проекте.

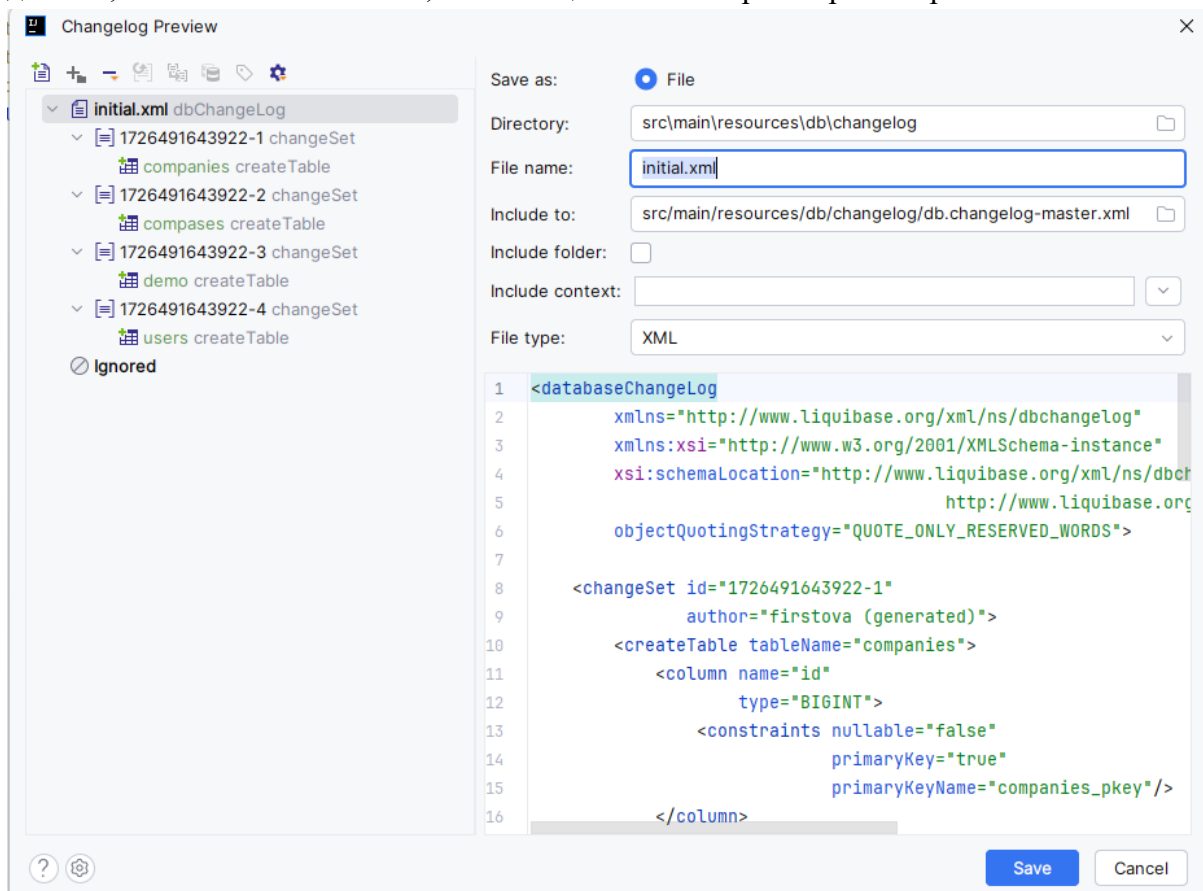
Для добавления нужной зависимости в проект необходимо добавить конфигурацию DB Migration Configuration. Для этого в дереве Amplicode Explorer нужно выделить Configurations, кликнуть по нему правой кнопкой мыши, выбрать Add Configuration, в открывшемся окне выбрать DB Migration Configuration.



В открывшемся окне нужно выбрать необходимую опцию.



В зависимости от выбора в окне будут отображены дополнительные опции для настройки. Нажмите Ок для добавления. После нажатия в проект добавится зависимость и сгенерируются инит скрипты на основе модели из проекта или из базы данных, в зависимости от того, какая опция была выбрана при настройке.



Стандартная последовательность действий при генерации дифференциальных скриптов

Обычная последовательность действий при генерации дифференциальных скриптов (так называются скрипты, фиксирующие различия между базой данных и классами модели) примерно одинакова для Flyway и Liquibase. Начиная с этого момента, мы будем называть их “миграционными скриптами”. Процесс создания миграционных скриптов будет показан на примере Liquibase.

Чтобы сгенерировать дифференциальный миграционный скрипт с помощью Amplicode, нажмите на правую кнопку мыши на нужном каталоге и выберите New → Liquibase Diff Changelog. В качестве альтернативы можно нажать на кнопку Plus над панелью Amplicode Explorer и выбрать соответствующий пункт из подменю DB Versioning.

В открывшемся диалоге выберите source (источник, желаемое состояние модели данных) и target (цель, старое состояние модели данных).

Полученные миграционные скрипты = Текущее состояние (Source, источник) – Предыдущее состояние (Target, цель).

Другими словами, Amplicode сгенерирует результирующие миграционные скрипты для обновления целевой базы данных до состояния источника.

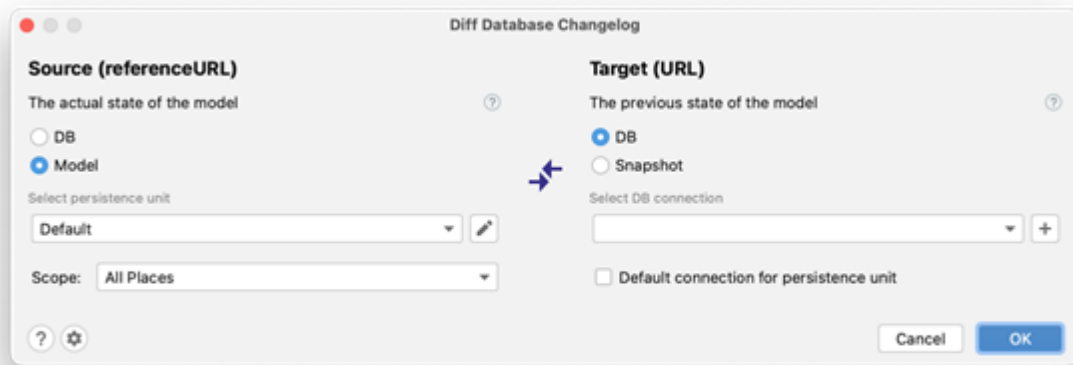
Можно выбирать между следующими опциями:

- DB — должна использоваться в случае, когда у вас есть база данных, чье состояние соответствует текущему, и вы хотите сгенерировать миграционные скрипты, чтобы обновить другую БД до того же состояния.
- Model — используйте эту опцию для генерации миграционных скриптов, отражающих разницу между текущим состоянием модели взаимоотношений между сущностями (JPA сущности) и старым (target) состоянием.

Target можно устанавливать в:

- DB — целевая БД со старой версией схемы.
- Snapshot — используйте эту опцию в случае, когда у вас есть желаемое состояние данных, сохраненное в snapshot (снимке) модели данных. Snapshot тоже можно генерировать при помощи Amplicode.

Нажмите ОК, чтобы пойти дальше. Amplicode проанализирует различия между Source и Target и выведет диалоговое окно Preview, чтобы разрешить тонкую настройку будущего миграционного скрипта. Нажмите Save чтобы добавить в проект новый скрипт или дописать новую информацию в конец существующего.



Опции при генерации дифференциальных миграционных скриптов

Использование базы данных

Если у вас есть база данных, выступающая в роли источника и уже синхронизированная с моделью данных, имеет смысл сравнить эту базу данных с другой базой данных или снимком (snapshot). Существует два популярных подхода к поддержанию базы данных в состоянии соответствия JPA сущностям:

- Используя автогенераторы схем (Hibernate и EclipseLink предоставляют свои собственные реализации; заметьте, что даже документация Hibernate не советует использовать это метод, за исключением случаев прототипирования или тестирования).
- Применяя изменения в JPA сущностях к схеме базы данных вручную (этот метод кажется слишком трудоемким, особенно на ранних стадиях разработки, когда модель данных меняется довольно часто).

Использование модели данных

Приложение использует JPA сущности для представления модели данных, включая сущности, ассоциации, индексы и другие относящиеся к модели данных элементы в соответствии с принципами JPA. Другими словами, оно уже содержит достаточно информации о схеме базы данных. Таким образом, ваш исходный код является единственным источником истины, который представляет текущее состояние схемы (source). Именно поэтому является предпочтительным подход, при котором модель данных сравнивается с базой данных или со snapshot с целью генерации дифференциальных скриптов.

Amplicode сканирует все JPA объекты, сравнивает их с целевой базой данных или snapshot и генерирует дифференциальный миграционный скрипт.

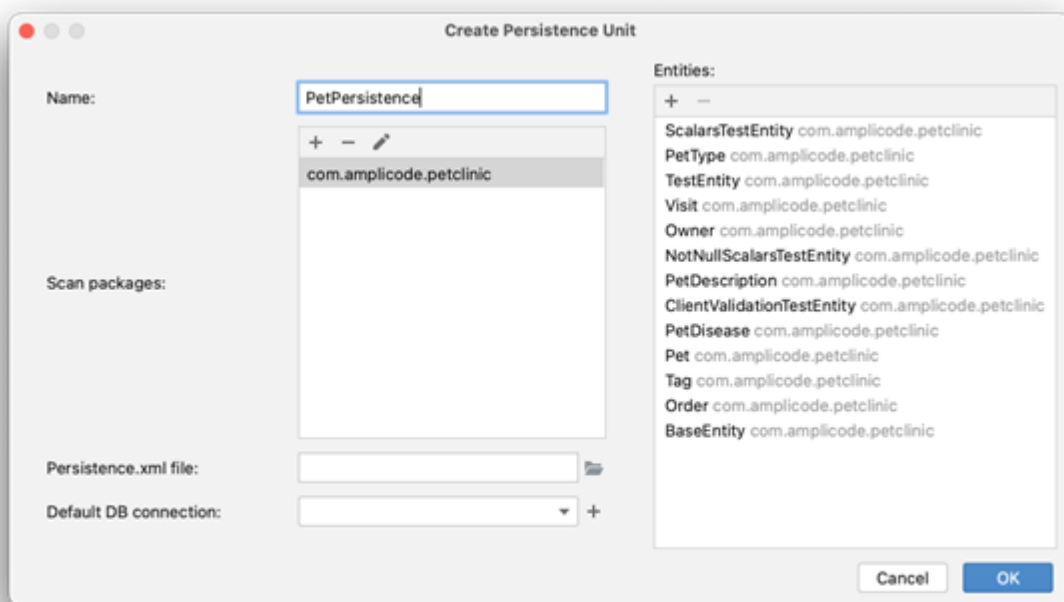
Когда модель данных служит источником для текущего состояния схемы, это приводит к необходимости выбора persistence unit. Цитата из документации:

Persistence unit задает набор всех классов сущности, которые управляются экземплярами *EntityManager* внутри приложения. Этот набор классов сущности представляет данные, помещенные в единое хранилище данных.

По сути, это означает, что, если ваше приложение использует несколько хранилищ данных, вам придется генерировать миграционные скрипты для каждого из них, задавая соответствующие *persistence units*.

Чтобы сконфигурировать новый *persistence unit*, нажмите на кнопку плюс в панели Amplicode Explorer и выберите Database → Persistence Unit. В открывшемся окне вы можете задать имя для *persistence unit*, подключение к базе данных по умолчанию, а также выбрать требуемые сущности. Для выбора сущностей у вас будут две возможности:

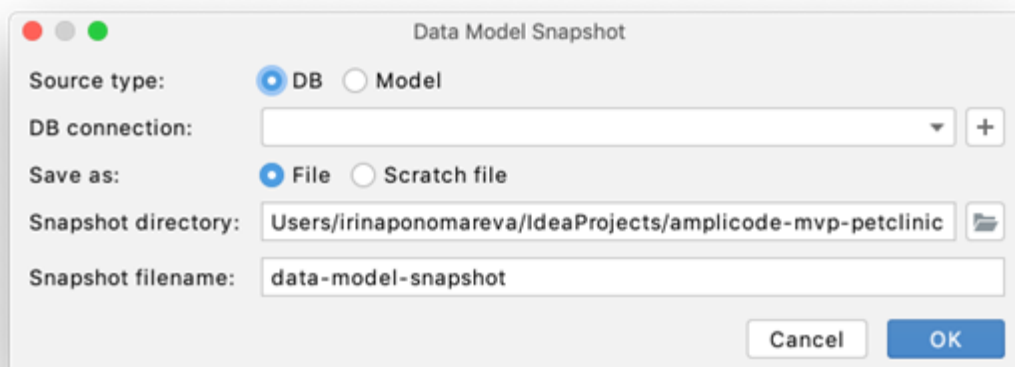
- Вы можете просканировать нужный пакет, и все размещенные в нем сущности будут добавлены автоматически.
- Вы можете вручную добавить сущности из проекта.



Использование snapshot модели данных

Amplicode позволяет использовать *snapshot* (снимок) модели данных как целевой объект при сравнении. Иногда бывает невозможно или слишком трудно получить саму базу данных для определенного состояния модели, например, при заливке (*merge*) изменений в более раннюю версию приложения. Это может оказаться просто неосуществимо, если мы попытаемся хранить дампы базы данных для каждого релиза. Amplicode позволяет вам выполнить действие *checkout* для нужной версии приложения и сгенерировать *JSON-snapshot*, базирующийся на JPA сущностях, тем самым устраняя необходимость использования базы данных при создании дифференциальных миграционных скриптов.

Чтобы сгенерировать snapshot, откройте панель Amplicode Explorer и нажмите на кнопку плюс. Затем выберите пункт меню DB Versioning → Data model snapshot.



Это позволяет вам сделать снимок состояния модели данных в какой-то момент из прошлого, таким образом, чтобы вы могли создать дифференциальный миграционный скрипт, описывающий все модификации, произошедшие с того момента и по настоящее время.

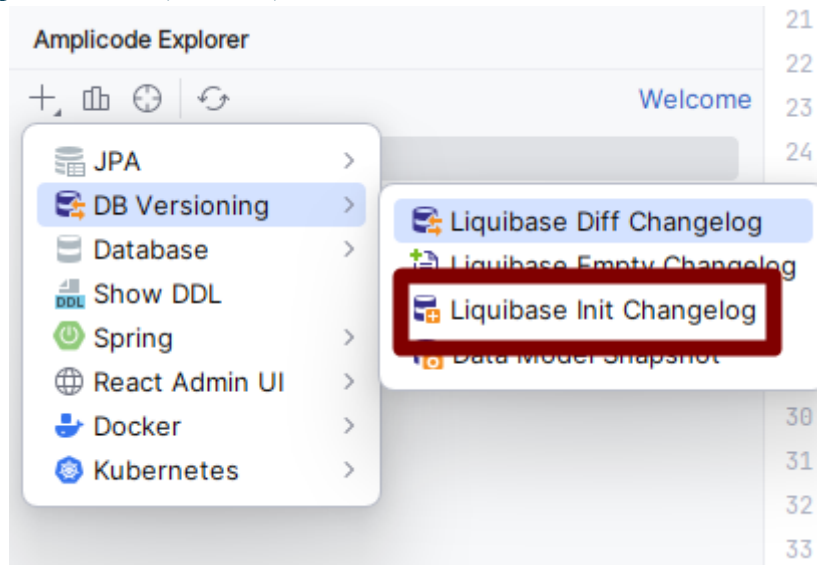
Например, вы работали над функциональной веткой (feature branch) и модифицировали модель. Перед слиянием необходимо создать дифференциальный скрипт, описывающий изменения только в этой ветке.

В зависимости от настроек может не существовать базы данных, которая всегда синхронизирована с веткой main. Ситуация становится еще более сложной, когда необходимо залить изменения в некое состояние приложения, отличное от ветки main, например, в релизную ветку. Поддерживать дампы базы для каждого релиза может оказаться непрактичным решением. Amplicode предлагает более простую альтернативу:

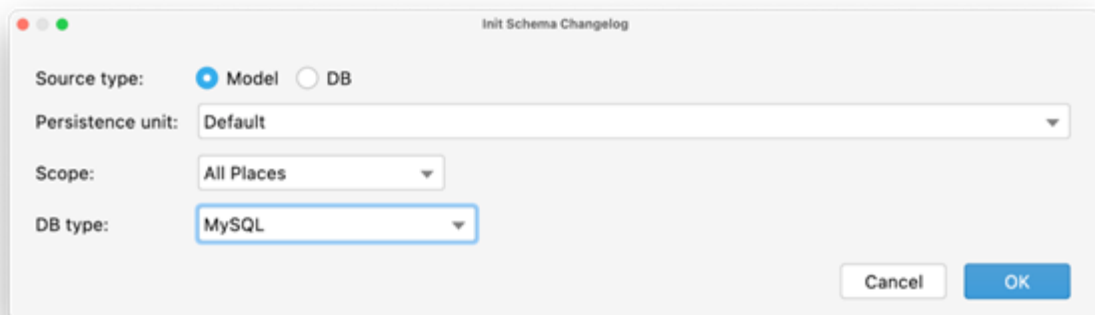
1. Выполнить checkout на целевую ветку (например, main или release).
2. Сделать snapshot модели в этой ветке.
3. Выполнить checkout на функциональную ветку (feature branch).
4. Сгенерировать дифференциальный миграционный скрипт через сравнение модели со snapshot, который вы создали на шаге 2.

В четыре простых шага вы получаете миграционный скрипт, который описывает изменения между текущей веткой и целевой веткой.

Генерация скриптов инициализации

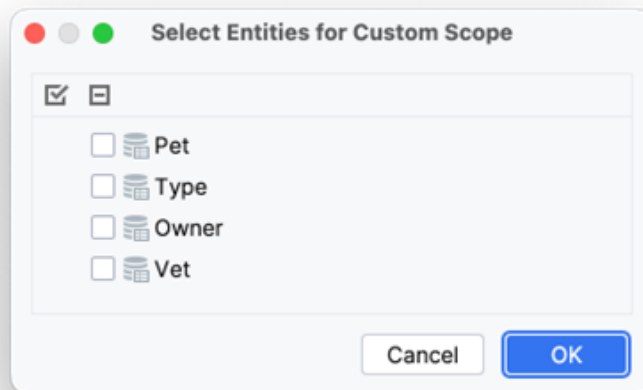


Для Liquibase и Flyway Amplicode предоставляет действие генерации скрипта инициализации для схемы. Когда вы выбираете это действие из панели Amplicode Explorer, появится соответствующее окно. По умолчанию Source type устанавливается в DB, но, если вы переключитесь на Model, окно будет выглядеть следующим образом:

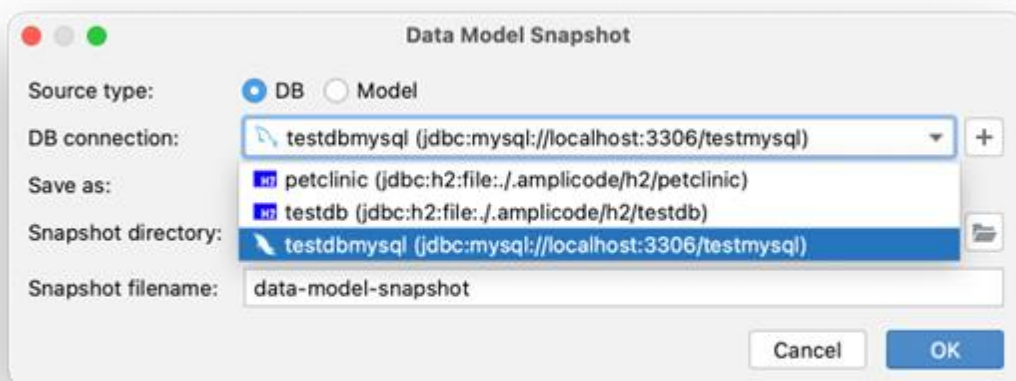


Чтобы сгенерировать DDL скрипт на базе модели данных, вам необходимо задать соответствующий persistence unit, охват (scope) и один из поддерживаемых типов БД.

В дополнении к сказанному Amplicode предоставляет опцию создания миграционных скриптов специально для изменений в выбранных сущностях как кастомизированный охват (custom scope). Это можно сделать, нажав на выпадающее меню Scope, выбрав Selected Entities и отметив нужные сущности в окне Select Entities for Custom Scope.



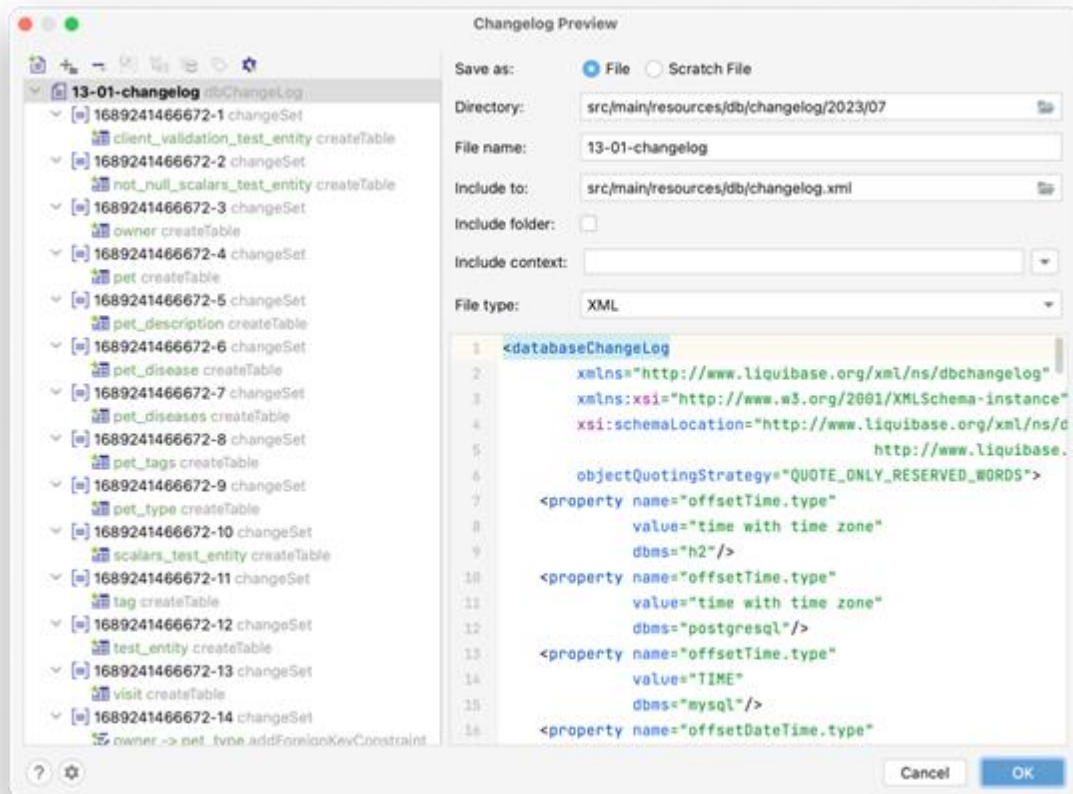
В случае, если вы хотите сравнить две базы данных, вам придется выбрать одно из существующих подключений для обеих БД.



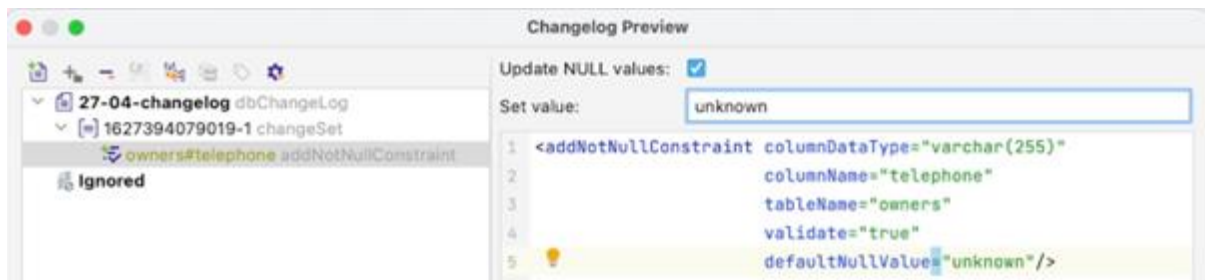
Окно предпросмотра миграционного скрипта

Нажмите ОК, чтобы перейти к окну предпросмотра (preview) миграционного скрипта.

Окно предпросмотра выглядит следующим образом:

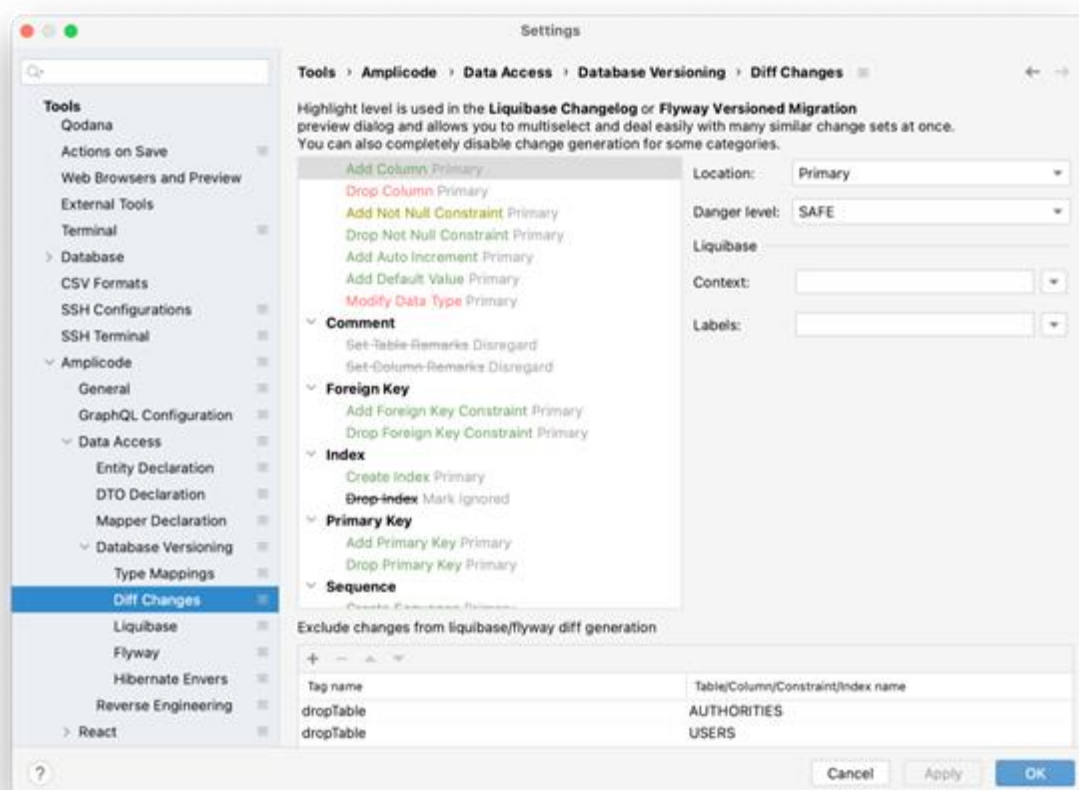


Некоторые типы изменений имеют кастомизированные поля в окне предпросмотра. Например, изменение `add not null constraint` (добавить ограничение недопустимости значения NULL) позволяет вам заменить все существующие значения NULL в базе данных на заданное значение:



Каждое изменение кодируется цветом в соответствии со своим уровнем опасности: зеленое для SAFE (безопасно), желтое для CAUTION (осторожно) и красное для DANGER (опасно). Операции уровня SAFE — это те операции, которые не могут никаким образом вызвать потерю данных, например, добавление колонки не влияет на существующие данные. Операции, помеченные как CAUTION, обычно безопасны, но требуют вашего внимания: например, добавление ограничения NOT NULL может не сработать, если в колонке есть значения NULL. Операции с уровнем DANGER могут вызвать потерю данных, например, удаление колонки или изменение типа данных.

Уровни опасности могут быть кастомизированы в настройках плагина в разделе Settings → Tools → Amplicode → Data Access → Database Versioning → Diff Changes:



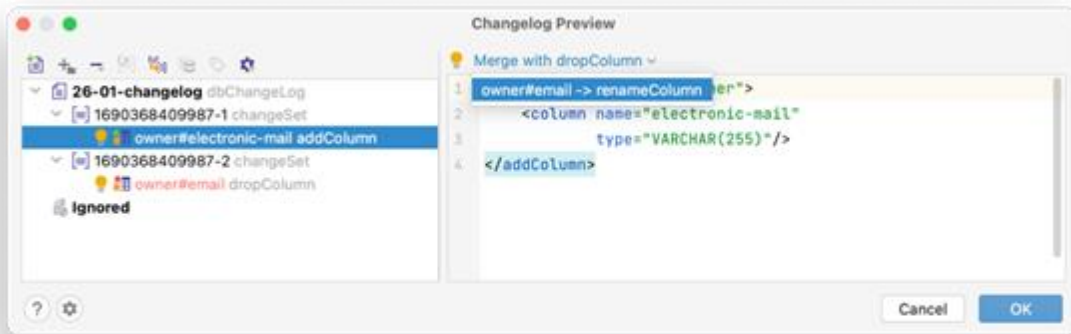
Вы можете сконфигурировать местоположение каждого типа изменений, либо первичное, либо вторичное местоположение, либо полностью проигнорировать эту настройку. По умолчанию, только что сгенерированные миграционные скрипты будут исключать игнорируемые изменения либо отображать их в разделе Ignored во время предпросмотра, чтобы их можно было добавить обратно вручную. Для Liquibase вы можете также установить контекст и метки, чтобы использовать их для каждого типа изменений.

Объединение действий в скрипте

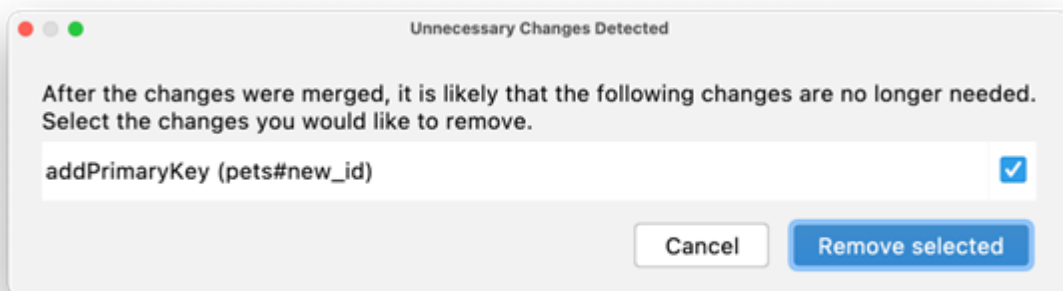
Как правило, переименование элементов схемы, таких как таблица, колонка и т.д., ведет к двум действиям:

- Удалить существующее значение
- Добавить новое.

Но Amplicode может заменить такие действия одиночным действием переименования или модификации. Например, вы увидите два действия в окне предпросмотра после переименования колонки/таблицы/последовательности или изменения типа колонки. Но, выбрав любые связанные между собой действия, вы можете объединить их. Просто воспользуйтесь функцией Merge with dropColumn в окне Changelog Preview.



После слияния действие удаления (drop) может оказаться ненужным. Вы можете выбрать изменения, которые вы хотите удалить из миграционного скрипта. Например, после переименования колонки id (вместо удаления старого значения и добавления нового) добавление нового первичного ключа уже не понадобится:

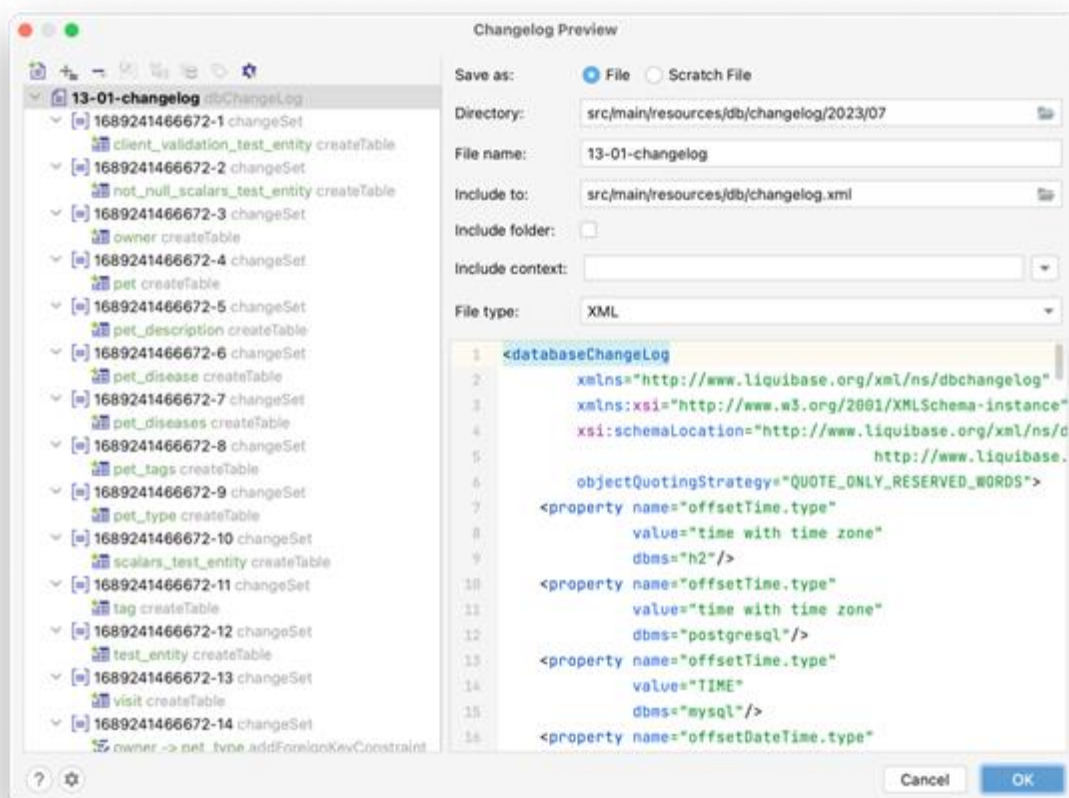


Поддержка Liquibase

Liquibase — это независимая от типа базы данных библиотека, которая помогает нам отслеживать, управлять и применять изменения в схеме базы данных. Все изменения в базе данных сохраняются в текстовых файлах (changelogs). Всякий раз, когда необходимо обновить базу данных, происходит обращение к списку changelogs, чтобы определить, какие изменения должны примениться. Liquibase является open-source решением.

Чтобы узнать об этой библиотеке больше, вы можете обратиться к документу [Introduction to Liquibase - Liquibase Documentation](#).

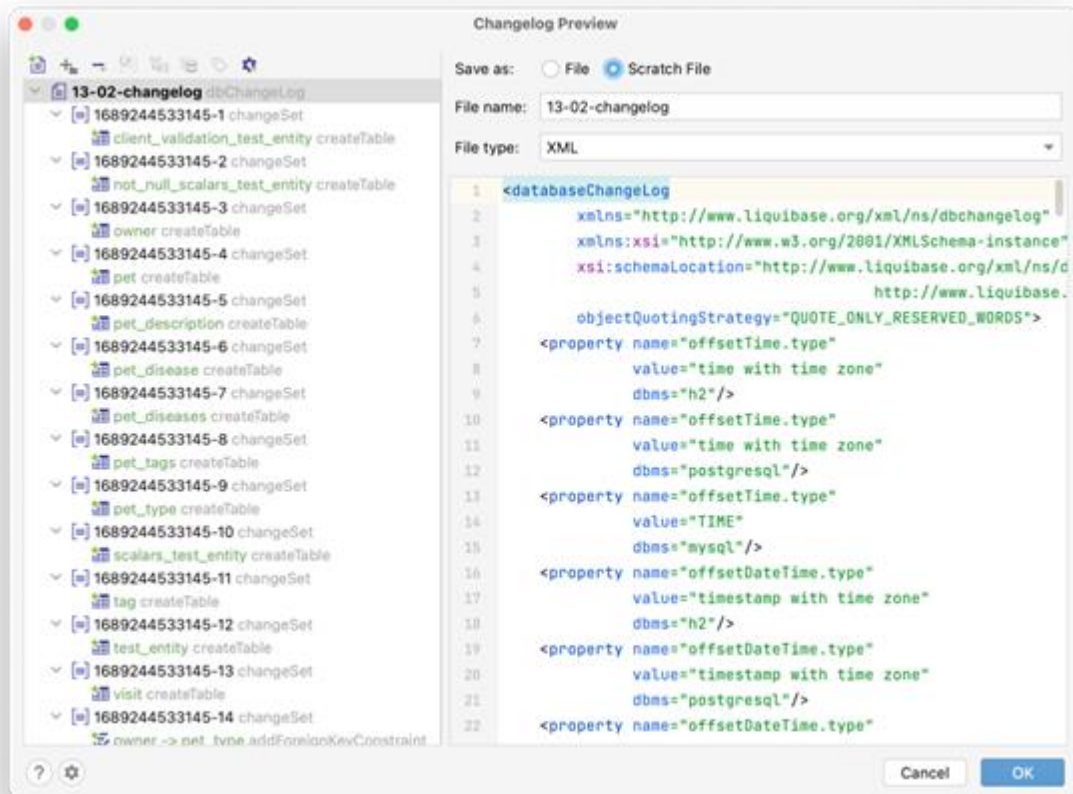
Окно предпросмотра changelog



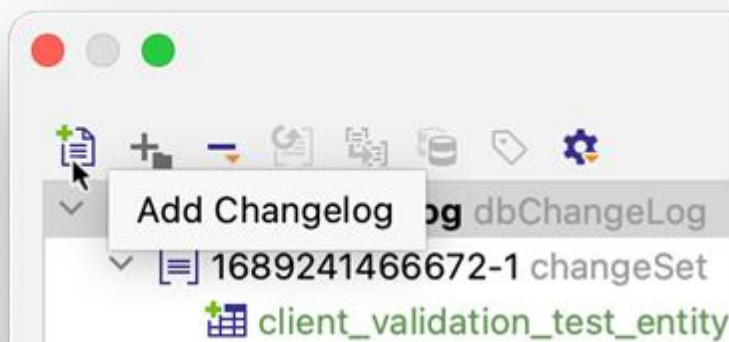
Если вы хотите сохранить changelog как регулярный файл, будут доступны следующие конфигурационные опции:

- Поля Directory и File name отвечают за конфигурацию местоположения сгенерированного changelog. Если changelog с заданным именем уже существует, вам будет показано предупреждение, после чего новые изменения будут дописаны в конец этого changelog.
- Вы можете использовать Include to, Include folder и Include context, чтобы указать, будет ли changelog включен в другой changelog. Если вы отметите чекбокс Include folder, будет сгенерировано действие include для всего каталога, а не только для текущего changelog.
- Из выпадающего списка File type вы можете выбрать один из четырех типов файлов (YAML, JSON, SQL, XML), поддерживаемых Liquibase, и Amplicode сгенерирует changelog в выбранном формате.

Если вы хотите сохранить changelog как временный файл (scratch file), вы можете сконфигурировать только его имя и тип.



В левой части окна отображается предпросмотр changesets (наборов изменений), которые будут сгенерированы. Вы можете щелкнуть мышью на каждом изменении, чтобы посмотреть, как оно выглядит. Чтобы скомбинировать несколько изменений в один changeset или поместить их в игнорируемые, просто перетащите их методом drag-and-drop. Верхний левый угол окна предпросмотра содержит иконки различных действий, позволяющих модифицировать результирующий changelog:



Набор иконок содержит следующие действия:

- Add Changelog — создать вторичный changelog.
- Add Change Set — создать новый changeset внутри выбранного changelog.

- Remove from Changelog с опциями:
 - Remove from Changelog — просто удалить изменения из текущего changelog.
 - Remove and Ignore — удалить изменения и переместить их в Ignored, чтобы они были исключены также из будущих changesets.
 - Restore from Ignored — переместить изменения из Ignored в changelog.
- Set Context (для changesets)
- Set Labels (для changesets)
- Show Other Actions — выбрать все изменения на базе уровня, развернуть/свернуть все изменения.

Первичные и вторичные changelogs

Amplicode позволяет складывать изменения в два типа changelogs: первичный (Primary) и вторичный (Secondary). Один из примеров использования этого функционала — это разделение безопасных изменений, которые можно запускать автоматически, и тех изменений, которые требуют вашего внимания и должны запускаться вручную.

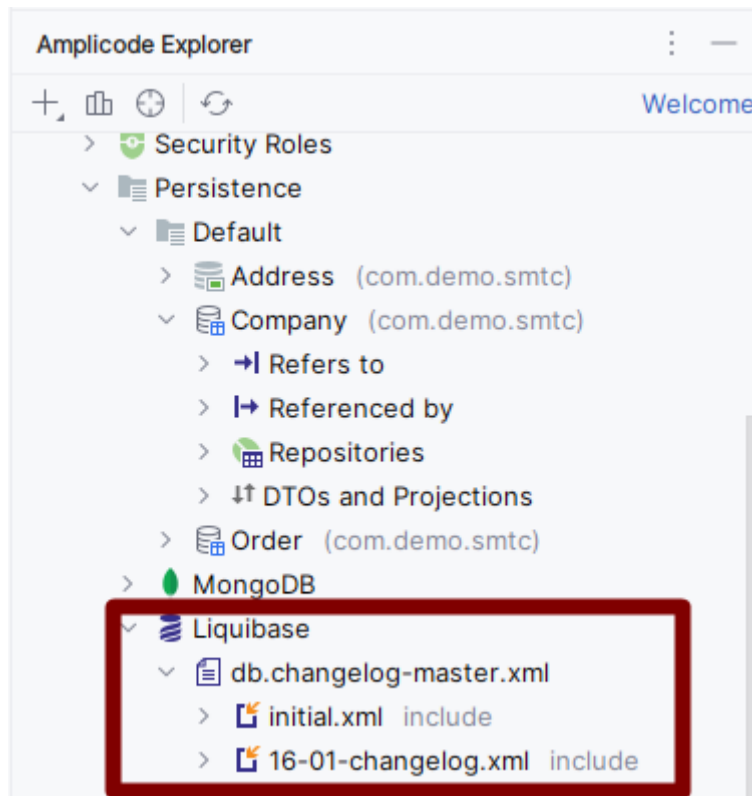
Изменения можно автоматически разделить по типу (воспользовавшись Settings → Diff Change Types). Есть и другой способ: вы можете создать вторичный changelog вручную в окне Preview, используя действие Add Changelog. Затем вы можете просто перетащить нужные вам changesets в новый changelog.

По умолчанию, первичные и вторичный changelogs генерируются в отдельных директориях, что можно поменять в настройках плагина. Вы можете прочесть об этом подробнее в разделе Settings → Database Versioning → Liquibase.

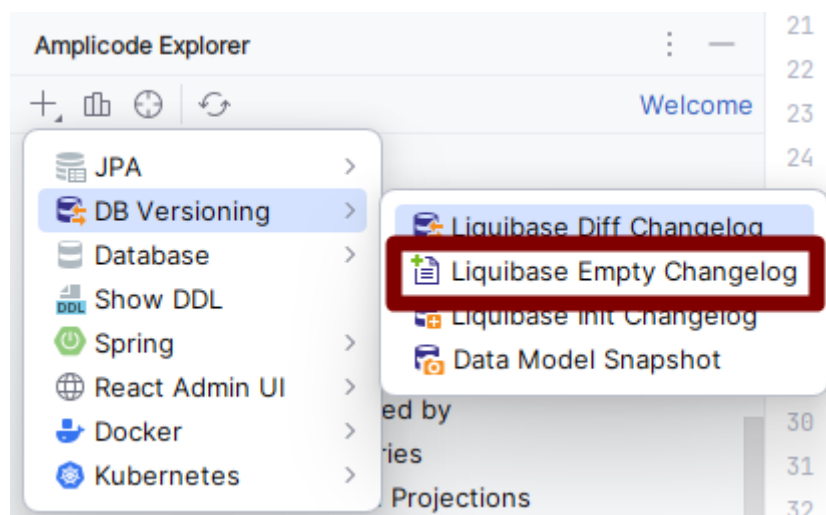
Создание и изменение changelogs

Amplicode также предлагает инструменты для просмотра, создания и изменения changelogs вручную. Она добавляет дополнительные элементы графического интерфейса к IntelliJ IDEA.

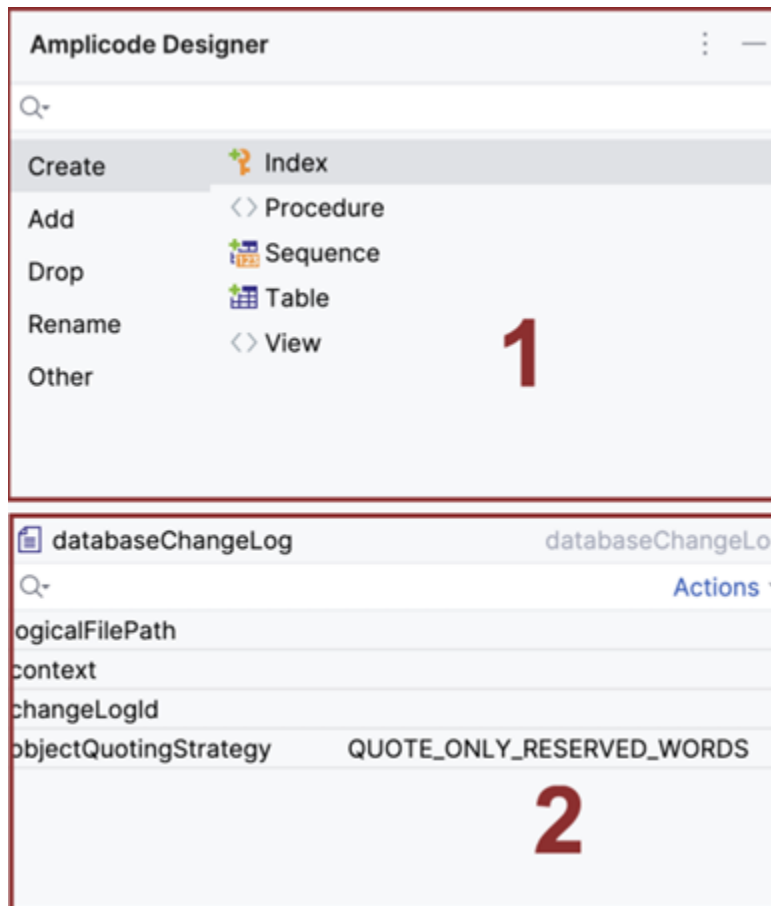
Панель Amplicode Explorer отображает все, что Amplicode знает о проекте. В случае с Liquibase changelogs, она показывает их иерархию и содержимое:



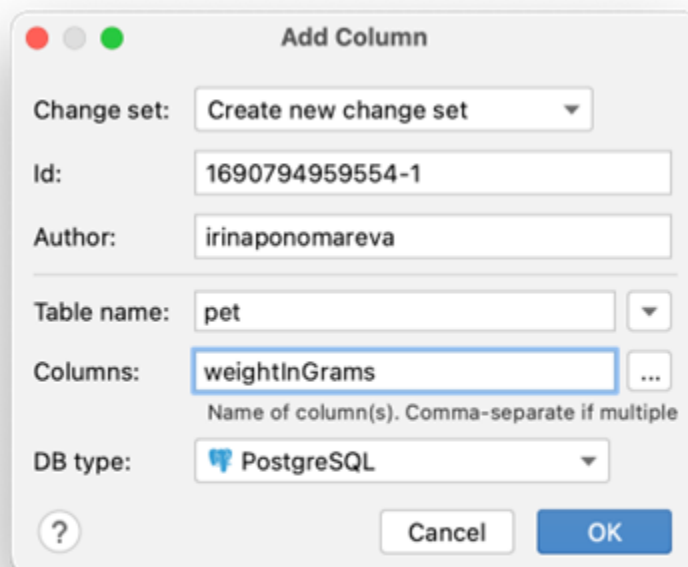
Чтобы создать новый changelog, в дереве проекта на выбранной директории по правому клику мыши выполните действие New → Other → Liquibase Empty Changelog. Или в Amplicode Explorer выполните действие по кнопке Плюс → DB Versioning → Liquibase Empty Changelog:



Панель Designer позволяет генерировать и модифицировать код. Эта панель контекстно-зависима и отображает опции, доступные для типа открытого файла. Эта панель состоит из двух частей: Палитра (1) и Инспектор (2). Палитра используется для генерации кода, а Инспектор для его модификации.



Amplicode понимает вашу модель данных и максимально возможно предзаполняет changesets. А с помощью Инспектора вы можете просматривать атрибуты каждого элемента changelog.



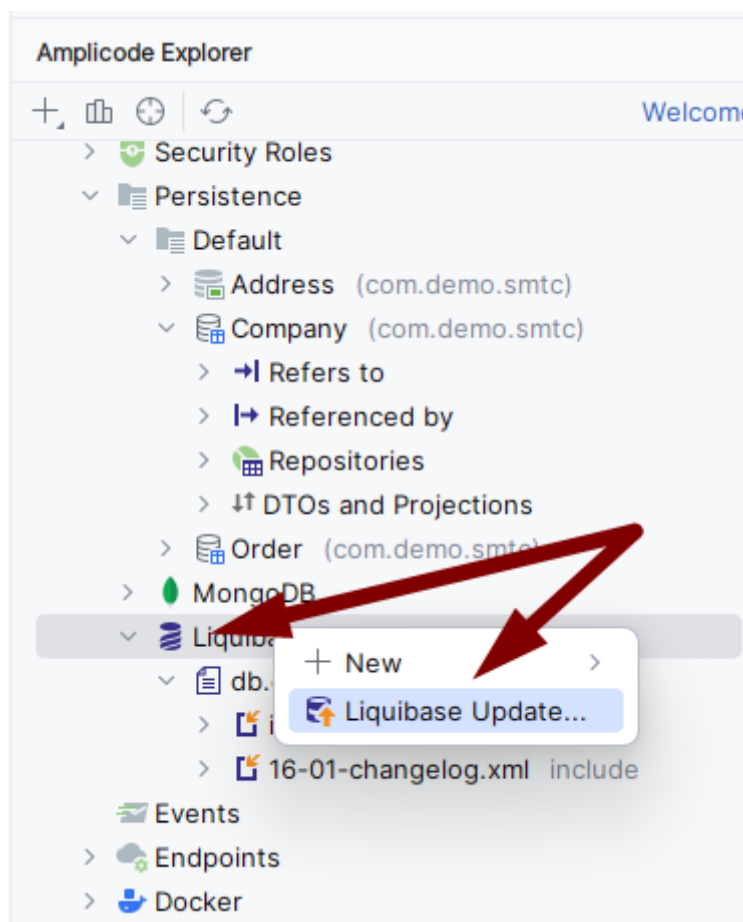
Amplіcode также упрощает задачу написания кода вручную, предлагая автодополнение код на базе модели данных JPA.

Пример:

```
<createTable tableName="pet">
  <column name="id"></column>
</createTable>
</changeSet>
<changeSet id="1691391">
  <addForeignKeyConstraint baseTableName="owner" constraintName="FK_
  owner_id
  type_id
  birth_date
  pet_description_id
  weight_in_grams
  Long
  String
  Owner
  PetType
  LocalDate
  PetDescription
  Integer
  Press ↵ to insert, ⇧ to replace Next Tip
  </dropForeignKeyConstraint baseTableName="owner" constraintName="FK_
```

Выполнение Liquibase changelogs и предпросмотр SQL без плагинов Gradle/Maven

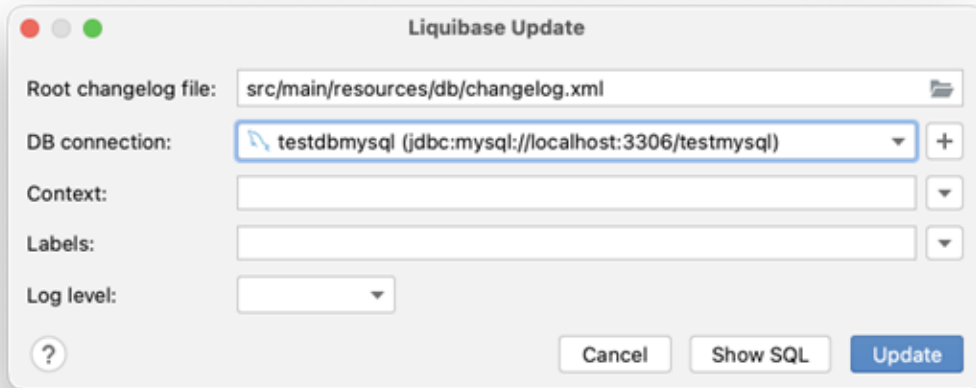
Панель Amplіcode Explorer также предоставляет возможность запускать обновления Liquibase и осуществлять предпросмотр SQL. Чтобы запустить обновление, нажмите на Liquibase Update (для этого разверните Persistence и нажмите правую клавишу мыши на разделе Liquibase внутри дерева Amplіcode Explorer).



Откроется окно Liquibase Update со следующими конфигурационными опциями:

- Путь к changelog файлу

- Какое подключение к БД использовать
- Контекст и метки Liquibase
- Уровень логирования операции.



Нажатие на кнопку Update запускает команду обновления Liquibase с параметрами, выбранными в процессе конфигурации обновления. Нажатие на кнопку Show SQL открывает окно предпросмотра:

DDL Preview

```
-- Run at: 7/17/23, 9:07 AM
-- Against: root@localhost@jdbc:mysql://localhost:3306/testmysql
-- Liquibase version: 4.18.0
-----

-- Create Database Lock Table
CREATE TABLE DATABASECHANGELOGLOCK
(
  ID INT NOT NULL,
  'LOCKED' BIT(1) NOT NULL,
  LOCKGRANTED datetime NULL,
  LOCKEDBY VARCHAR(255) NULL,
  CONSTRAINT PK_DATABASECHANGELOGLOCK PRIMARY KEY (ID)
);

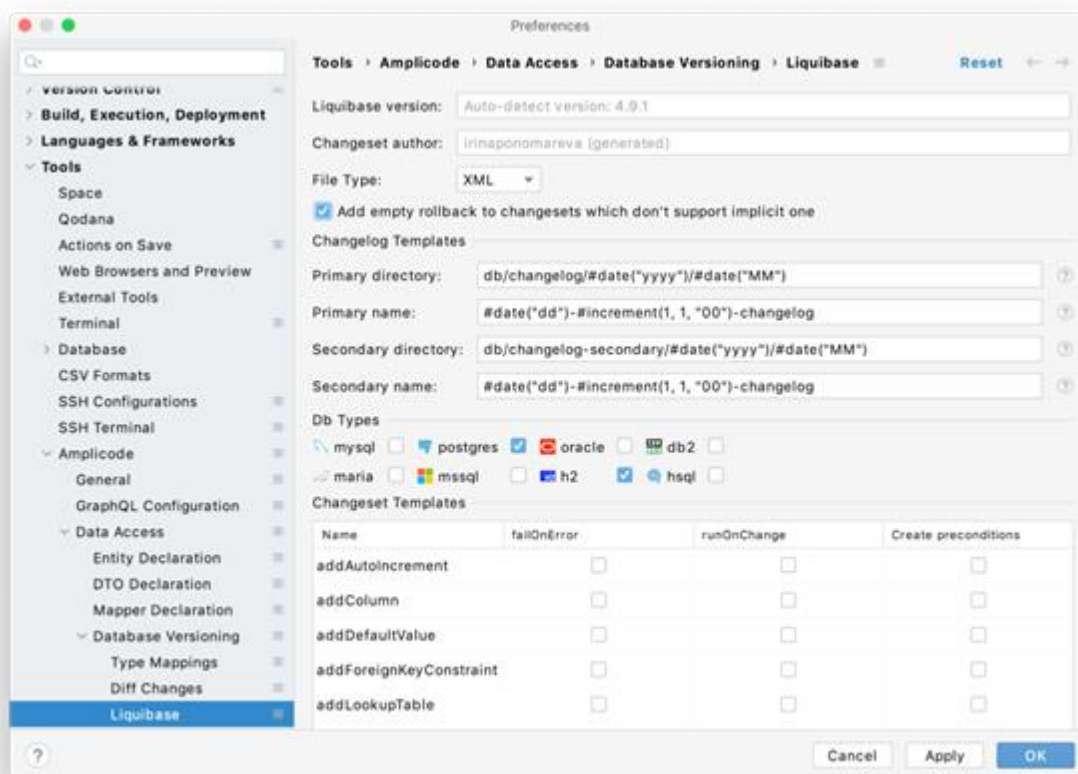
-- Initialize Database Lock Table
DELETE
FROM DATABASECHANGELOGLOCK;

INSERT INTO DATABASECHANGELOGLOCK (ID, 'LOCKED')
VALUES (1, 0);

-- Lock Database
UPDATE DATABASECHANGELOGLOCK
SET 'LOCKED' = 1,
    LOCKEDBY = 'MacBook-Pro-Irina.local ( )',
    LOCKGRANTED = NOW()
WHERE ID = 1
AND 'LOCKED' = 0;
```

OK

Настройки Liquibase



Базовые настройки

Amplicode позволяет задавать:

- Версию Liquibase, которую вы хотите использовать;
- Имя автора changeseat author name;
- Тип файла по умолчанию. Поддерживаются следующие 4 типа:
 - XML
 - SQL
 - YAML
 - JSON

Шаблоны changelog-файлов

Каждый раз, когда создается пустой или дифференциальный Liquibase changelog, Amplicode генерирует имя файла на основании шаблона. Вы можете сконфигурировать первичную или вторичную директорию и имя для файлов changelogs.

Следующие переменные и макросы помогут сделать процесс конфигурации более точным:

- `#date([format])` — текущая дата в системе в заданном SimpleDateFormat. Например, `#date(\"уууу-ММ-dd\")` возвращает дату в формате 2020-12-31.
- `#increment([start], [step], [decimalFormat])` — число, используемое для поддержания уникальности имен; значение start используется для первого файла

и увеличивается на значение `step` для каждого следующего файла; параметр `decimalFormat` задает десятичный формат числа. Например: `#increment(1.0, 0.1, \"#.0\")` возвращает значение в формате 1.1, 1.2, 1.3 и т.д.

- `semVer` — семантическая версия проекта (также известная как `SemVer`) — это широко распространенная схема для номеров версий, которая использует последовательность из трех цифр (`Major.Minor.Patch` — мажорная версия, минорная версия, номер патча), опциональный предрелизный тег и опциональный метатег сборки. Объект содержит следующие методы (полная версия в примерах — это 1.2.3-SNAPSHOT+meta):
 - `${semVer.getRawVersion()}: 1.2.3-SNAPSHOT`
 - `${semVer.getMajor()}: 1`
 - `${semVer.getMinor()}: 2`
 - `${semVer.getPatch()}: 3`
 - `${semVer.getPreRelease()}: SNAPSHOT`
 - `${semVer.getMeta()}: meta.`

Типы БД

Иногда программное обеспечение должно обеспечивать поддержку для более чем одного типа СУБД. В этом случае `Liquibase` является наилучшим выбором, поскольку он предлагает решение, не зависящее от типа БД, для объявления модификаций DDL. `Amplicode` также поддерживает это решение. При генерации БД-независимых `changelog` она использует свойства `Liquibase`, чтобы задать правильные типы данных для каждой СУБД:

```
<property name="string.type" value="varchar" dbms="postgresql"/>
<property name="string.type" value="nvarchar" dbms="mssql"/>
<changeSet id="1622118750064-2" author="amplicode">
  <createTable tableName="owners">
    <column autoIncrement="true" name="id" type="INT">
      <constraints nullable="false" primaryKey="true" primaryKeyName="PK_OWNERS"/>
    </column>
    <column name="first_name" type="${string.type}(255)"/>
    <column name="last_name" type="${string.type}(255)"/>
    <column name="address" type="${string.type}(255)"/>
    <column name="city" type="${string.type}(255)"/>
  </createTable>
</changeSet>
```


Следовательно, необходимость в создании отдельных changelogs для разных СУБД отсутствует.

Шаблоны для changeset

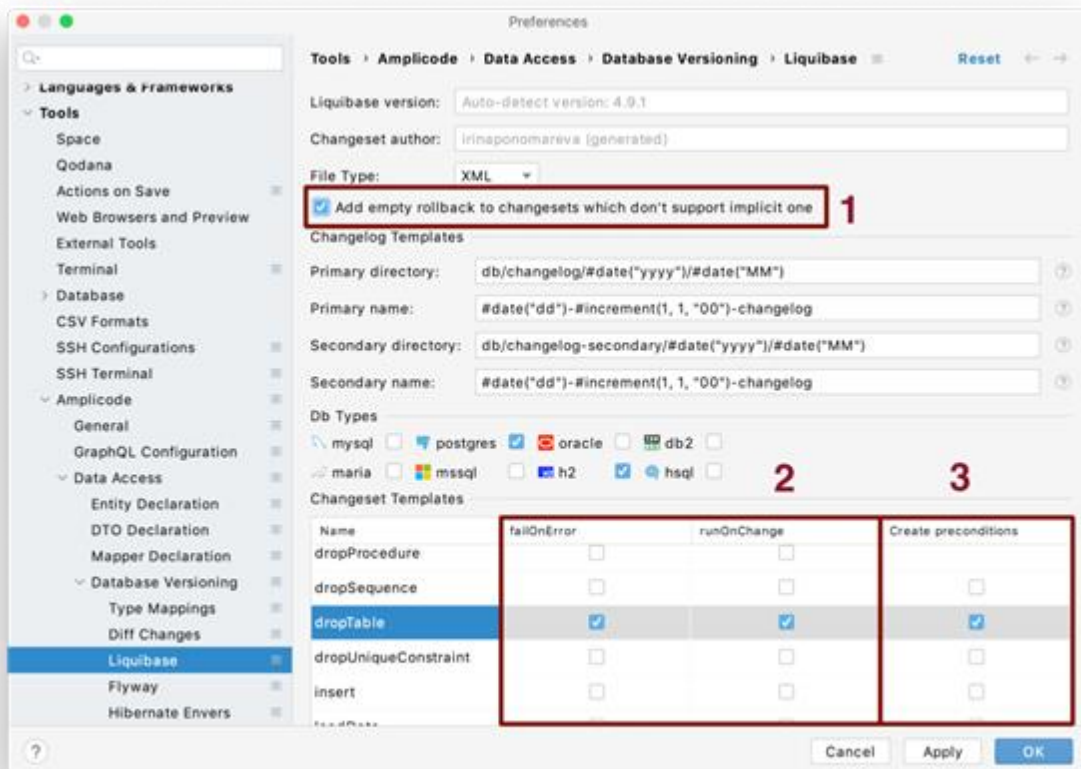
Шаблоны для changeset — это predefined структуры, которые предоставляют стандартизированный формат для задания схем изменений базы данных с использованием Liquibase. Эти шаблоны служат фундаментом для создания единообразных и пригодных к повторному использованию changesets, обеспечивая единство формы и простоту в обслуживании на всех деплоиментах базы данных.

Amplicode дает возможность применять шаблоны при генерации changesets из палитры. Эта функциональность дает возможность включать в них различные кастомизируемые элементы.

- Добавление пустого отката изменений к changesets, которые не поддерживают неявный откат — эта опция автоматически добавляет тег пустого отката с комментарием TODO к любому changeset, в котором отсутствует неявный откат.
- failOnError и runOnChange: Amplicode поддерживает часто используемые атрибуты внутри тега changeSet, позволяя пользователям устанавливать значения по умолчанию для failOnError и runOnChange.
- Создание необходимых условий — каждый changeset может иметь собственные необходимые начальные условия (preconditions). Например, теги tableExists и columnExists будут добавлены к действию addColumn:

```
<changeSet id="1685085536452-1" author="amplicode">
  <preConditions>
    <tableExists tableName="customer"/>
    <not>
      <columnExists tableName="customer" columnName="id"/>
    </not>
  </preConditions>
  <addColumn tableName="customer">
    <column name="id" type="BIGINT">
      <constraints nullable="false" primaryKey="true"
primaryKeyName="pk_customer"/>
    </column>
  </addColumn>
</changeSet>
```

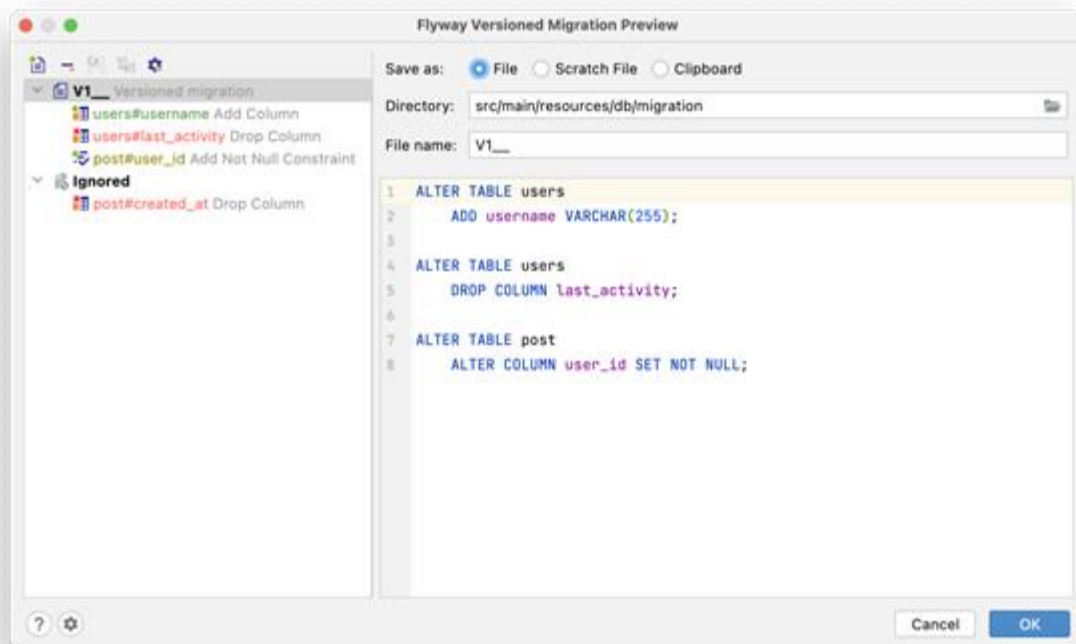
Важно отметить, что некоторые Liquibase changesets могут не предлагать эту опцию. Например, тег procedureExists недоступен для действия createProcedure.



Далее приведен пример использования всех четырех возможностей для changeset удаления таблицы:

```
<changeSet id="1680594632747-1" author="amplicode" runOnChange="true"
failOnError="true">
  <preConditions>
    <tableExists tableName="customer"/>
  </preConditions>
  <dropTable tableName="customer"/>
  <rollback><!--TODO--></rollback>
</changeSet>
```

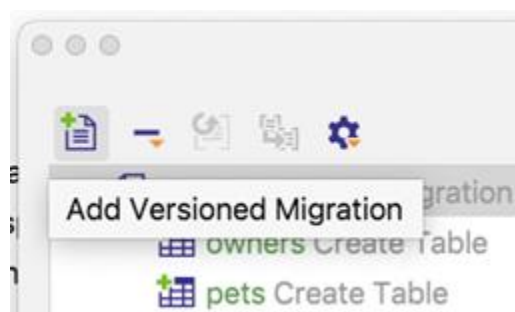
Поддержка Flyway Окно предпросмотра миграции



Amplіcode позволяет выбирать место для хранения сгенерированного скрипта: вы можете выбрать обычный файл или временный (scratch) файл в IDE или в буфере обмена.

Поля Directory и File name отвечают за конфигурацию местоположения сгенерированной миграции. Если миграция с заданным именем уже существует, вы получите предупреждение, после чего изменения будут дописаны в конец к этой миграции.

С левой стороны окна имеется возможность предварительного просмотра реальных изменений, которые будут сгенерированы. Вы можете посмотреть, как будет выглядеть каждое изменение, щелкая на них мышью.



Над списком изменений находится панель с кнопками, включающая следующие действия:

- Add Versioned Migration — создать вторичную версионированную миграцию.

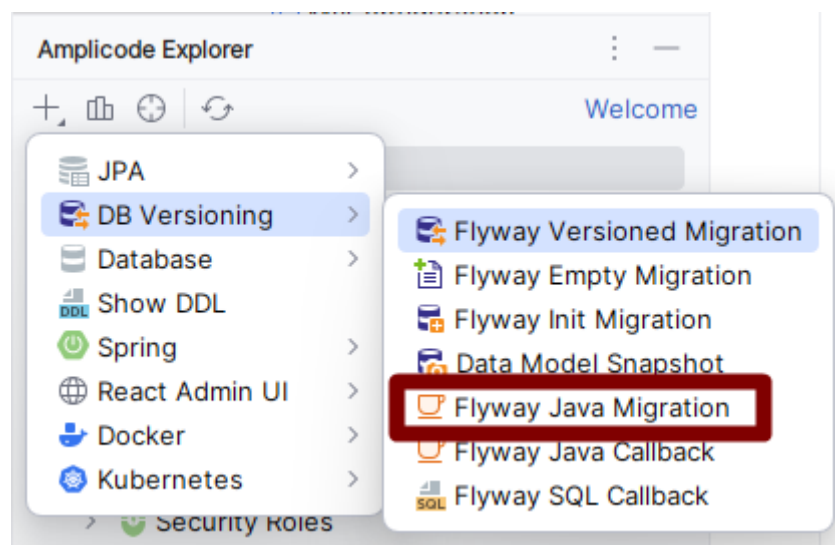
- Remove from Versioned Migration с опциями:
 - Remove from Versioned Migration — просто удалить изменения из текущей миграции.
 - Remove and Ignore — удалить изменения и добавить их в раздел Ignored, чтобы они были исключены также и из будущих миграций.
 - Restore from Ignored — переместить изменения из Ignored в миграцию.
- Move to Another Versioned Migration — по умолчанию создается один миграционный скрипт на каждую генерацию списка различий, куда войдут все изменения. Это действие позволяет переместить изменение в другой миграционный файл.
- Show Other Actions — Эта кнопка поможет вам удобным образом взаимодействовать с большим количеством изменения в миграционных файлах:
 - Select all ... (выбрать все)
 - Expand/collapse all (развернуть/свернуть все)

Чтобы скомбинировать несколько изменений в одном миграционном файле или проигнорировать их, можно перетаскивать их с помощью мыши.

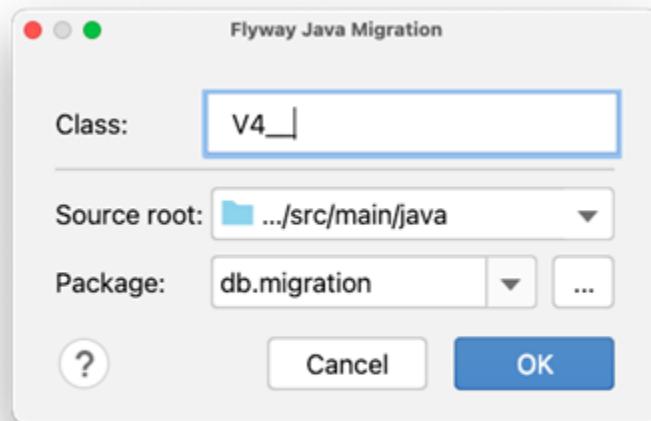
Миграция на Java

Миграции, базирующиеся на Java, хорошо подходят для всех изменений, который можно легко выразить через SQL. Эти миграции представляют классы Java, реализующие интерфейс `JavaMigration` или наследующие от класса `BaseJavaMigration`. Amplicode следует второй опции и генерирует имена классов в соответствии с [соглашением по именованию](#), которое Flyway использует по умолчанию. Это позволяет Flyway автоматически извлекать версию и описание из имени класса.

Чтобы сгенерировать миграцию на Java, нажмите на кнопку Плюс (+) в панели Amplicode Explorer и выберите соответствующий пункт.



В открытом окне вы сможете установить имя класса (Class), корневой каталог для исходного кода (Source root) и имя пакета (Package):

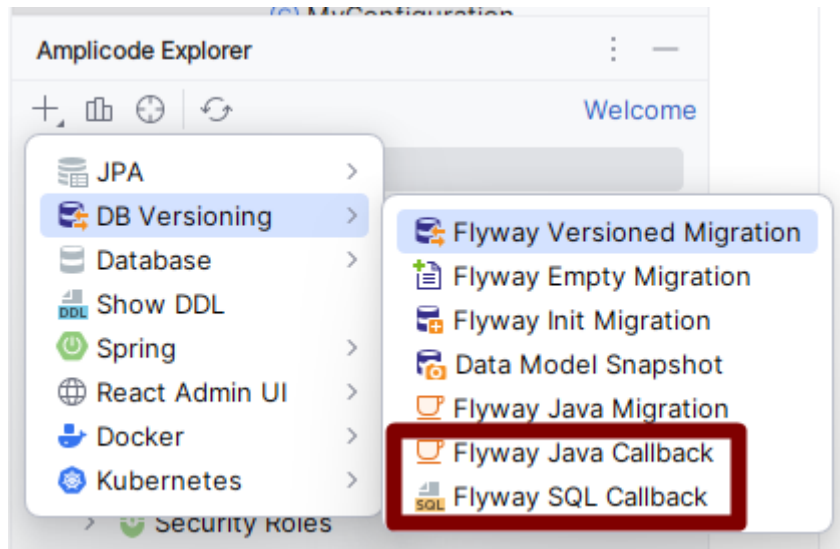


После этого будет сгенерирован следующий Java класс:

```
public class V4__ extends BaseJavaMigration {  
    @Override  
    public void migrate(Context context) {  
        new JdbcTemplate(new SingleConnectionDataSource(context.getConnection(), true))  
            .execute("");  
    }  
}
```

Flyway Callbacks

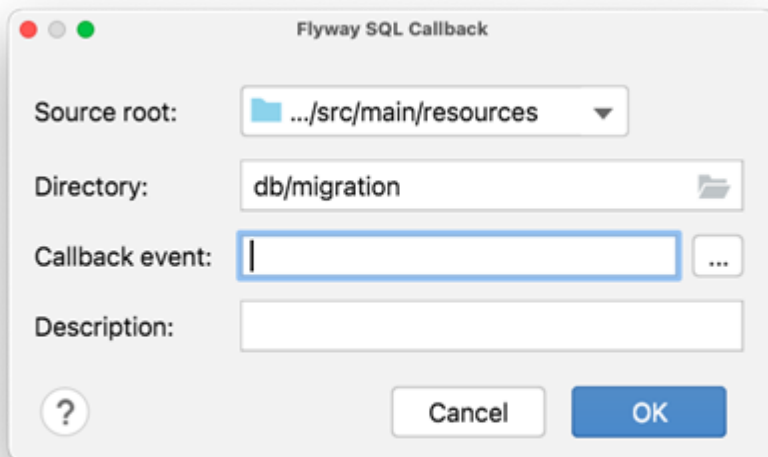
В то время как миграции являются достаточными для большинства наших нужд, определенные ситуации требуют исполнять одно и то же действие снова и снова. С помощью Amplicode вы можете генерировать все события, которые поддерживает Flyway. Чтобы сгенерировать обратные вызовы SQL или Java, нажмите кнопку со знаком «плюс» на вкладке Amplicode Explorer и выберите соответствующий элемент.



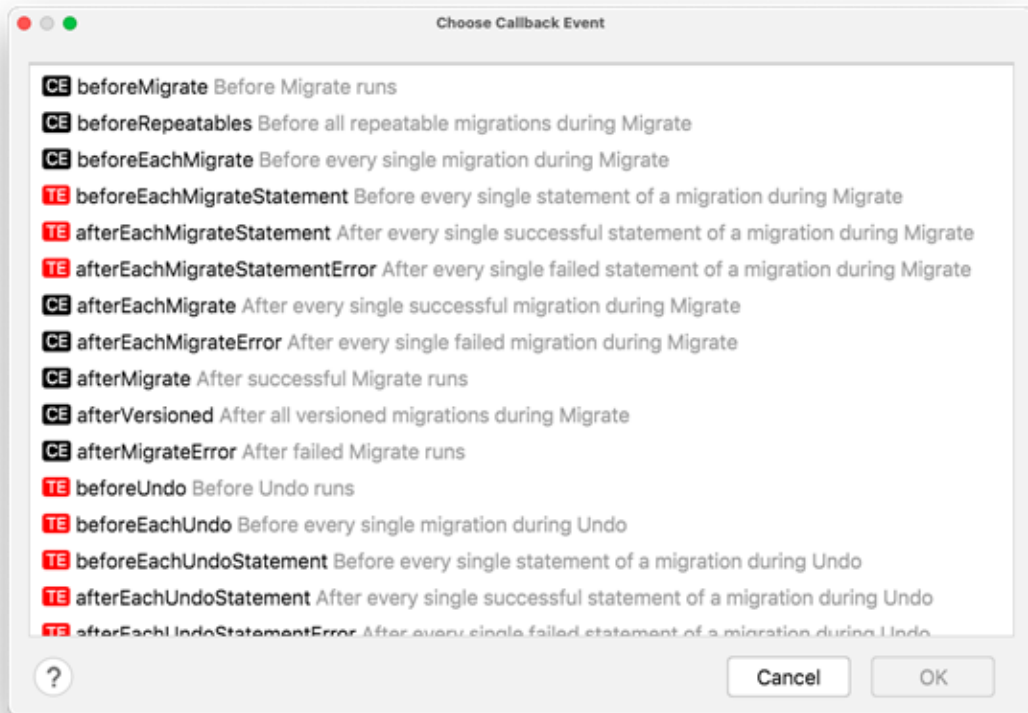
SQL Callbacks

Amplicode предоставляет окно Flyway SQL Callback со следующими полями:

- Поля Source root и Directory отвечают за местоположение сгенерированного файла:



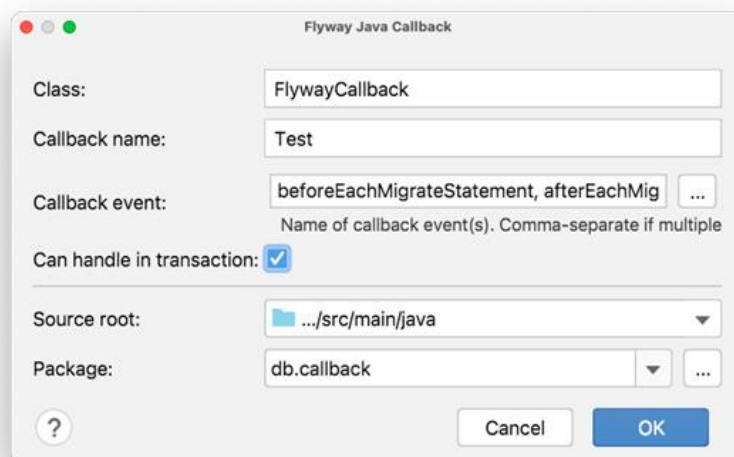
- Поле Callback event позволяет выбирать одно из событий, которые поддерживает Flyway



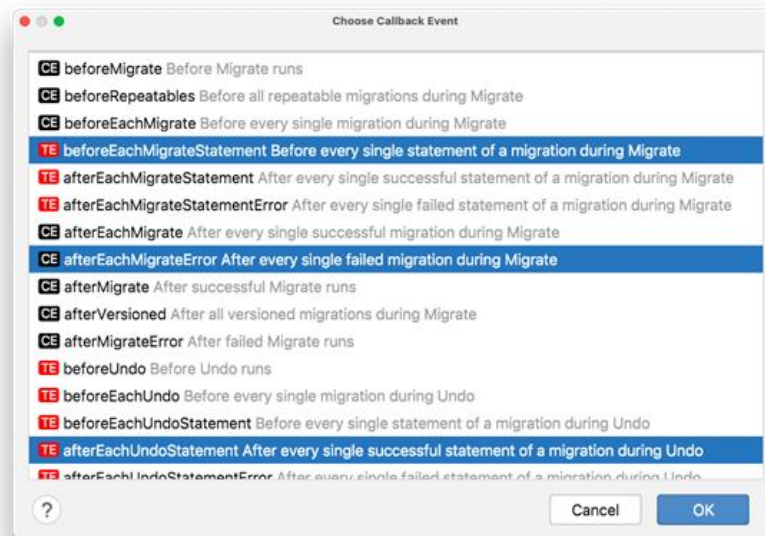
- Опционально, события типа callbacks (обратные вызовы) могут включать описание. Значение поля Description вместе с разделителем будет добавлено к концу к имени callback.

Java Callbacks

Если SQL Callbacks недостаточно гибки для ваших целей, Flyway также поддерживает Java Callbacks. Amplicode предоставляет окно Flyway Java Callback со следующими полями:



- ☞ Поля *Class* и *Class name* отвечают за конфигурацию соответствующих значений для генерируемого Java класса.
- ☞ Поле *Callback event* позволяет выбирать из нескольких событий, которые поддерживает Flyway:



- ☞ Чекбокс *Can handle in transaction* определяет, какое значение вернет переопределенный метод `canHandleInTransaction()` из интерфейса `Callback`, `true` или `false`.
- ☞ Поля *Source root* и *Directory* отвечают за местоположение генерируемого файла.

```
public class TestCallback implements Callback {

    @Override

    public boolean supports(Event event, Context context) {

        return event.equals(Event.*BEFORE_EACH_MIGRATE_STATEMENT*) ||

        event.equals(Event.*AFTER_EACH_MIGRATE_ERROR*) ||

        event.equals(Event.*AFTER_EACH_UNDO_STATEMENT*);

    }

    @Override

    public boolean canHandleInTransaction(Event event, Context context) {

        return true;

    }

}
```



```

@Override

public void handle(Event event, Context context) {

    /**TODO handle logic...*/

}

public String getCallbackName() {

    return "Test";

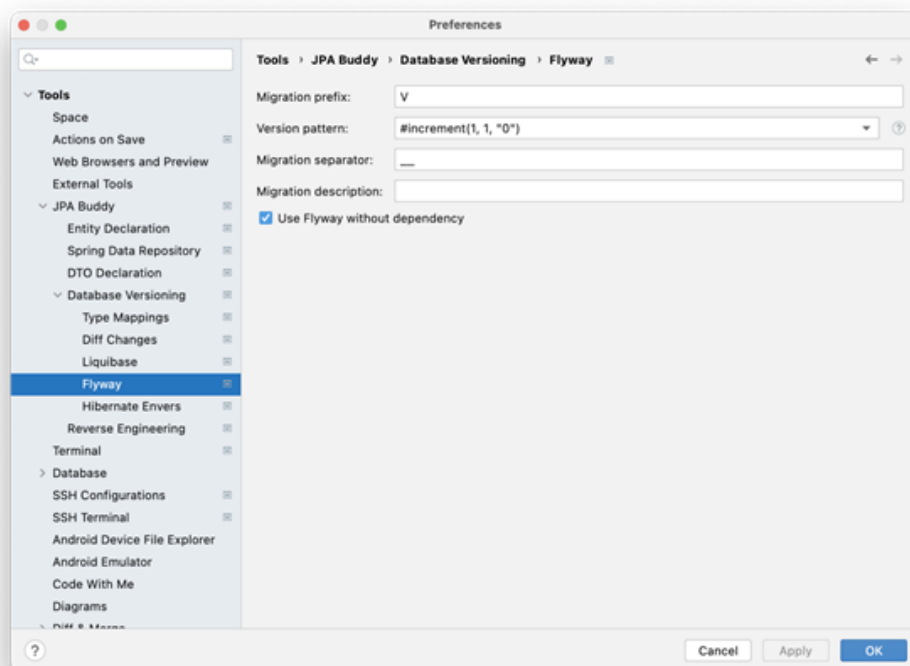
}

}

```

Настройка Flyway

Всякий раз, когда создается пустая или дифференциальная миграция Flyway, Amplicode генерирует имя файла на базе [шаблона именования](#) Flyway. В настройках плагина вы можете сгенерировать следующие значения для генерации имени:



- € **Migration prefix** (префикс миграции). Значение по умолчанию равно V.
- € **Version pattern** (шаблон версии). Пример генерируемой последовательности представлен после дефиса.

- € **Migration separator** (разделитель миграции). Значением по умолчанию является "_".
- € **Migration description** (описание миграции).

Следующие переменные и макросы доступны для шаблонов:

- € `#date([format])` — текущая системная дата, заданная в формате [SimpleDateFormat](#). Например, `#date("\yyyy-MM-dd")` возвращает дату в формате 2020-12-31.
- € `#increment([start], [step], [decimalFormat])` — число, которое используется для сохранения уникальности имени. Значение переменной `start` используется для первого файла и увеличивается на значение шага (`step`) для каждого файла. Параметр `decimalFormat` задает [десятичный формат](#) числа. Например, `#increment(1.0, 0.1, "\#.0")` возвращает значение в формате 1.1, 1.2, 1.3 и т.д.
- € `semVer` — семантическая версия проекта (она же `SemVer`), широко распространенная схема формирования номеров версий, использующая три цифры (Major.Minor.Patch), опциональный тег пре-релиза и опциональный мета тег сборки. Объект содержит следующие методы полного номера версии в примере равен 1.2.3-SNAPSHOT+meta):
 - `${semVer.getRawVersion()}: 1.2.3-SNAPSHOT`
 - `${semVer.getMajor()}: 1`
 - `${semVer.getMinor()}: 2`
 - `${semVer.getPatch()}: 3`
 - `${semVer.getPreRelease()}: SNAPSHOT`
 - `${semVer.getMeta()}: meta`

Преобразование нестандартных типов

Общего метода автоматически преобразовывать нестандартные типы Java на типы SQL/Liquibase не существует. Поэтому вам придется вручную задавать целевой тип для таких атрибутов. Если такие атрибуты существуют в вашем проекте, то после генерации миграционного скрипта Amplicode отобразит предупреждение.

Вы можете сохранять и редактировать конфигурации для преобразования в любое время, используя следующий пункт меню: **Settings** → **Tools** → **Amplicode** → **Data Access** → **Database Versioning** → **Type Mappings**.

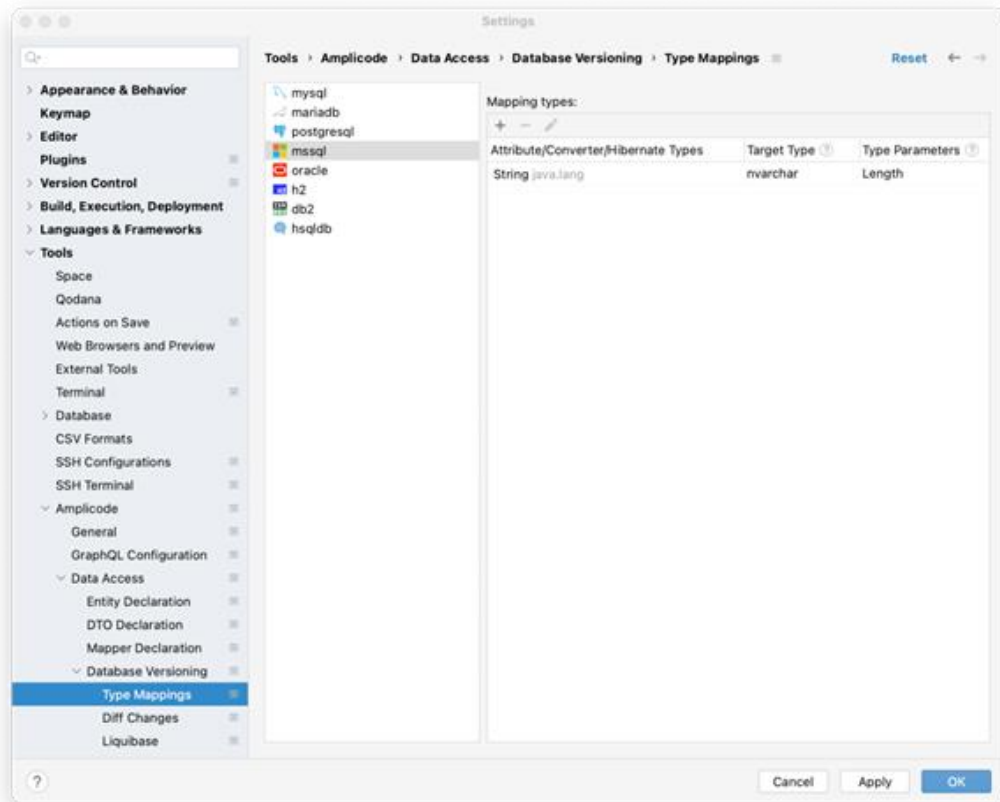
Пункт меню **Settings** находится в меню **File**, если вы работаете под **Windows** или **Linux**, и в меню **IntelliJ IDEA**, если вы работаете под **MacOS**.

Эта функциональность может помочь также в тех случаях, когда приложение работает с базами данных от различных поставщиков. В этом случае ваша схема может иметь несколько различные типы данных для каждой из них.

Предположим, что приложение должно поддерживать как PostgreSQL, так и MSSQL. И вы хотите хранить символы Unicode в своих строковых данных. PostgreSQL

поддерживает символы Unicode в типе VARCHAR, но в MSSQL для этой цели есть отдельный тип данных NVARCHAR.

Amplicode позволяет вам задавать мапирование типов для каждой СУБД. Есть также возможность задавать мапирование для JPA конвертеров и для типов Hibernate:



Конвертеры

Чтобы упростить преобразование типов, Amplicode вводит БД-независимые SQL типы, которые преобразуются в типы, специфичные для каждой БД. Например, varchar преобразуется в varchar2 для Oracle DB и остается varchar для PostgreSQL.

Каждый БД-независимый класс имеет набор алиасов (например, java.sql.Types.VARCHAR или *character varying*), которые в большинстве случаев взаимозаменяемы. Неизвестные типы используются как есть без каких-либо преобразований. Вы можете посмотреть полный список алиасов в приведенной ниже таблице:

Имя	Алиасы	Класс Liquibase
-----	--------	-----------------

bigint	java.sql.Types.BIGINT java.math.BigInteger java.lang.Long integer8 bigserial serial8 int8	liquibase.datatype.core.BigIntType
blob	longblob longvarbinary java.sql.Types.BLOB java.sql.Types.LONGBLOB java.sql.Types.LONGVARBINARY java.sql.Types.VARBINARY java.sql.Types.BINARY varbinary binary image tinyblob mediumblob	liquibase.datatype.core.BlobType
boolean	java.sql.Types.BOOLEAN java.lang.Boolean bit bool	liquibase.datatype.core.BooleanType
char	java.sql.Types.CHAR bpchar	liquibase.datatype.core.CharType

clob	longvarchar text longtext java.sql.Types.LONGVARCHAR java.sql.Types.CLOB nclob longnvarchar ntext java.sql.Types.LONGNVARCHAR java.sql.Types.NCLOB tinytext mediumtext	liquibase.datatype.core.ClobType
currency	money smallmoney	liquibase.datatype.core.CurrencyType
function	liquibase.statement.DatabaseFunction	liquibase.datatype.core.DatabaseFunctionType
datetime	java.sql.Types.DATETIME java.util.Date smalldatetime datetime2	liquibase.datatype.core.LiquibaseDataType
date	java.sql.Types.DATE java.sql.Date	liquibase.datatype.core.DateType
decimal	java.sql.Types.DECIMAL java.math.BigDecimal	liquibase.datatype.core.DecimalType

double	java.sql.Types.DOUBLE java.lang.Double	liquibase.datatype.core.DoubleType
float	java.sql.Types.FLOAT java.lang.Float real java.sql.Types.REAL	liquibase.datatype.core.FloatType
int	integer java.sql.Types.INTEGER java.lang.Integer serial int4 serial4	liquibase.datatype.core.IntType
mediumint		liquibase.datatype.core.MediumIntType
nchar	java.sql.Types.NCHAR nchar2	liquibase.datatype.core.NCharType
number	numeric java.sql.Types.NUMERIC	liquibase.datatype.core.NumberType
nvarchar	java.sql.Types.NVARCHAR nvarchar2 national	liquibase.datatype.core.NVarcharType
smallint	java.sql.Types.SMALLINT int2	liquibase.datatype.core.SmallIntType

timestamp	java.sql.Types.TIMESTAMP java.sql.Types.TIMESTAMP_WITH_TIMEZONE java.sql.Timestamp timestampz	liquibase.datatype.core.TimestampType
time	java.sql.Types.TIME java.sql.Time timetz	liquibase.datatype.core.TimeType
tinyint	java.sql.Types.TINYINT	liquibase.datatype.core.TinyIntType
uuid	uniqueidentifier java.util.UUID	liquibase.datatype.core.UnknownType
varchar	java.sql.Types.VARCHAR java.lang.String varchar2 character varying	liquibase.datatype.core.VarcharType
xml	xmltype java.sql.Types.SQLXML	liquibase.datatype.core.XMLType

Аннотация @JavaType

Ниже приведен пример того, как вы можете конфигурировать мапирование в Amplicode, чтобы воспользоваться аннотацией @JavaType из Hibernate 6:

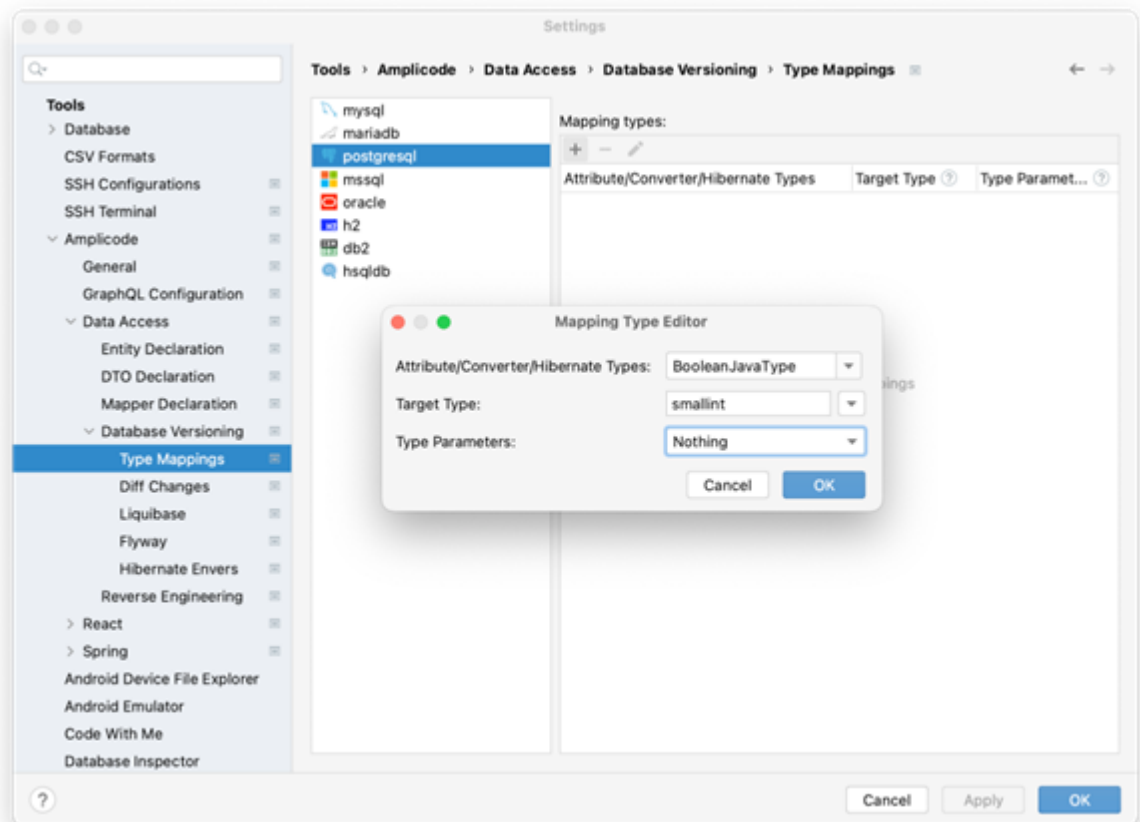
✉ Добавьте аннотацию @JavaType к подходящему атрибуту и задайте требуемую реализацию:

```
@JavaType(BooleanJavaType.class)
```

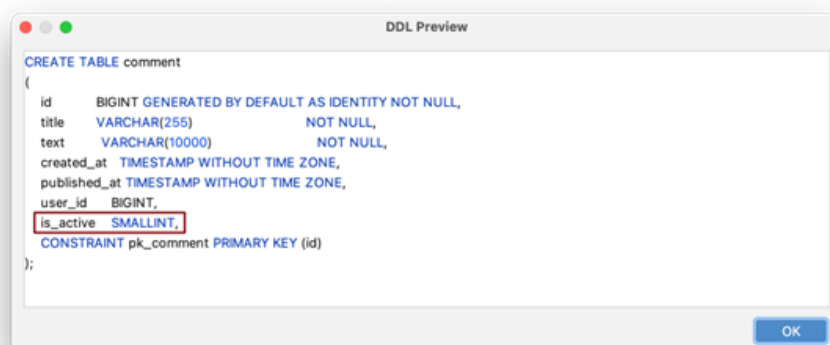
```
@Column(name = "is_active")
```

private Boolean isActive;

✉ Сконфигурируйте мапирование типов:



В процессе генерации скриптов Amplicode найдет аннотацию @JavaType и заменит колонку типа BOOLEAN на тип SMALLINT.



В дополнение к вышесказанному Hibernate предоставляет аннотацию @JavaTypeRegistration. Вы можете использовать ее, чтобы задать все правила преобразования в одном месте, и Amplicode подхватит также и их.


```
@JavaTypeRegistration(javaType = Boolean.class, descriptorClass =  
BooleanJavaType.class)
```

Стратегия именования и настройки по максимальному идентификатору

Поскольку Amplicode поддерживает одновременно шесть баз данных, важно иметь возможность конфигурировать стратегии именования и максимальную длину идентификатора.

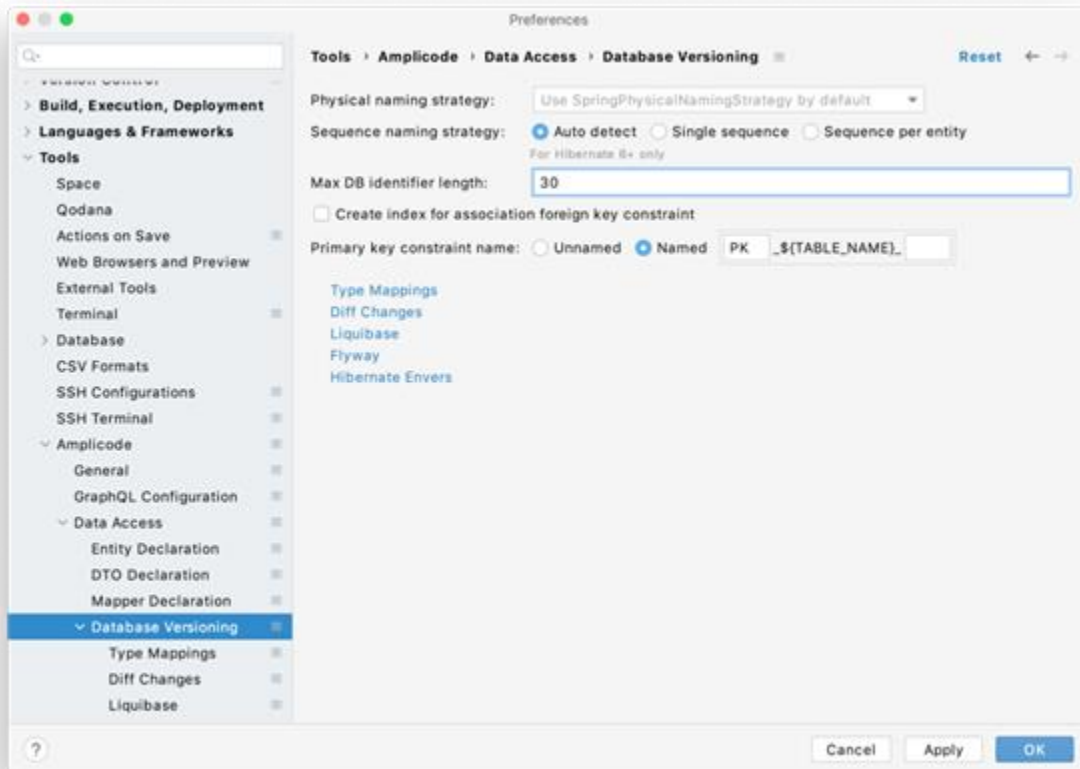
По умолчанию Spring Boot конфигурирует физическую стратегию именования с использованием SpringPhysicalNamingStrategy. Эта реализация генерирует все имена таблиц в нижнем регистре, разделяя слова символами подчеркивания. Например, сущность TelephoneNumber преобразуется на таблицу telephone_number, даже если вы аннотируете сущность как @Table(name = "TelephoneNumber"). Те же имена должны использоваться в миграционных скриптах, поэтому Amplicode применяет физическую стратегию именования ко всем именам во время генерации скрипта.

Поддерживаются следующие стратегии:

- ✘ SpringPhysicalNamingStrategy – опция по умолчанию
- ✘ PhysicalNamingStrategyStandardImpl
- ✘ CamelCaseToUnderscoresNamingStrategy (только для проектов с Hibernate 6 и более поздними версиями).

Максимальный идентификатор

Реляционные СУБД имеют свои ограничения. Например, имена таблиц для OracleDatabase всех версий старше 12.1 ограничены до 30 байт. Чтобы избежать проблем со скриптами версионирования, вы можете ограничить имена таблиц по длине, как показано на рисунке:

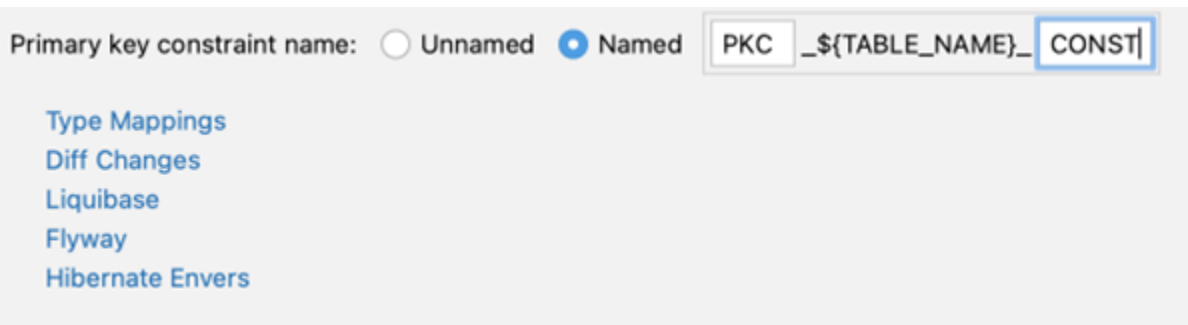


Кроме того, вы можете задать:

- ☒ Создавать ли индекс для ограничения по внешнему ключу на ассоциации или нет (задается пометкой или снятием пометки с соответствующего чекбокса)
- ☒ Имя ограничения по первичному ключу (процедура описана ниже).

Ограничение может иметь или не иметь имени. Если мы помечаем радио баттон *Named*, префиксом по умолчанию для всех ограничений по первичному ключу будет **PK** (сокращение от *primary key*). Мы можем поменять эту настройку, введя другую комбинацию символов в поле, предшествующее переменной $\$(TABLE_NAME)$. Также существует возможность добавить постфикс, введя его в следующее поле после $\$(TABLE_NAME)$.

Например, если мы сохраним настройки по умолчанию, ограничение по первичному ключу для таблицы User будет pk_user. Но если мы поменяем настройки, как показано на картинке внизу, имя ограничения поменяется на pkc_user_const.



Reverse Engineering

Введение

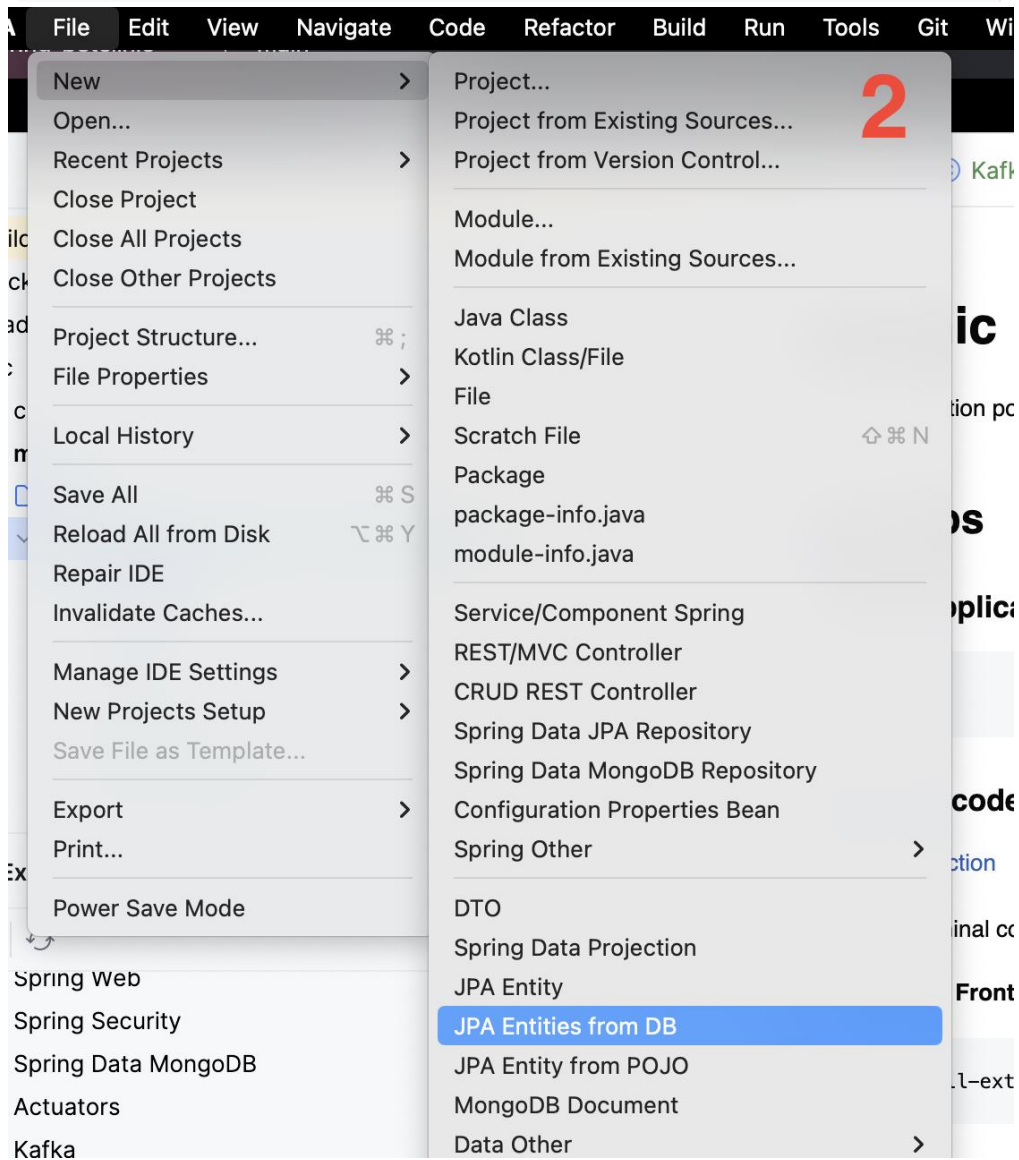
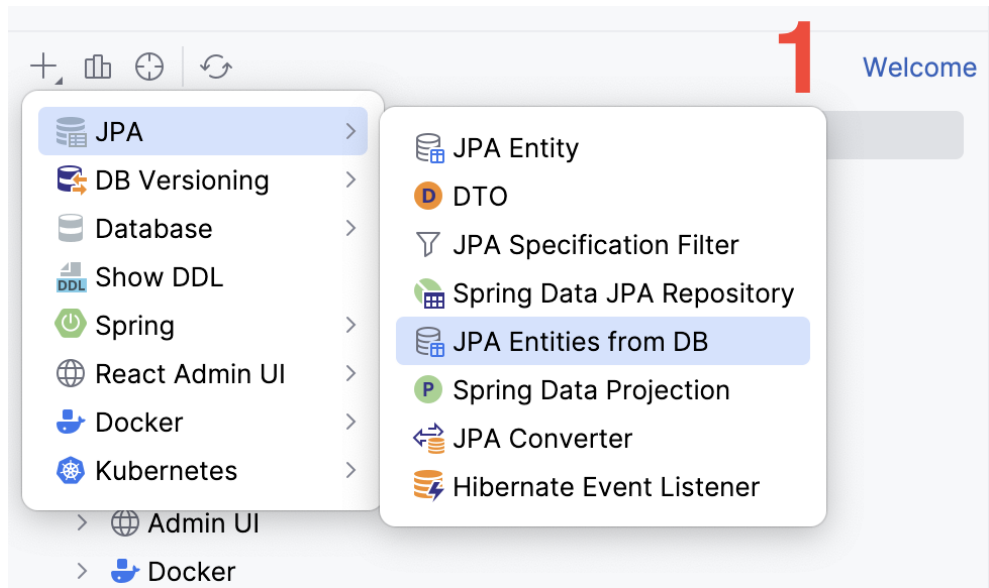
Amplicode позволяет вам выборочно отметить таблицы или представления (views), а также поля в вашей реляционной базе данных и затем получить их как JPA сущности.

Примечание:

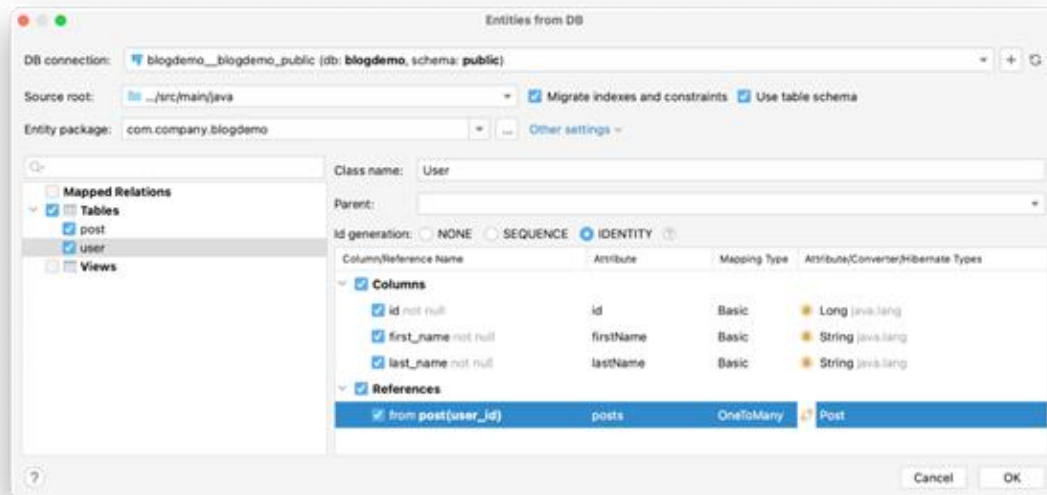
Первое, что необходимо сделать, чтобы использовать функциональность reverse engineering — это создать подключение к базе данных. Правильный способ создавать подключения и связанные с этим возможные проблемы описаны в отдельной главе в документации. [Обратитесь к ней, чтобы узнать больше](#)

В *IntelliJ IDEA Community Edition* вы можете генерировать сущности из БД с помощью:

- ☞ Панели Amplicode Explorer (1), используя кнопку **Плюс**
- ☞ Панели Project (2), используя правую кнопку мыши.



Визард Entities from DB



Конфигурация

В верхней части окна вы видите поля формы, которые позволяют конфигурировать:

- € **DB connection** (подключение к БД)
- € Кэш схемы БД
- € **Source root** (корневой каталог исходника) и **Entity package** (пакет сущности), где будут сохраняться сгенерированные сущности
- € Надо ли мигрировать ограничения и индексы
- € Надо ли задавать имя схемы в аннотации `@Table`.

Также, через выпадающий список **Other settings** вы можете попасть в декларацию сущности и в настройки reverse engineering.

Мапированные отношения, таблицы и представления

В левой части окна вы видите:

- € **Mapped Relations** (мапированные отношения) — таблицы и представления, мапированные на JPA сущности
- € **Tables** — таблицы, которые существуют в базе данных, но не мапируются на сущности
- € **Views** — представления, которые существуют в базе данных, но не мапируются на сущности.

После выбора любого элемента из дерева появится панель для миграции атрибутов из колонок. Кроме того, вы сможете задать имя класса в поле **Class name**.

Миграция атрибутов

Основная часть окна позволяет конфигурировать все, что относится к атрибутам. Вы можете выбирать, какие атрибуты вы хотите добавить, и поменять все их параметры кроме *Column Name*. Типы для преобразования и типы атрибутов/конвертеров/Hibernate представлены как выпадающие списки.

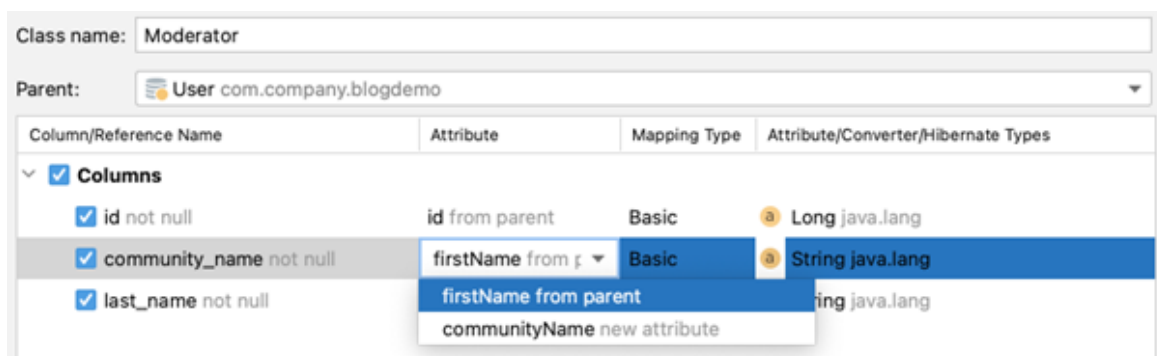
Все атрибуты делятся на три категории:

- ∉ **Migrated Columns** — колонки, которые уже присутствуют в сущности (доступно только для преобразованных отношений)
- ∉ **Columns** — новые колонки, которые еще не преобразованы в сущность или родительский @MappedSuperclass
- ∉ **References** — опциональные ассоциации, которые не представлены в виде колонок в выбранной таблице.

Родительские сущности

Amplicode предлагает возможность определять родительскую сущность посредством выбора класса, аннотированного @MappedSuperclass, из выпадающего списка *Parent*. Это позволяет сгенерированным сущностям расширять родительский класс и автоматически наследовать все атрибуты, имеющие то же имя и тип.

В тех случаях, когда имя колонки в @MappedSuperclass не соответствует таблице дочерней сущности, мы все еще можем унаследовать атрибут, используя аннотацию @AttributeOverride. Мы просто выбираем имя атрибута и атрибут для перезаписи, а Amplicode помогает нам в управлении наследованием.



Во время генерации сущности Amplicode предупреждает нас, если какие-либо унаследованные атрибуты из @MappedSuperclass отсутствуют в базе данных. Чтобы привести модель в соответствие с базой данных, воспользуйтесь действием *Generate DDL by Entities* в меню Amplicode Explorer и выберите опцию *Existing DB update*.

Создание Enums

Для атрибутов, соответствующих типам String или Integer вы можете сменить тип преобразования с Basic на Enum, и Amplicode создаст соответствующий enum в проекте. Вам придется заполнить enum нужными значениями вручную.

Работа с неизвестными типами

Для некоторых SQL типов точное соответствие в Java классах отсутствует. В нашем случае Amplicode не устанавливает тип для предотвращения генерации неработающего кода. Вам понадобится выбрать тип атрибута самостоятельно. Вы также можете сконфигурировать типа для преобразования по умолчанию для каждой СУБД в настройках.

Если у вас есть библиотека [Hypersistence Utils](#) (прежде известная как Hibernate Types) в списке зависимостей проекта, Amplicode может автоматически предложить подходящие типы из библиотеки для неподдерживаемых SQL типов во время процедуры reverse engineering.

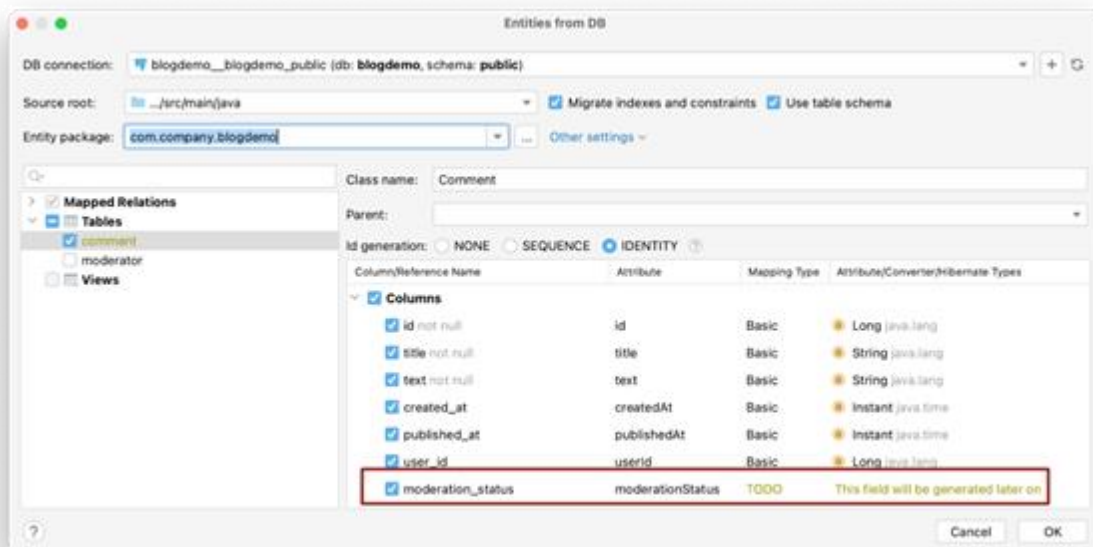
За инструкциями по добавлению этой библиотеки в ваш проект вы можете обратиться к [руководству по инсталляции, доступному на GitHub странице проекта](#).

Также, библиотеку можно добавить через *Amplicode Explorer* → *Create* → *Spring* → *Configurations* → *Hypersistence Utils (Hibernate Types)*

После подключения библиотеки выберите пункт *Columns* в меню *Palette* → *Reverse Engineering*, и предложения подходящих типов будут доступны для вас. Просто нажмите **OK** для подтверждения своего выбора.

Комментарии // TODO

Если вы хотите отложить создание атрибутов для отдельных колонок, вы можете выбрать `//todo comment` в качестве типа преобразования.



Amplicode сгенерирует комментарий `//todo` с соответствующим действием quick-fix (быстрая коррекция), зависящим от типа колонки. Вы можете вызвать эти действия через комбинацию **⌘+B/Ctrl+B**:

- ⊘ Для известных базовых и ассоциативных типов вы можете:
 - Раскомментировать как есть
 - Удалить преобразование колонки
- ⊘ Для неизвестных типов колонки вы можете:
 - Задать целевой Java тип
 - Раскомментировать как есть
 - Удалить преобразование колонки.

Ниже представлен пример сгенерированных комментариев `//todo` для атрибута с неизвестным типом колонки:

```
/*
```

```
TODO [Amplicode] create field to map the 'moderation_status' column
```

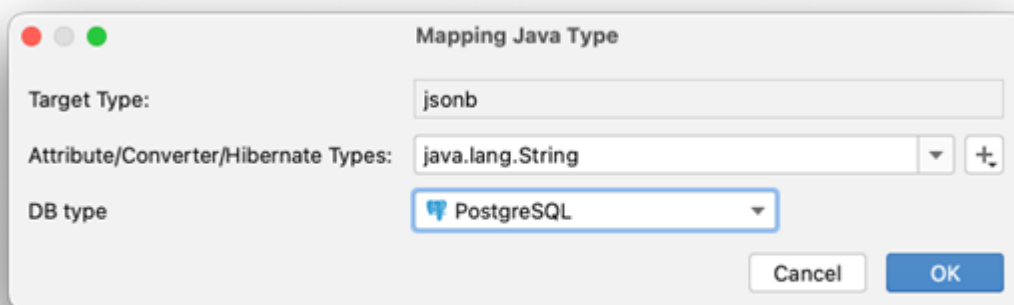
```
Available actions: Define target Java type | Uncomment as is | Remove column mapping
```

```
@Column(name = "moderation_status", columnDefinition = "jsonb(0, 0)")
```

```
private Object moderationStatus;
```

```
*/
```

После вызова действия **Define target Java type** появится следующее всплывающее окно:



Amplicode запомнит преобразование данных, чтобы применить его при последующих вызовах функции `reverse engineering`. Вы всегда можете поменять эти настройки.

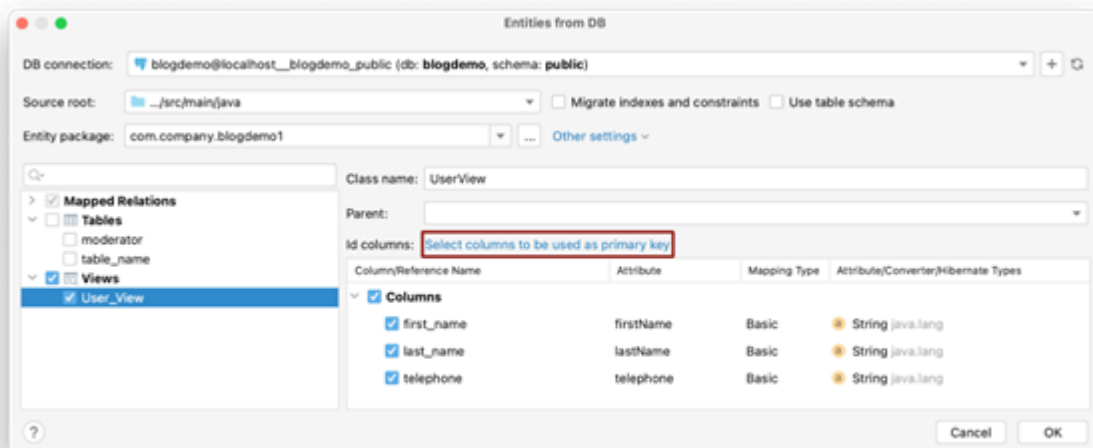
Преобразование представлений БД (Views) на JPA сущности

Amplicode следует всем *best practices*, предоставляя наиболее эффективное мапирование для представлений БД при использовании `reverse engineering`:

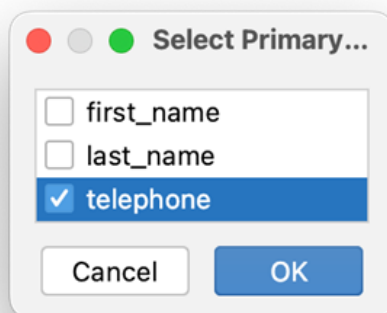
1. Поскольку представления не имеют первичного ключа, Amplicode позволяет выбрать поле или набор полей для использования в качестве идентификатора для целевой сущности.
2. Большинство представлений БД немутабельны. Поэтому Amplicode добавляет аннотацию `@Immutable` к сущности и генерирует только геттеры. Это помогает улучшить производительность приложения
3. Amplicode генерирует только `no-arg protected` конструктор для сущностей, завязанных на БД представление, в соответствии с JPA спецификациями, что не позволяет разработчикам создавать новые экземпляры таких сущностей в коде реализации бизнес-логики.

Чтобы создать JPA сущность из представления:

- ☞ Нажмите правую кнопку мыши на вкладке **Database** и выберите **JPA Entities from DB**
- ☞ Щелкните мышью на ссылке **Select columns to be used as primary key**



☞ Выберите колонку (или несколько), наиболее подходящую для использования в качестве первичного ключа JPA сущности.



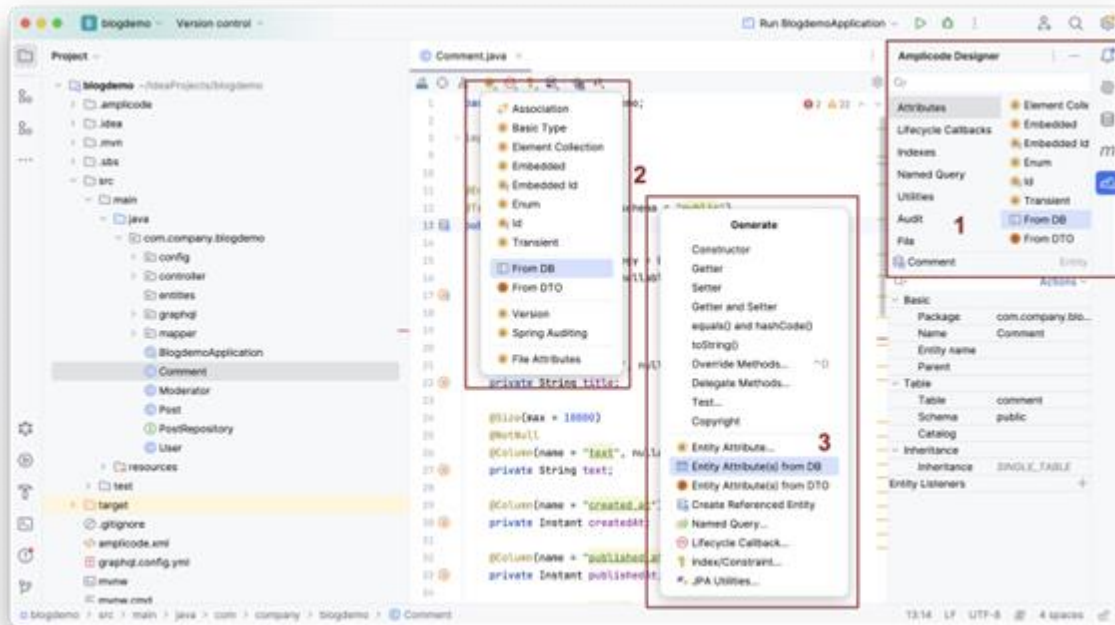
☞ Нажмите

OK

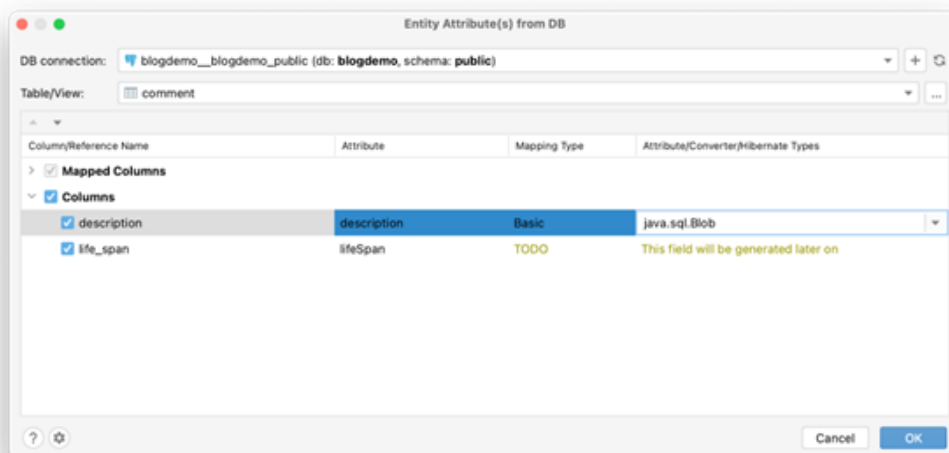
дважды.

Колонки для функции reverse engineering

Некоторые разработчики предпочитают подход к разработке, начинающийся с базы данных (DB-first). Прежде всего, они добавляют колонки напрямую к базе данных и только затем обновляют JPA модель. Amplicode может автоматизировать этот процесс. Чтобы добавить атрибуты к существующей сущности, выберите действие **From DB** в **Amplicode Designer** (1), **Editor Toolbar** (2) или из меню **Generate** (3) от IntelliJ IDEA:



После этого появится визард *Entity attribute(s) from DB*:



Процесс миграции атрибутов идентичен тому, что описано в разделе “Миграция атрибутов”.

Умный поиск отношений

Amplcode глубоко понимает вашу модель. В некоторых случаях она может правильно выбрать кардинальность: `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`. Самое впечатляющее здесь то, что Amplcode может показать отношения даже тогда, когда в текущей таблице отсутствуют соответствующие колонки.

Давайте посмотрим более пристально на каждый из этих случаев.

@OneToOne (один к одному)

Существует две ситуации, в которых мы можем уверенно полагать, что кардинальность отношения будет @OneToOne:

1. Таблица содержит колонку с уникальным ограничением, которое ссылается на первичный ключ другой таблицы.
2. Первичный ключ таблицы является внешним ключом.

@OneToMany и @ManyToOne (один ко многим и многие к одному)

Если таблица содержит колонку, которая ссылается на первичный ключ другой таблицы, это скорее всего ассоциация @ManyToOne. Но вы также можете поменять кардинальность на @OneToOne, если это необходимо. Таким образом, в зависимости от того, к какой таблице применяется функция reverse engineering, Amplicode определит типа преобразования как @OneToMany или @ManyToOne.

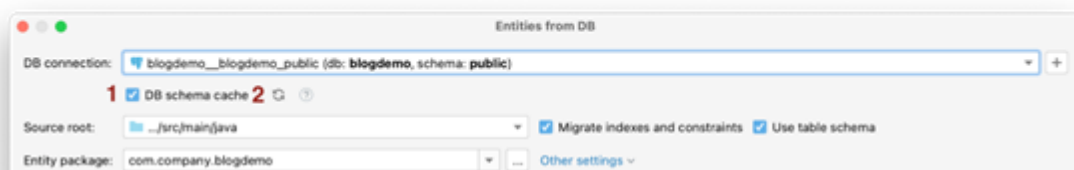
@ManyToMany (многие ко многим)

Чтобы установить отношение вида “многие ко многим” между двумя таблицами, вам необходимо использовать соединительную таблицу. В этом случае соединительная таблица содержит всего две колонки — внешние ключи. Amplicode может автоматически обнаружить такую таблицу и идентифицировать кардинальность отношения между двумя таблицами, чьи первичные ключи представлены как внешние ключи в соединительной таблице, как @ManyToMany.

Если эта ассоциация не существует ни в одной из сущностей, Amplicode сгенерирует ее внутри сущности, для которой была вызвана функция reverse engineering

Работа с удаленной БД

Чем больше база данных и чем медленнее соединение (например, если это удаленная БД), тем дольше будет процесс загрузки схемы базы данных. Для лучшей usability Amplicode предоставляет чекбокс для кэширования схемы БД Amplicode для IntelliJ IDEA Community edition. Когда вы его отмечаете (1), создается файл снимка (snapshot) для выбранной базы данных во временной директории. В противном случае схема БД будет загружаться из базы данных при каждом использовании функции reverse engineering. Когда необходимо, вы можете обновить кэш сохраненной схемы (2).



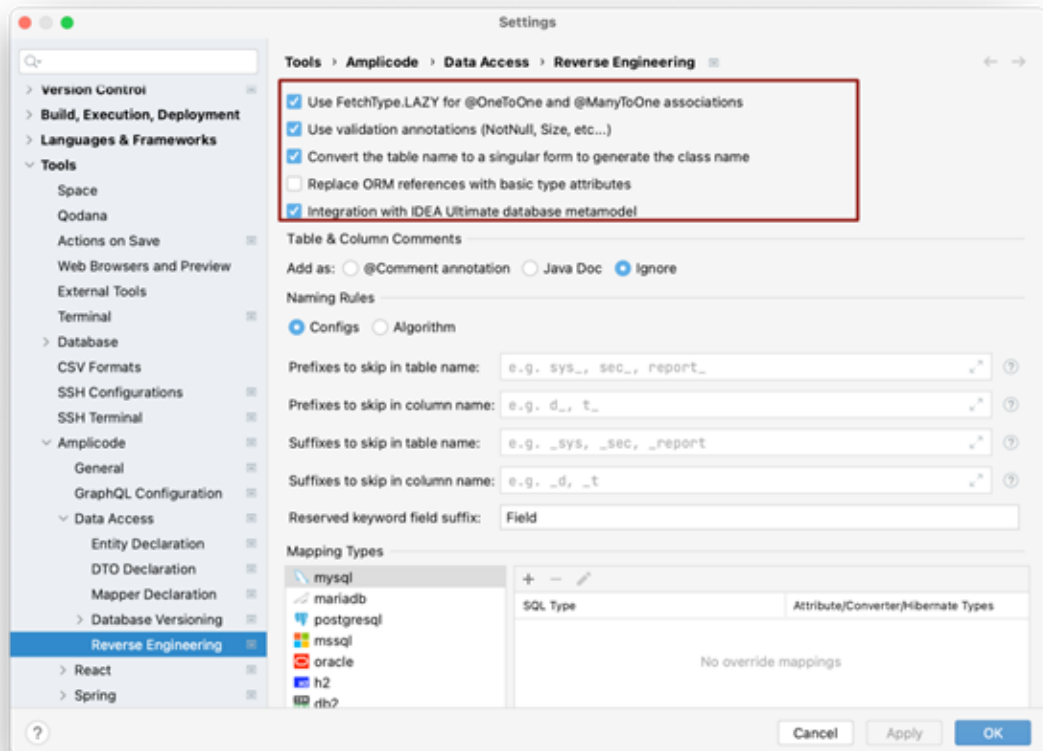
Настройки

Чтобы зайти в настройки функции Reverse Engineering, выберите *Settings* → *Tools* → *Amplicode* → *Data Access* → *Reverse Engineering*.

Общие

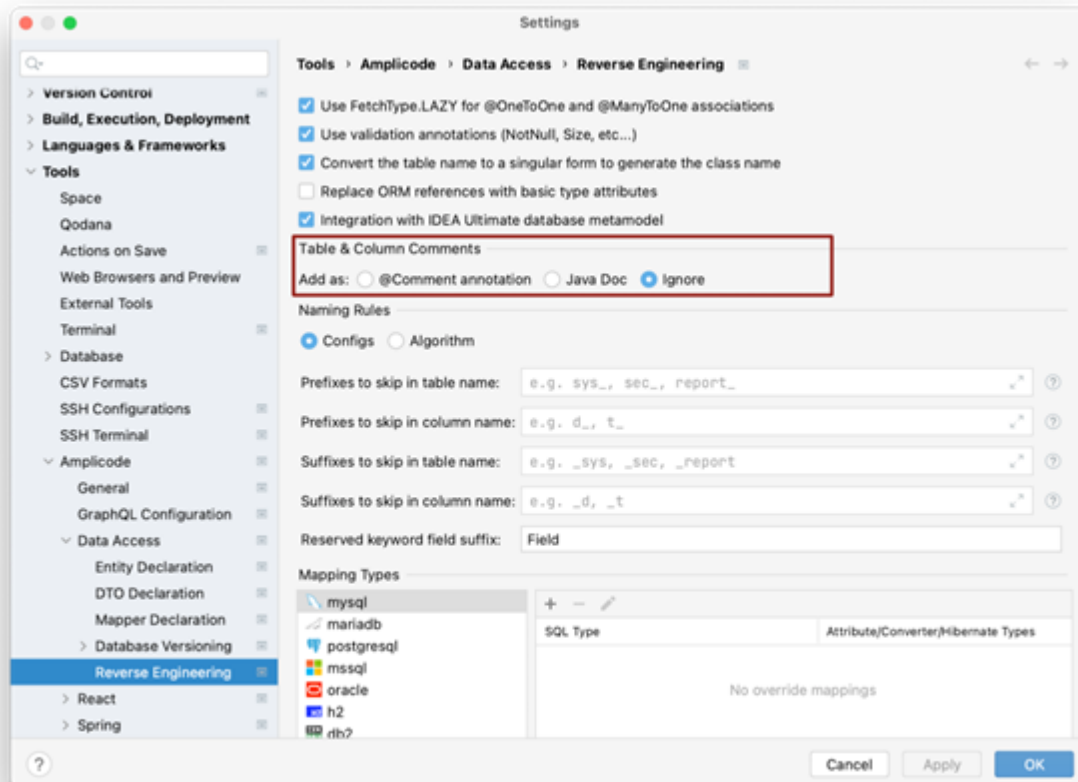
Общие настройки включают в себя следующие параметры:

1. **Fetch Type** — чтобы следовать best practices и избегать потенциальных проблем с производительностью, Amplicode по умолчанию устанавливает FetchType.LAZY для ассоциаций @OneToOne и @ManyToOne.
2. **Validation Annotations** — аннотации валидации дают вам дополнительный слой защиты в добавление к БД ограничениям. По умолчанию Amplicode будет накладывать такие аннотации на атрибуты сущностей во время reverse engineering.
3. **Pluralization** — по-умолчанию Amplicode использует форму единственного числа для имен сущностей. Например, если у вас есть таблица с именем users, Amplicode создаст сущность User. Если вы отключите эту опцию, Amplicode будет сохранять первоначальное имя таблицы и только делать первую букву заглавной – Users.
4. **Basic type attribute** — когда эта опция включена, Amplicode проанализирует ORM ссылки в схеме базы данных и сгенерирует атрибуты базовых типов вместо создания ассоциаций и отношений между сущностями. Это может быть полезно в некоторых сценариях, где вы можете предпочесть простые атрибуты сложным ассоциациям.
5. **IDEA Ultimate integration** — включите эту опцию, если хотите использовать метамодель баз данных, предоставляемую IntelliJ IDEA Ultimate для генерации объектов, относящихся к данным, вместо использования JDBC драйвера для получения мета информации. Это гарантирует нам, что сгенерированные объекты идеально соответствуют структуре базы данных. Заметим, что Amplicode может использовать все настройки подключения, заданные в интерфейсе
IntelliJ IDEA Ultimate.



Комментарии к таблицам и колонкам

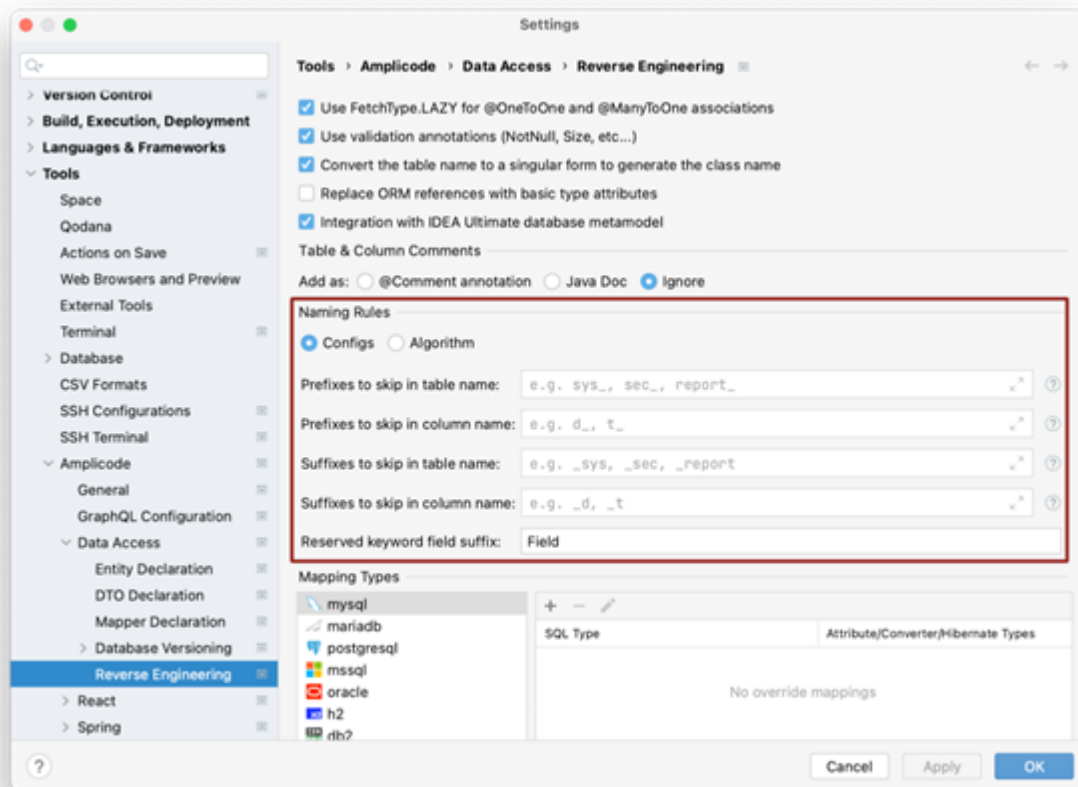
Чтобы сохранить комментарии, добавленные к объектам базы данных, Amplicode переносит их в соответствующую сущность, используя Hibernate аннотацию `@Comment` или `JavaDocs`, в зависимости от ваших настроек.



Пожалуйста обратите внимание, что только аннотации @Comment на сущностях могут включаться в сгенерированные DDL скрипты, в зависимости от настроек, а JavaDocs будут проигнорированы в любом случае.

Правила именования

Через конфиги

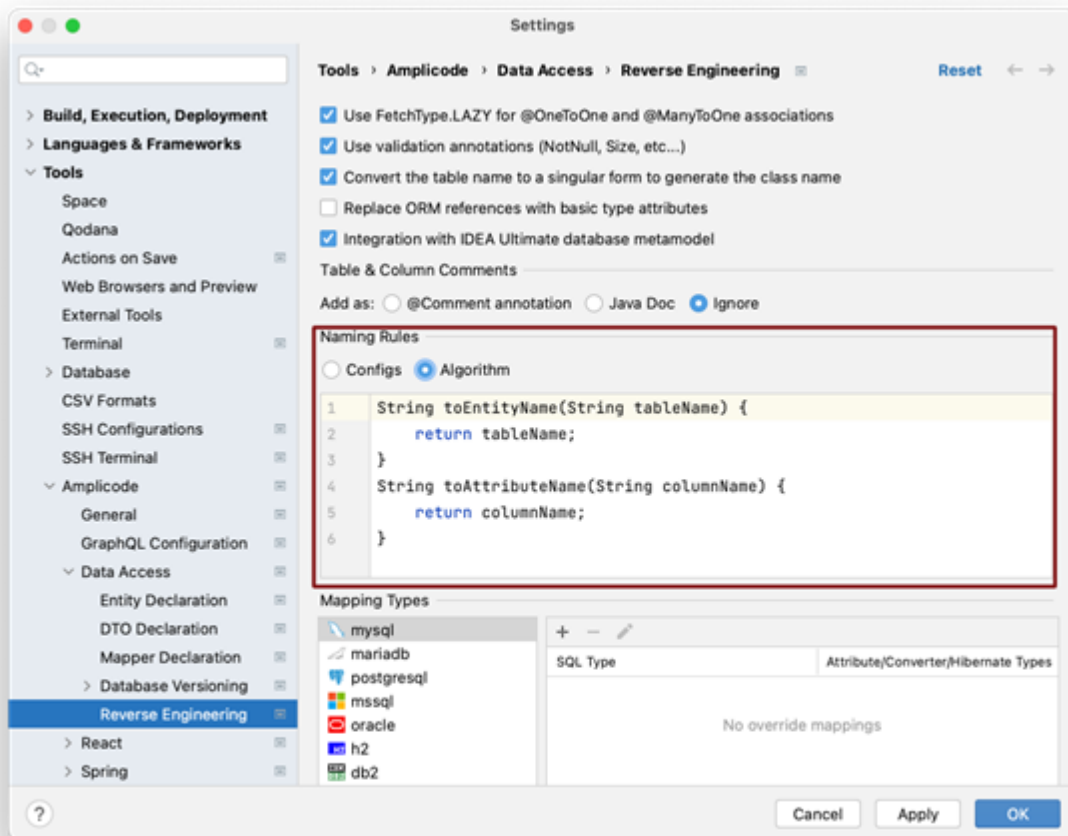


Зачастую DBA специалисты придерживаются определенных договоренностей по именованию объектов баз данных. Например, имена всех таблиц и колонок содержат определенный префикс или суффикс. Однако, Java разработчики обычно предпочитают убирать эти префиксы/суффиксы для JPA модели. Amplicode позволяет вам задавать префиксы и суффиксы, которые надо игнорировать.

Предположим, мы установили `sys_` и `p_` в качестве префиксов, которые следует пропускать. После этого мы применили функцию `reverse engineering` к таблицам `sys_user` и `p_product`. В результате префиксы не появляются в соответствующих именах сущностей. Окончательными именами сущностей будут `User` и `Product`, а не `SysUser` и `PProduct`.

Кроме того, имена колонок баз данных иногда совпадают с зарезервированными словами Java, например, `public`, `interface`, и т.д. В этом случае вы можете сконфигурировать суффикс для поля, и Amplicode будет добавлять его к первоначальному имени колонки. Например, для суффикса `Field` результирующими именами будут `publicField` и `interfaceField`.

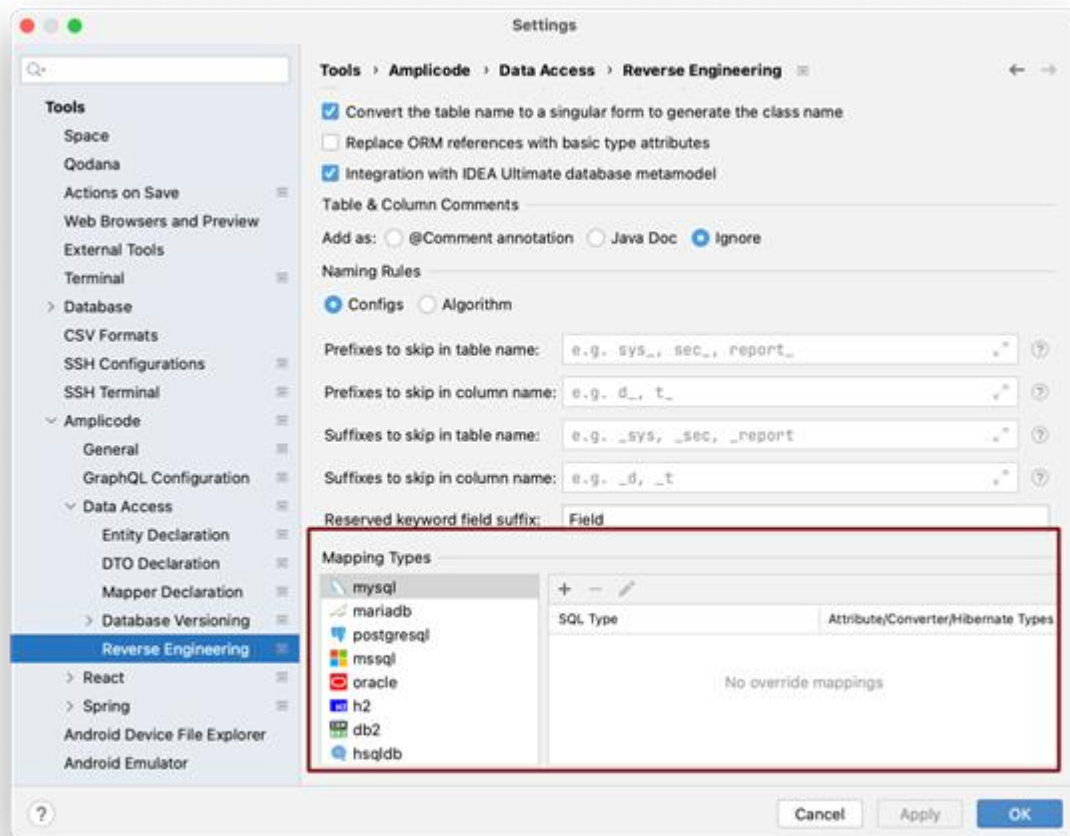
Через алгоритм



Несмотря на гибкие опции для конфигурации префиксов, суффиксов, зарезервированных слов и т.д., в некоторых случаях этого все еще может быть недостаточно. Amplicode не ограничивает вас только этими настройками. Вы можете написать кастомизированный код для обработки имен таблиц и колонок баз данных. Более того, вы можете не только писать код в актуальном редакторе, но и импортировать существующие классы и использовать его методы.

Важно отметить, что Amplicode не отслеживает изменения в классах, используемых в алгоритмах именования, в реальном времени. Поэтому после изменения в классе, используемом внутри алгоритма, либо обновите свои настройки, либо перезапустите среду разработки.

Преобразование типов



Когда приложение работает с несколькими СУБД, ваша схема может содержать немного разные типы данных для каждой из них.

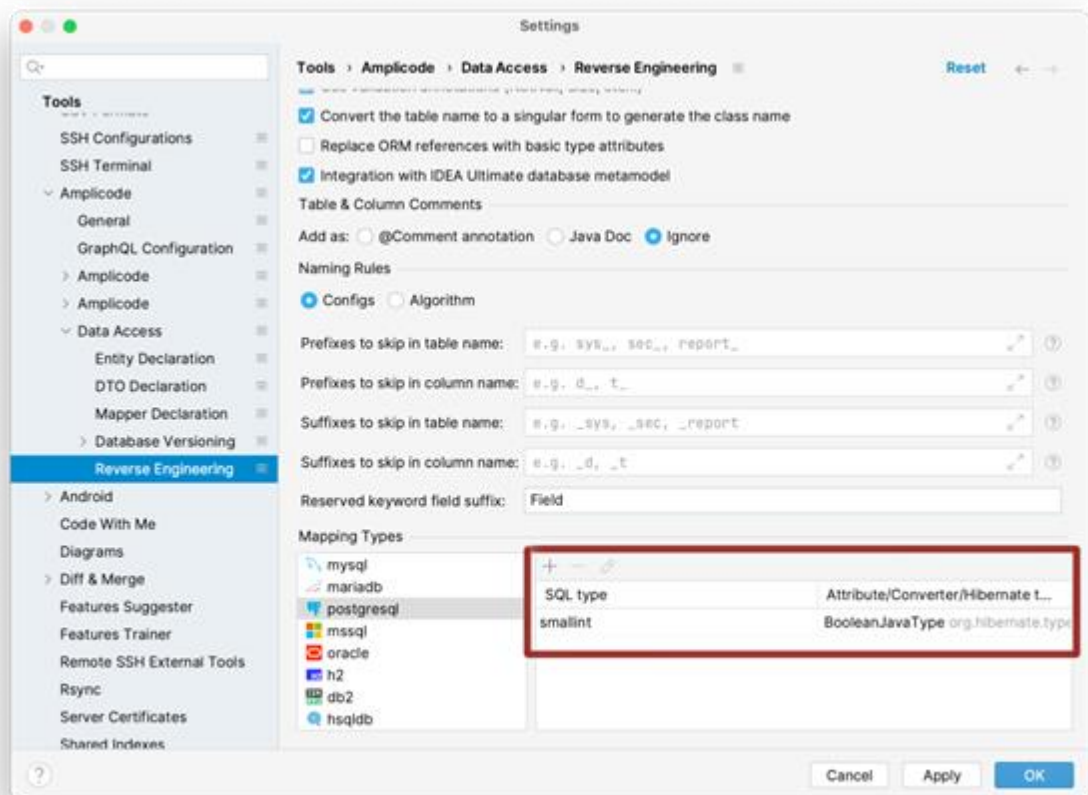
Допустим, приложение должно поддерживать как PostgreSQL, так и MS SQL, и вам необходимо хранить символы Unicode в строковых данных. PostgreSQL поддерживает символы Unicode в типе VARCHAR, но в MS SQL для этой цели существует отдельный тип данных NVARCHAR.

Amplicode позволяет задавать типы для преобразования на каждую DBMS. Можно также задавать преобразования для JPA конвертеров и Hibernate Types. Аннотация @JavaType из Hibernate 6 играет важную роль в разработке приложений, которые используют преобразования типов.

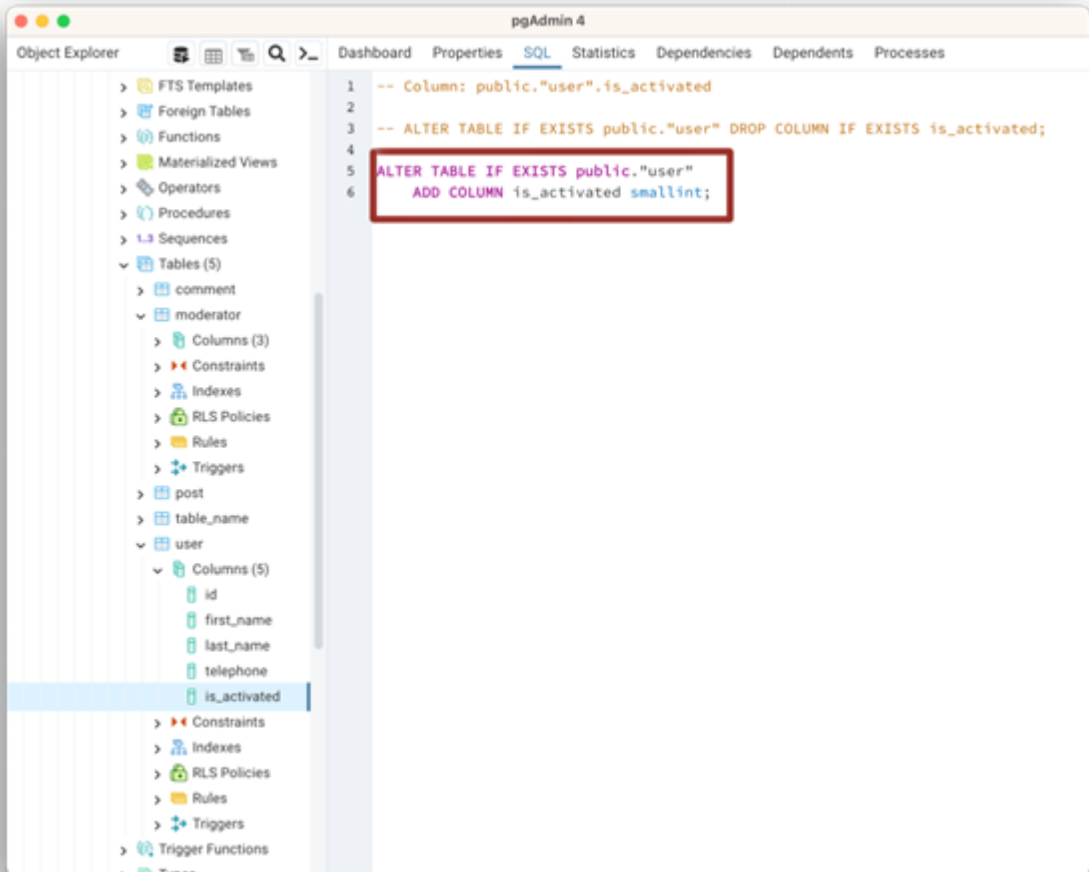
О том, как использовать конфигурацию типов для преобразования в Amplicode, чтобы использовать аннотацию @JavaType, можно [прочитать здесь](#).

Вы также можете сконфигурировать Amplicode, чтобы применить аннотацию @JavaType при выполнении reverse engineering. Для этого необходимо выполнить перечисленные ниже шаги (с примерами):

€ Преобразуем smallint на BooleanJavaType в пункте меню **Reverse Engineering Settings**:

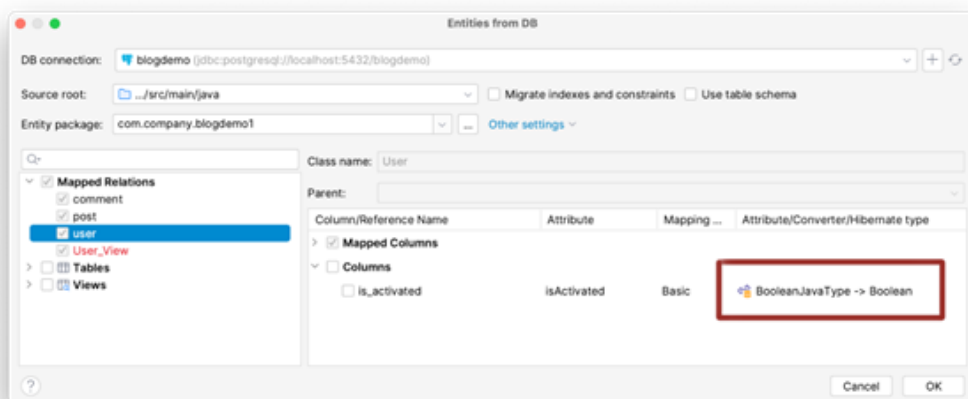


€ Добавляем поле типа smallint к базе данных.



€ Выполняем reverse engineering нового поля в сущность:

- Вызываем *JPA Entities from DB*
- Разворачиваем пункт меню *Mapped Relations* и нажимаем на имя таблицы, которую мы только что модифицировали (*user*). Новое поле (*is_activated*) будет показано в таблице справа. Должно появиться предложение от Amplicode конвертировать это поле как *BooleanJavaType* → *Boolean*.



☒ Отметьте чекбокс рядом с именем поля и нажмите **ОК**. Amplicode добавит необходимый код к классу.

```
@JavaType(BooleanJavaType.class)
@Column(name = "is_activated")
private Boolean isActivated;

public Boolean getIsActivated() {
    return isActivated;
}

public void setIsActivated(Boolean isActivated) {
    this.isActivated = isActivated;
}
```

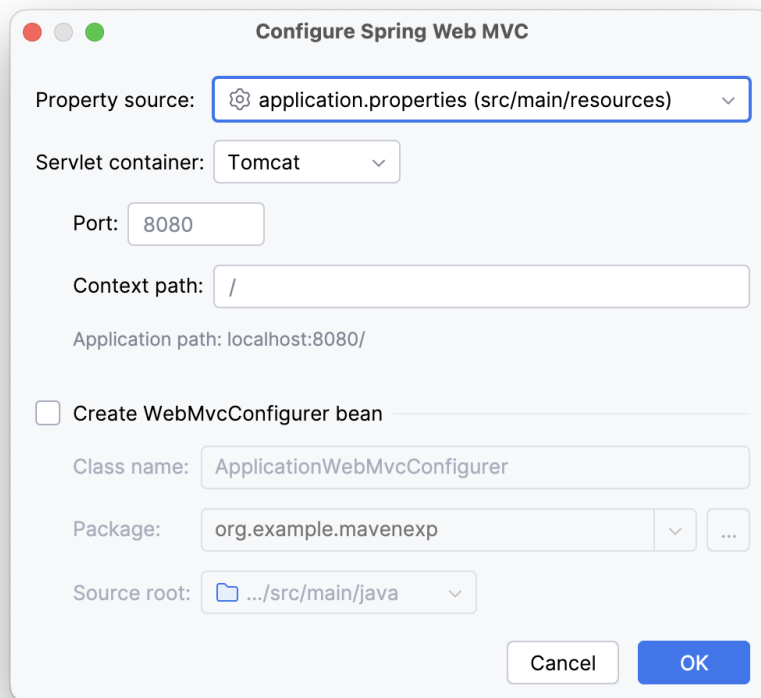
Отметим, что аннотация `@JavaType(BooleanJavaType.class)` также была добавлена к объявлению поля.

Spring MVC

Spring MVC — это веб-фреймворк Spring. Он позволяет создавать веб-сайты или RESTful сервисы (например, JSON/XML) и хорошо интегрируется в экосистему Spring, например, он поддерживает контроллеры и REST контроллеры Spring Boot приложениях. С помощью Amplicode вы сможете создать и настроить все необходимые контроллеры и их методы.

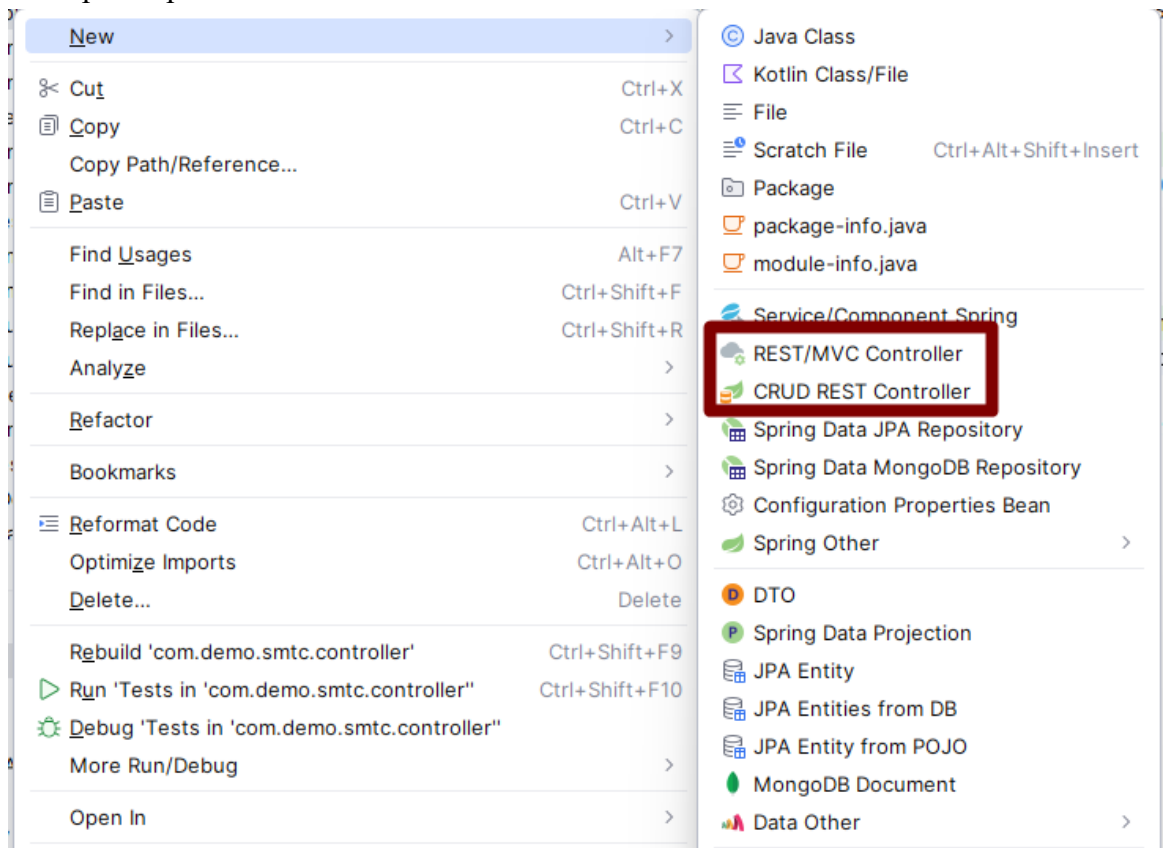
Создание контроллеров

В первую очередь, в проект необходимо добавить Spring Web конфигурацию для доступности действий по созданию контроллеров. Для этого откройте Amplicode Explorer, разверните дерево проекта, выберите Configurations -> Add Configuration -> Spring Web Configuration. В открывшемся окне нажмите Ок.

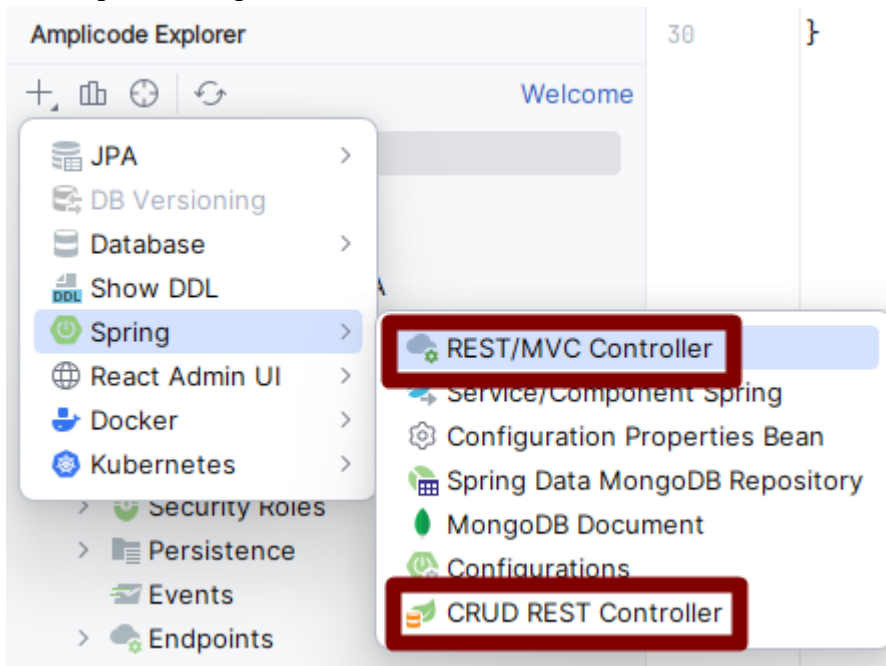


Контроллер можно создать несколькими способами

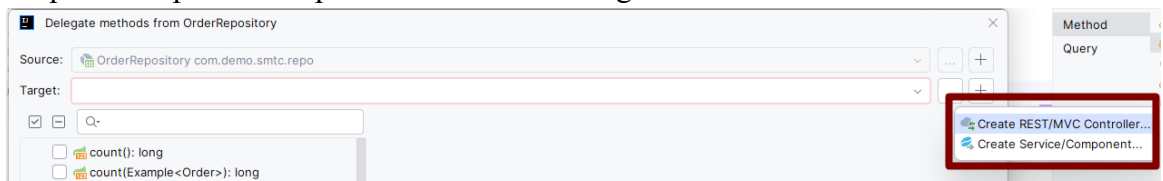
- Из дерева проекта



- Из Amplicode Explorer



- Из репозитория или сервиса в действии Delegate to



Экран создания контроллера во всех случаях выглядит следующим образом:

Для создания необходимо заполнить поле Class и нажать Ок.

Делегирование методов в контроллер

Остановимся более подробно на действии Delegate to.

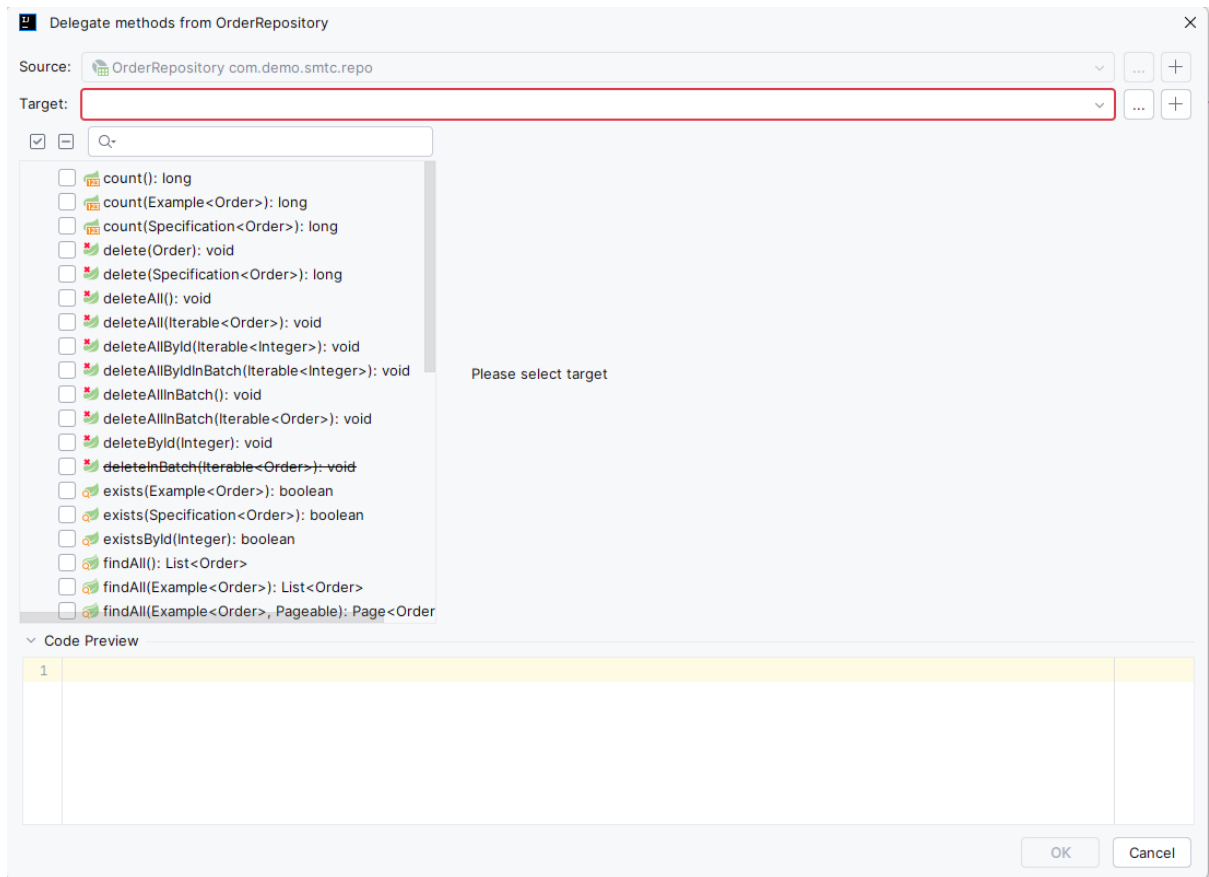
Одним из способов создания метода в контроллере является делегирование метода репозитория в метод контроллера.

Рассмотрим отдельно добавление метода из репозитория в контроллер и добавление метода в контроллере.

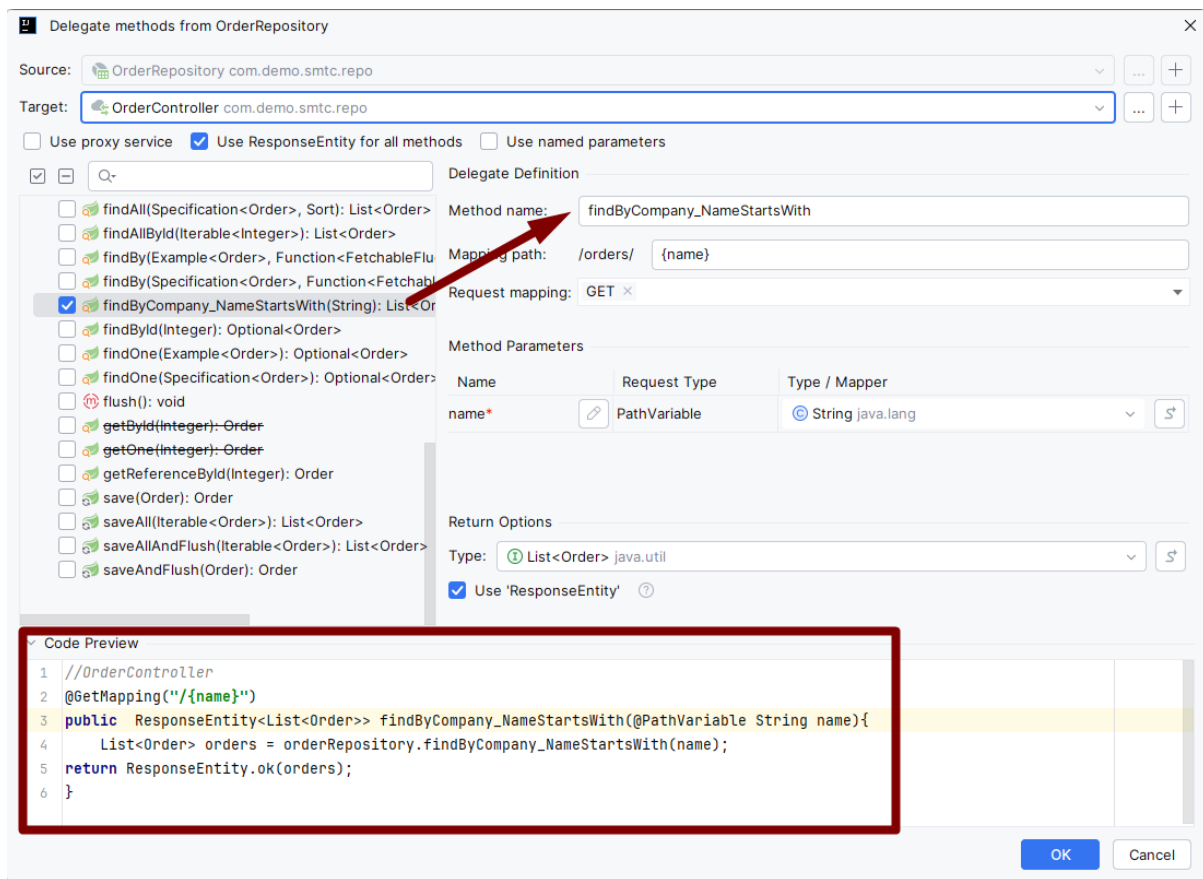
Для того, чтобы делегировать метод из репозитория в контроллер, нужно из репозитория вызвать метод Delegate to. Это можно сделать несколькими способами:

- Вызвать действие из Show Repository actions
- Установить курсор непосредственно на методе и вызвать меню Generate (ALT+Insert)

Откроется экран делегирования метода. Поле Target может быть заполнено по умолчанию, если в проекте уже создан контроллер, в противном случае необходимо создать контроллер непосредственно в окне делегирования.



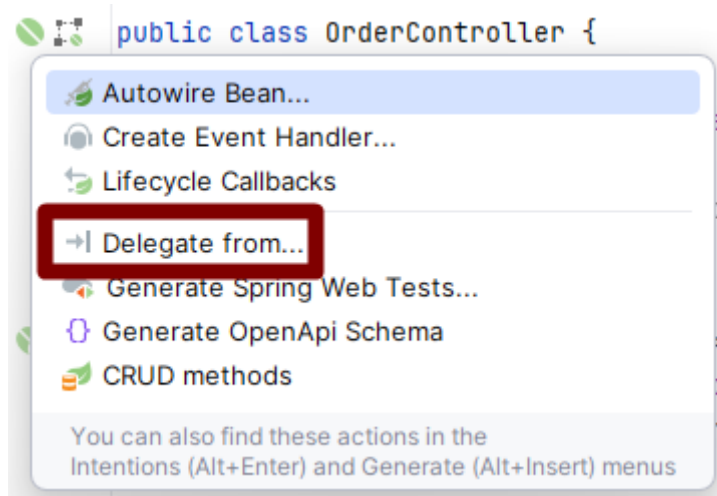
После того, как класс контроллера будет выбран, в окне отобразятся дополнительные поля для настройки делегирования метода. На экране можно изменить имя метода, запрос, параметры метода, возвращаемые параметры и посмотреть все изменения в окне Code Preview.



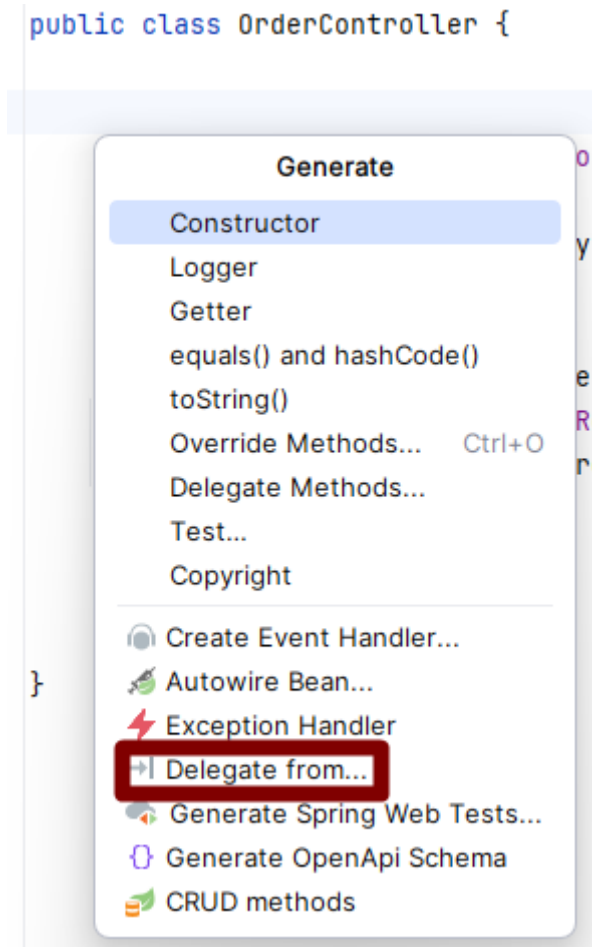
После настройки всех необходимых параметров нажмите Ок для добавления метода в контроллер.

Для того, чтобы открыть окно делегирования из контроллера, необходимо вызвать действие Delegate from. Это можно сделать несколькими способами:

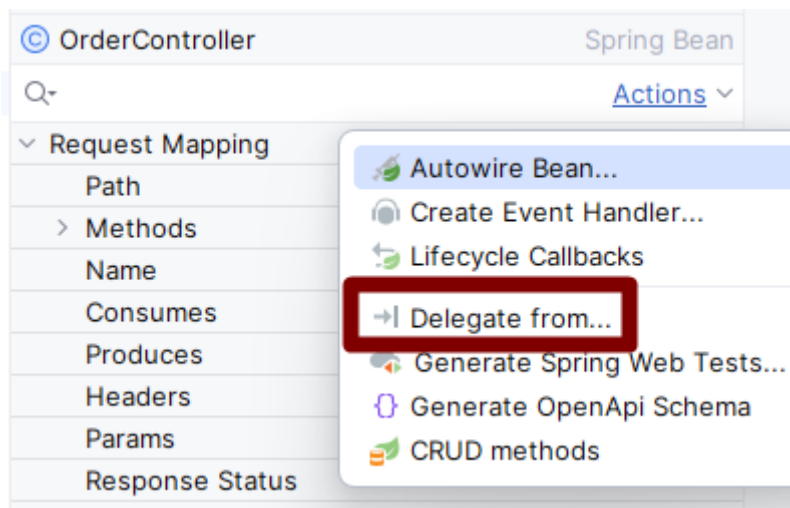
- Вызвать действие Delegate from из Show Bean actions



- Установить курсор в тело метода и вызвать меню Generate (ALT+Insert)



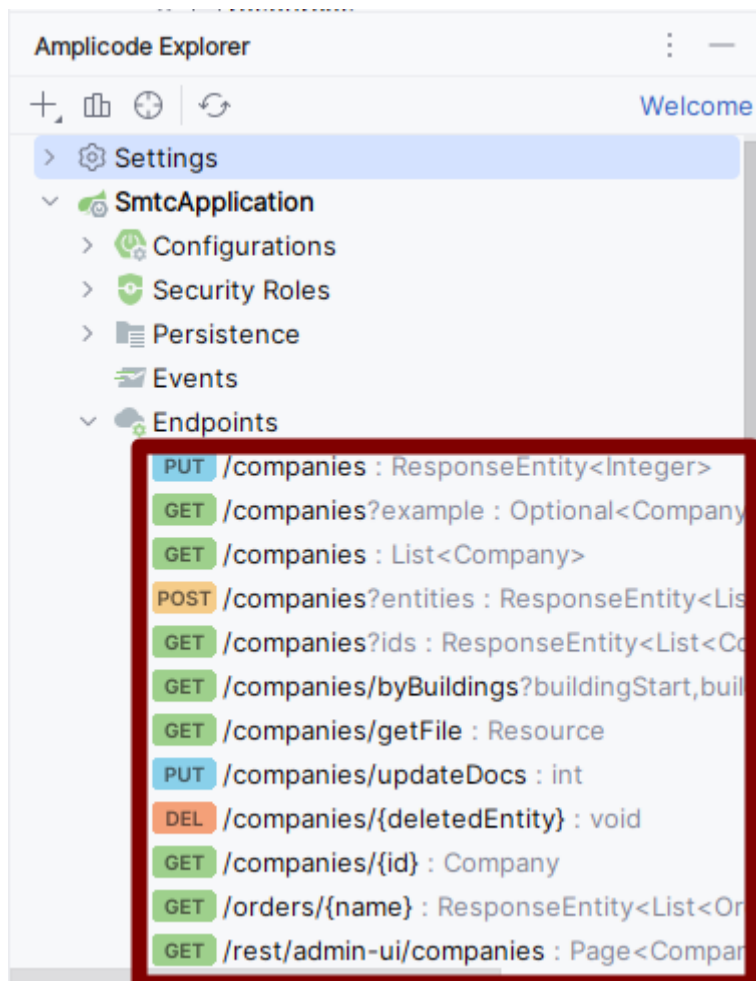
- Вызвать действие Delegate from из Инспектора для класса контроллера



Действие Delegate from открывает то же окно, что и действие Delegate to, с небольшой разницей, что поле Target уже будет предзаполнено именем контроллера и не будет доступно для редактирования и пользователю будет доступно для редактирования поле Source.

Отображение методов в дереве Amplicode Explorer

Все методы, добавленные в контроллер, будут отображаться в Amplicode Explorer в разделе Endpoints. При клике на эндпоинт в дереве произойдет переход в контроллер к конкретному методу класса.



Инспектор методов контроллера

После создания методов их можно изменять и дорабатывать с помощью Инспектора методов контроллера, расположенного в нижней части панели Дизайнера.

В инспекторе можно задать новый запрос для метода, тип метода (GET, POST, PUT), настроить параметры запроса, Response Status и даже задать ограничения доступа, если в проекте создана Security Configuration (см. раздел Spring Security)

m findByNameLike		Spring Bean Method
Q		Actions ▾
▾ Request Mapping		
Path	/{newName}	
▾ Methods		
GET	<input checked="" type="checkbox"/>	
HEAD	<input type="checkbox"/>	
POST	<input type="checkbox"/>	
PUT	<input type="checkbox"/>	
PATCH	<input type="checkbox"/>	
DELETE	<input type="checkbox"/>	
OPTIONS	<input type="checkbox"/>	
TRACE	<input type="checkbox"/>	
Name		
Consumes		
Produces		
Headers		
Params		
Response Status		
Async	<input type="checkbox"/>	
▾ Endpoint Params Add argument		
> newName		
> Cross Origin		
> Caching		
▾ Security		
Authenticated	<input checked="" type="checkbox"/>	
@Secured		
▾ Transactional		
Enabled	<input type="checkbox"/>	

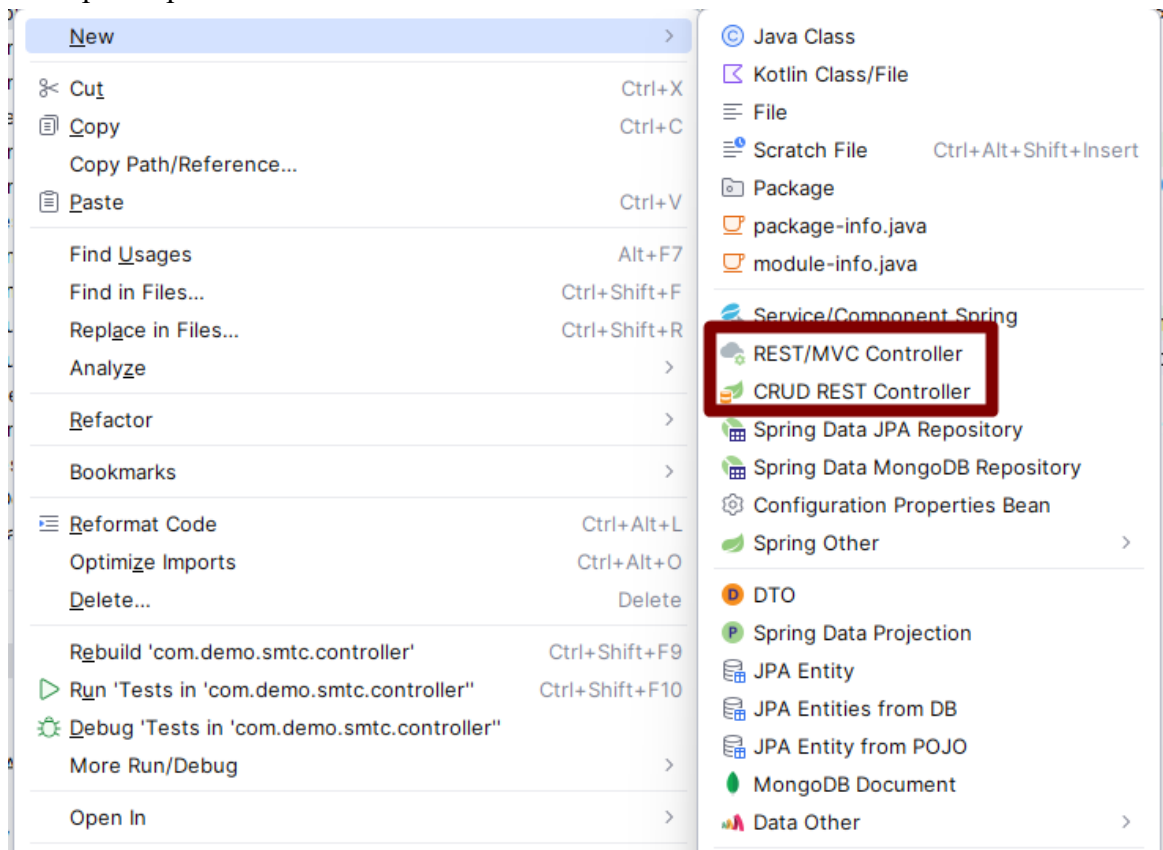
CRUD REST Controllers

Amplicode позволяет создать в проекте CRUD REST Controller - контроллер с минимальным набором стандартных методов для операций Create, Read, Update, Delete:

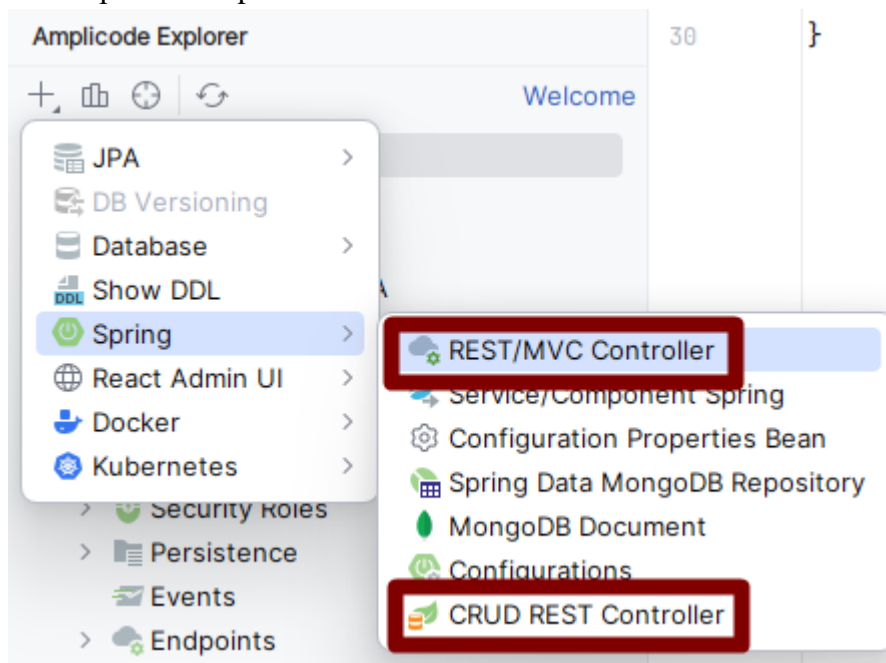
- операции создания — создание ресурса через метод POST;
- операции чтения — возврат представления ресурса через метод GET;
- операции редактирования — перезапись ресурса через метод PUT или редактирование через PATCH;
- операции удаления — удаление ресурса через метод DELETE.

Контроллер можно создать несколькими способами:

- Из дерева проекта



- Из Amplicode Explorer



После выбора действия откроется окно создания репозитория

New CRUD REST Controller

JPA repository: CompanyRepository com.demo.bnr

DTO class: Company com.demo.bnr
Entity or MapStruct DTO

Controller class: CompanyResource

Request path: /rest/admin-ui (Base path) /companies (Resource path)

List filter: <None>

Proxy service: <None>

Pagination

Source root: [bnr.main] .../src/main/java

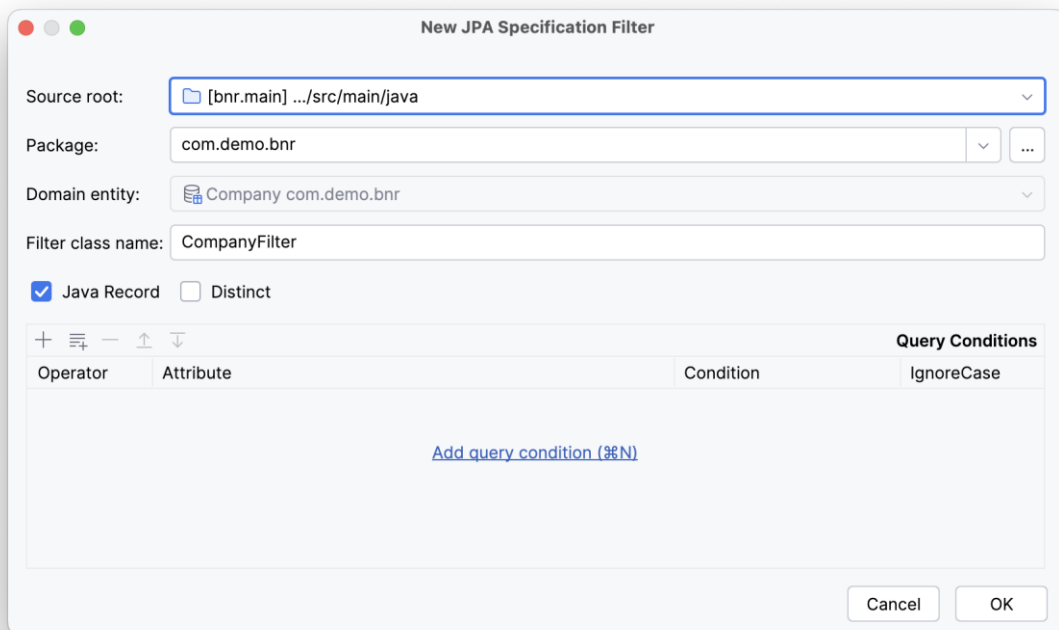
Install Amplicode React Admin Utils
A Spring starter that simplifies development of React Admin CRUD controllers. [Learn more](#)

Package: com.demo.bnr

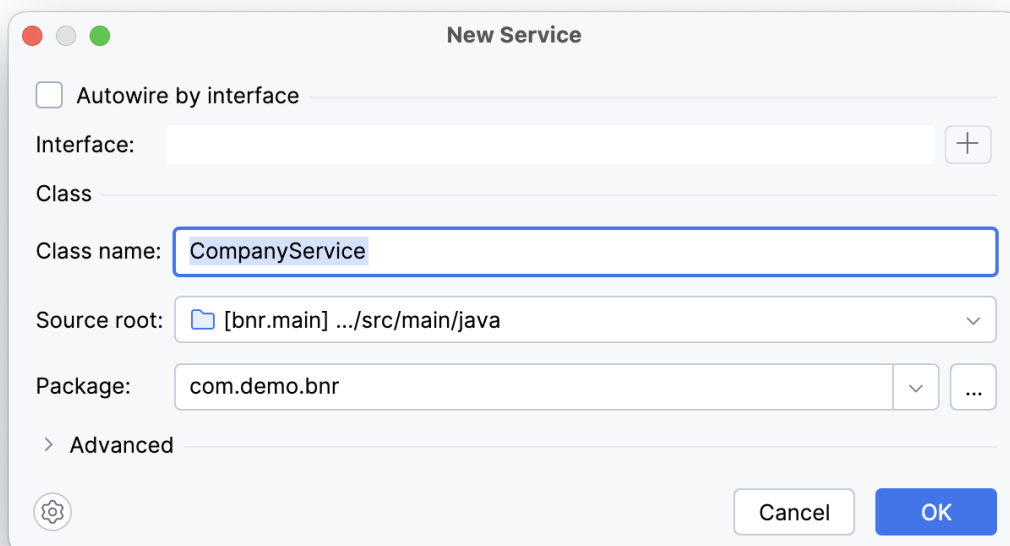
Cancel OK

Поля JPA Repository и DTO class заполняются автоматически классами из проекта, новые классы можно создать непосредственно из диалога, нажав на кнопку плюс в поле. После нажатия Ок в проекте генерируется контроллер со стандартными CRUD-методами для выбранной сущности или DTO.

Опционально в экране создания можно настроить фильтр



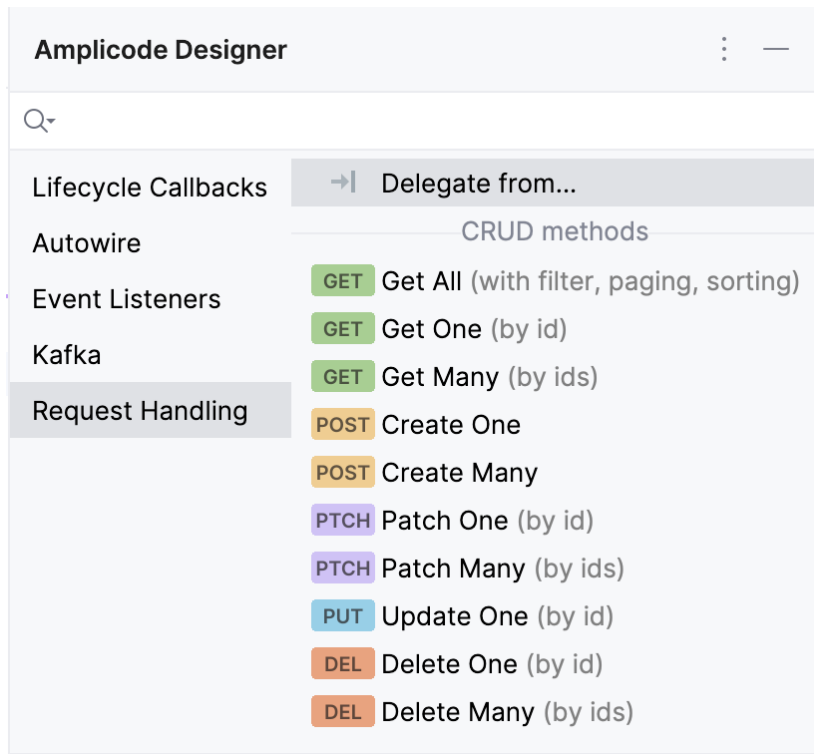
и прокси сервис для репозитория.



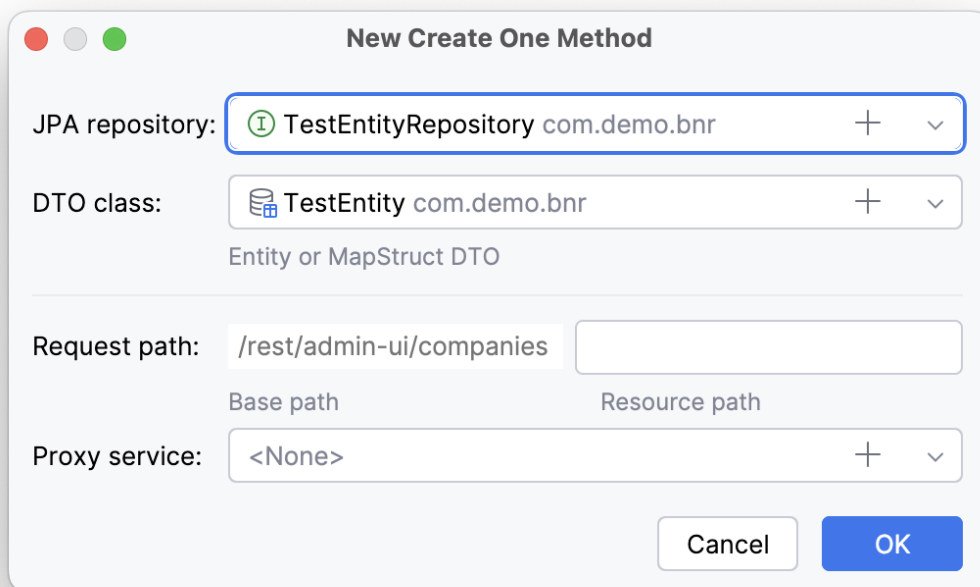
Методы контроллера так же отображаются в Amplicode Explorer в разделе Endpoints. Для контроллера так же доступен дизайнер.

Создание новых методов в контроллере

В Amplicode Designer в Палитре в разделе Request Handling доступна возможность генерации дополнительных CRUD-методов в контроллере.



После выбора нужного метода откроется окно создания, в котором можно указать репозиторий и сущность/DTO, для которой нужно создать метод. Так же можно настроить запрос и прокси сервис.



Для каждого метода так же доступен Инспектор, в котором можно настроить дополнительные параметры.

m getAll		Spring Bean Method
Q		Actions ▾
▾ Request Mapping		
Path	/rest/admin-ui/companies	
▾ Methods		
GET	<input checked="" type="checkbox"/>	
HEAD	<input type="checkbox"/>	
POST	<input type="checkbox"/>	
PUT	<input type="checkbox"/>	
PATCH	<input type="checkbox"/>	
DELETE	<input type="checkbox"/>	
OPTIONS	<input type="checkbox"/>	
TRACE	<input type="checkbox"/>	
Name		
Consumes		
Produces		
Headers		
Params		
Response Status		
Async	<input type="checkbox"/>	
Endpoint Params	Add argument	
▾ Cross Origin		
Enabled	<input type="checkbox"/>	
▾ Caching		
Enabled	<input type="checkbox"/>	
▾ Security		
Authenticated	<input checked="" type="checkbox"/>	
@Secured		
▾ Transactional		
Enabled	<input type="checkbox"/>	

Spring Security

Amplicode предоставляет набор конфигураторов и генераторов для управления настройкой аутентификации и авторизации в приложении средствами Spring Security.

Поддерживаются несколько типов аутентификации:

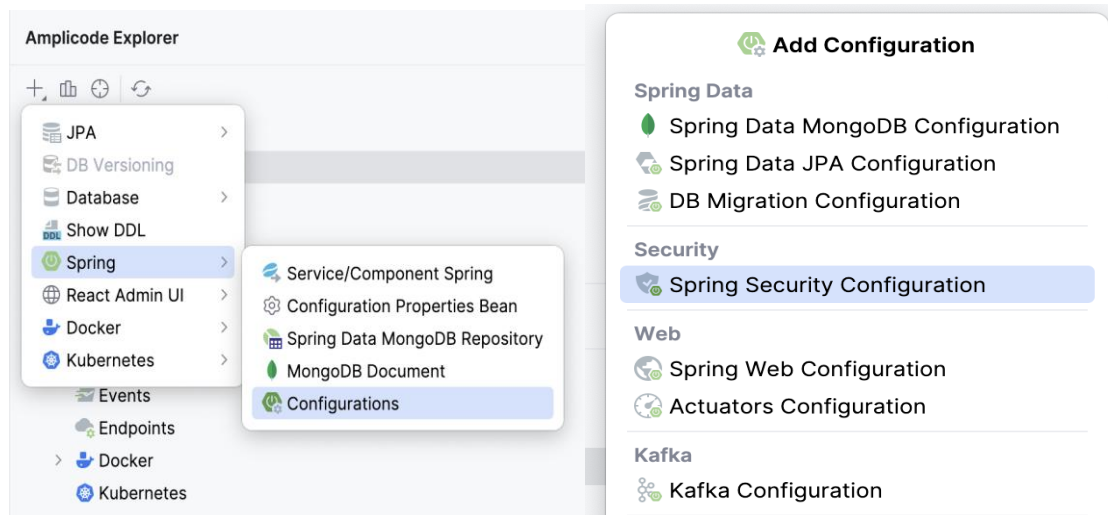
- Аутентификация с использованием HTTP сессией. Пользователи хранятся в памяти приложения, в JPA или в другом месте.
- Аутентификация без использования HTTP сессии через JWT токен.
- OIDC аутентификация с использованием HTTP сессии.
- LDAP аутентификация
- Прочие

Помимо настройки типа аутентификации Amplicode позволяет детально настраивать Spring Security и настраивать правила авторизации для REST API с помощью ролей Spring Security.

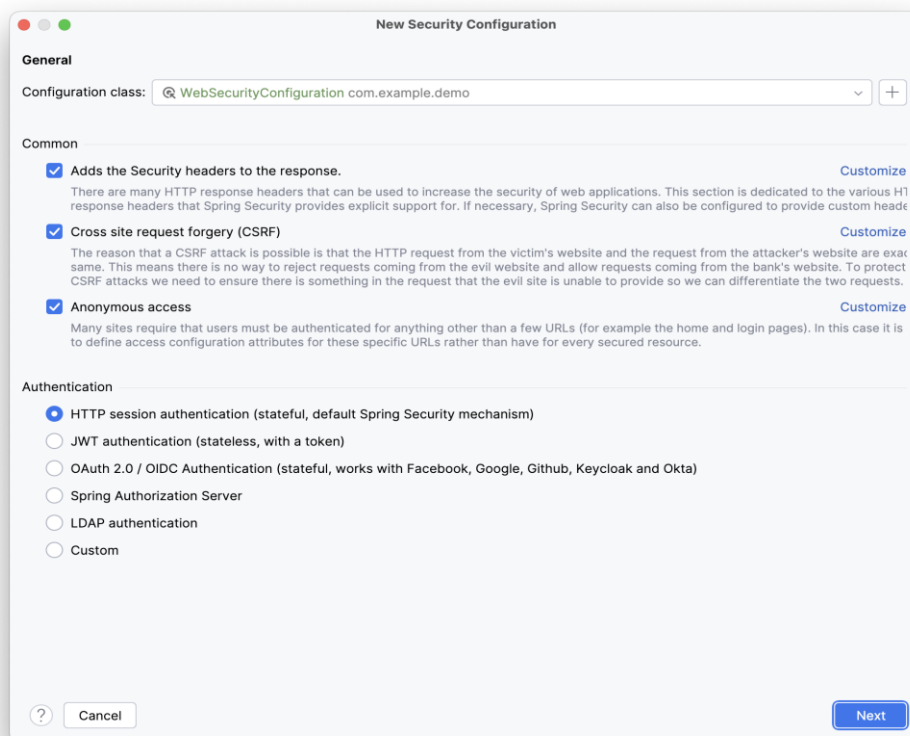
Основным элементом Spring Security является Security Configuration, описывающая правила аутентификации, авторизации, CORS и прочее. Рассмотрим инструменты по созданию Security Configuration.

Создание Security Configuration

Для создания Security Configuration воспользуйтесь действием Create (+) в Amplicode Explorer и выберите Spring -> Configurations -> Spring Security Configuration.



Откроется диалог позволяющий настроить тип аутентификации и прочие возможности Spring Security.



На данном шаге можно обновить существующий класс Security Configuration или создать новый. Настройка осуществляется в поле *Configuration Class*.

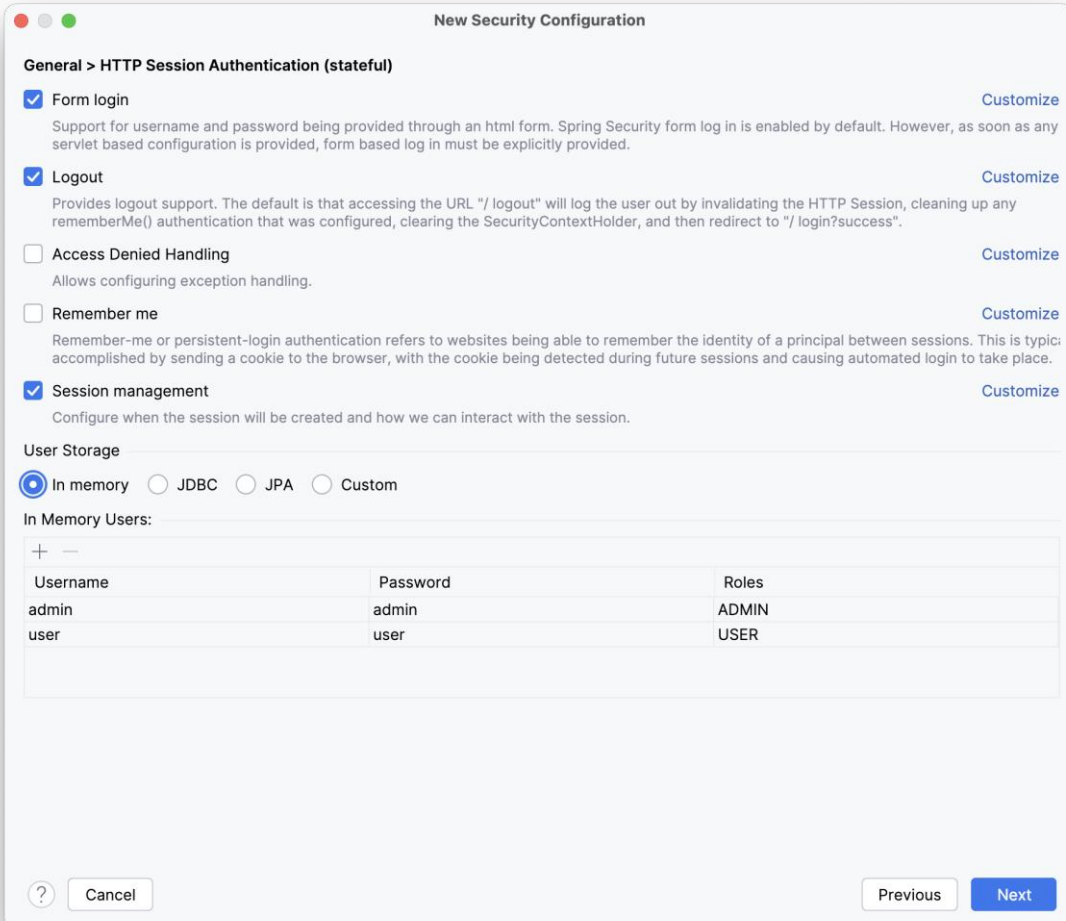
Рассмотрим настройки Security Configuration для некоторых основных типов аутентификации.

Аутентификация с использованием HTTP сессией

В диалоге создания Security Configuration для *Authentication* выбираем HTTP session authentication (statefull, default Spring Security mechanism) и нажимаем *Next*, переходя к следующему шагу.

Указываем параметры:

- *User storage*: выбираем хранилище для пользователей в памяти, или в базе данных, или с использованием JPA сущностей. Возможно изменить список предустановленных пользователей и их роли
- Остальные параметры можно оставить неизменными



The screenshot shows the 'New Security Configuration' dialog box. The title bar reads 'New Security Configuration'. The main section is 'General > HTTP Session Authentication (stateful)'. It contains several options with 'Customize' links:

- Form login** (Customize): Support for username and password being provided through an html form. Spring Security form log in is enabled by default. However, as soon as any servlet based configuration is provided, form based log in must be explicitly provided.
- Logout** (Customize): Provides logout support. The default is that accessing the URL "/logout" will log the user out by invalidating the HTTP Session, cleaning up any rememberMe() authentication that was configured, clearing the SecurityContextHolder, and then redirect to "/login?success".
- Access Denied Handling** (Customize): Allows configuring exception handling.
- Remember me** (Customize): Remember-me or persistent-login authentication refers to websites being able to remember the identity of a principal between sessions. This is typically accomplished by sending a cookie to the browser, with the cookie being detected during future sessions and causing automated login to take place.
- Session management** (Customize): Configure when the session will be created and how we can interact with the session.

Below this is the 'User Storage' section with radio buttons for 'In memory' (selected), 'JDBC', 'JPA', and 'Custom'.

Under 'In Memory Users:', there is a table:

Username	Password	Roles
admin	admin	ADMIN
user	user	USER

At the bottom, there are buttons for '?', 'Cancel', 'Previous', and 'Next'.

Нажимаем Next.

Проходим следующие шаги без внесения изменений изменений и нажимаем кнопку Create.

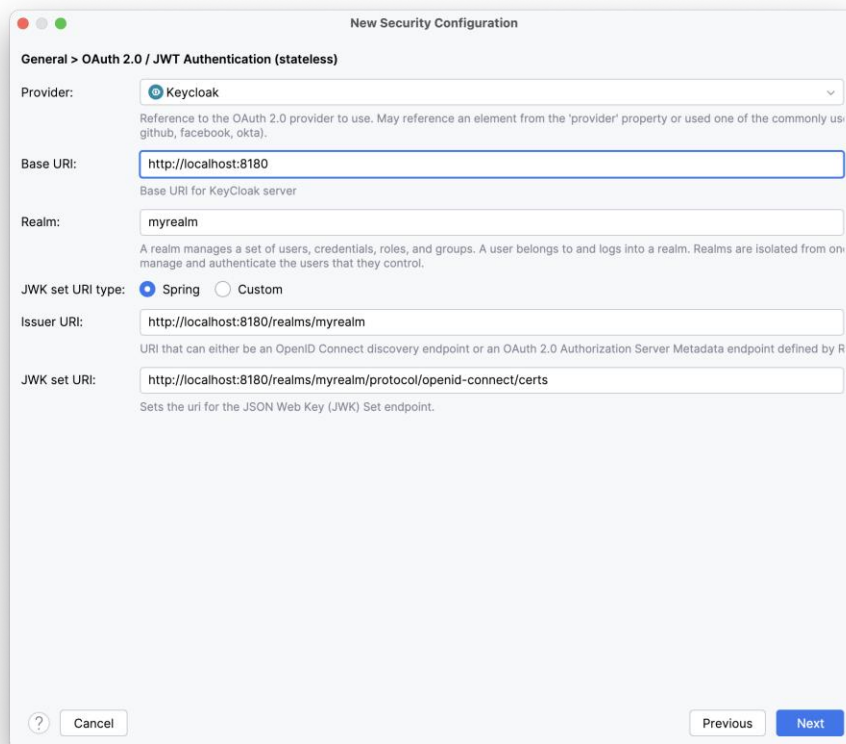
В результате работы будут создан класс `WebSecurityConfiguration` с описанием настроек безопасности для аутентификации с использованием HTTP сессии.

JWT аутентификация с Keycloak

В диалоге создания Security Configuration для Authentication выбираем JWT authentication (stateless, with a token) и нажимаем Next, переходя к следующему шагу.

Указываем параметры:

- *Provider*: Keycloak
- *Base URI*: URI где расположен KeyCloak. Например, <http://localhost:8180>
- *Realm name*: Имя realm в Keycloak
- Остальные параметры можно оставить не измененными



Нажимаем Next.

Проходим следующие шаги без внесения изменений изменений и нажимаем кнопку Create.

В результате работы будут добавлены необходимые библиотеки в проект и создан класс `WebSecurityConfiguration` с описанием настроек безопасности для JWT аутентификации.

Так же будут добавлены дополнительные свойства в файл `application.properties`:

```
#Security configuration spring.security.oauth2.resourceserver.jwt.issuer-  
uri=http://localhost:8180/realms/myrealm spring.security.oauth2.resourceserver.jwt.jwk-set-  
uri=http://localhost:8180/realms/myrealm/protocol/openid-connect/certs
```

OIDC аутентификация с использованием HTTP сессии

В диалоге создания Security Configuration для *Authentication* выбираем OAuth2.0/OIDC authentication (statefull, working with Google ...) и нажимаем *Next*, переходя к следующему шагу.

Указываем параметры:

- *Provider*: Keycloak
- *Base URI*: URI где расположен KeyCloak. Например, <http://localhost:8180>
- *Realm name*: Имя realm в Keycloak
- *Client ID*: идентификатор клиента в Keycloak
- *Client Secret*: секретный ключ, в случае если он настроен для клиента в Keycloak
- Остальные параметры можно оставить не измененными

General > OAuth 2.0 / OIDC Authentication (stateful)

Provider:

Reference to the OAuth 2.0 provider to use. May reference an element from the 'provider' property or used one of the commonly known providers (github, facebook, okta).

Base URI:

Base URI for KeyCloak server

Realm:

A realm manages a set of users, credentials, roles, and groups. A user belongs to and logs into a realm. Realms are isolated from each other and manage and authenticate the users that they control.

Client ID:

Client ID for the registration.

Issuer URI:

URI that can either be an OpenID Connect discovery endpoint or an OAuth 2.0 Authorization Server Metadata endpoint defined by the provider.

Client secret:

Client secret for the registration.

Authorization scopes:

The scope(s) used for the client. Comma separated list.

? Cancel Previous Next

Нажимаем Next.

Проходим следующие шаги без внесения изменений и нажимаем кнопку Create.

В результате работы будут добавлены необходимые библиотеки в проект и создан класс `WebSecurityConfiguration` с описанием настроек безопасности для OAuth2/OIDC аутентификации.

Так же будут добавлены дополнительные свойства в файл `application.properties`:

```
#Security configuration
spring.security.oauth2.client.registration.keycloak.client-id=clientId
spring.security.oauth2.client.registration.keycloak.provider=keycloak
spring.security.oauth2.client.registration.keycloak.client-secret=clientSecret
spring.security.oauth2.client.registration.keycloak.client-name=KEYCLOAK
spring.security.oauth2.client.registration.keycloak.scope=openid
spring.security.oauth2.client.provider.keycloak.issuer-uri=http://localhost:8180/realms/myrealm
```

Настройка правил авторизации в Spring Security

Правила авторизации в Spring Security позволяют настраивать различный доступ к ресурсам, в том числе:

- Анонимный доступ, не требующий аутентификации в приложение
- Аутентифицированный доступ
- Доступ с определенными ролями

Правила авторизации могут задаваться в следующих местах:

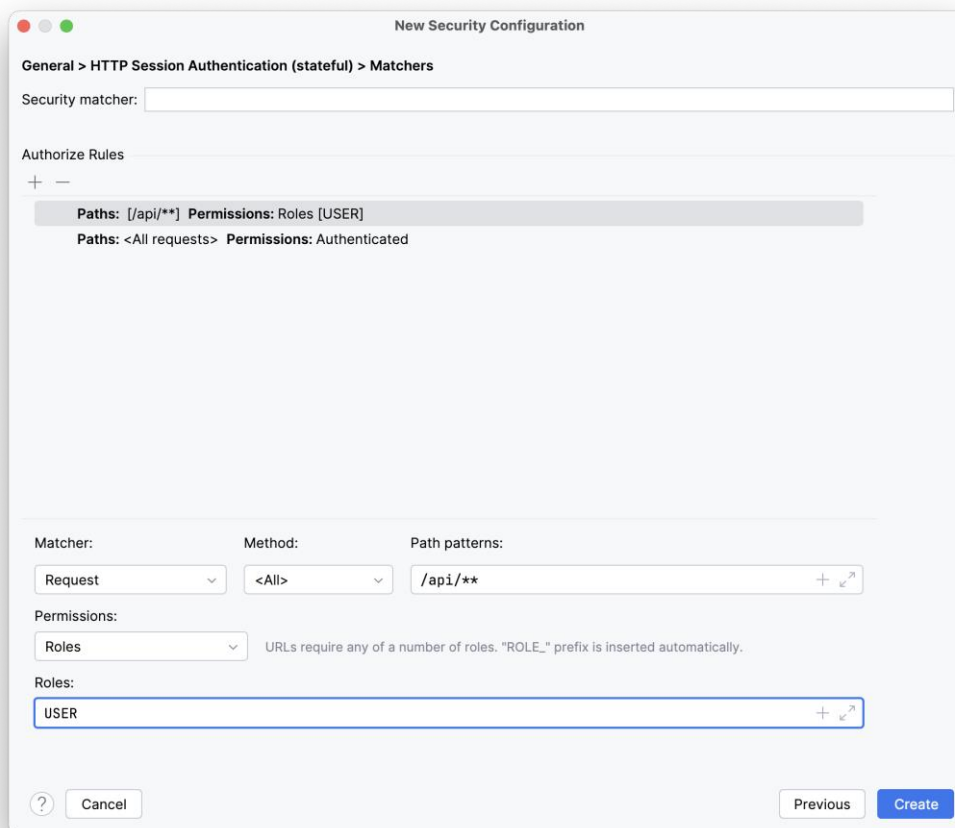
- На Security Configuration
- На методах WEB/REST контроллеров в виде настройке ролей для доступа

Amplicode позволяет упростить настройку правил авторизации.

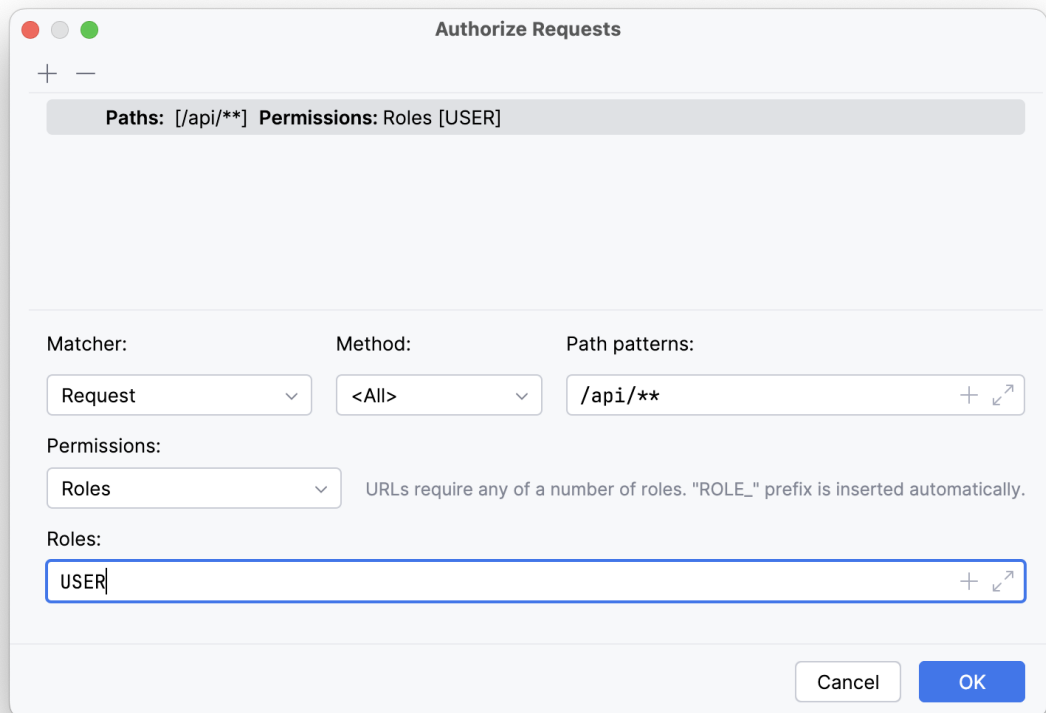
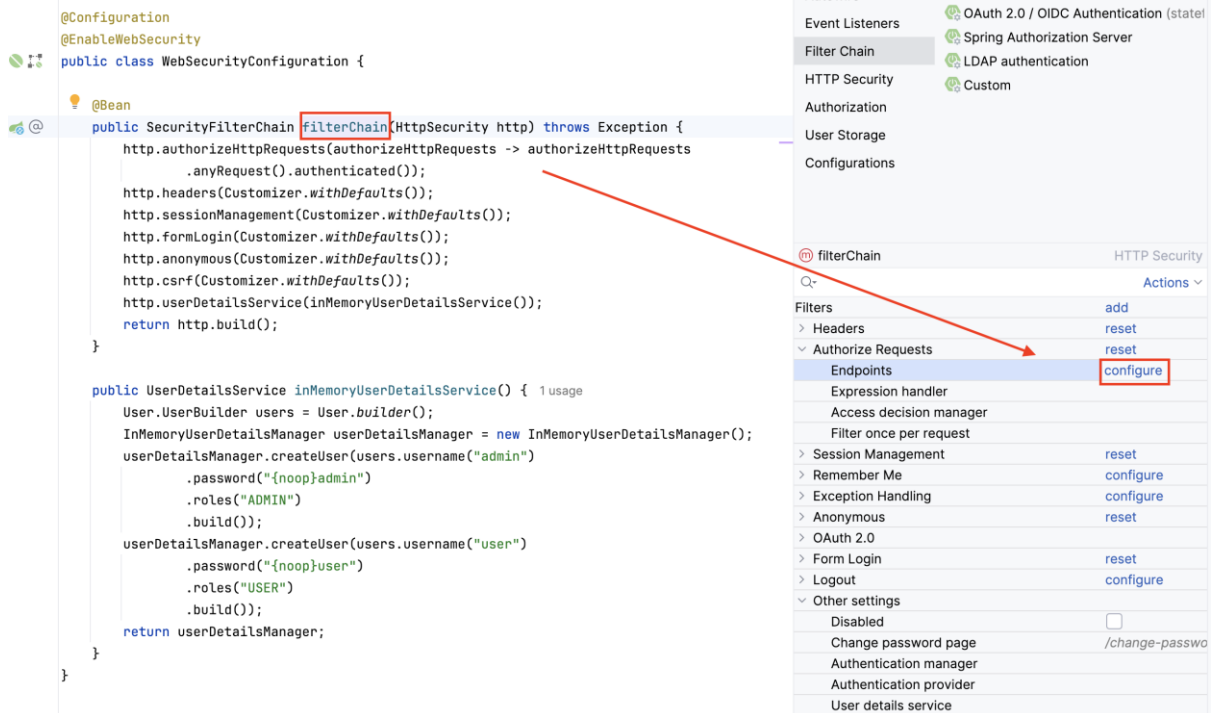
Настройка правил авторизации для Security Configuration

Настройка правил авторизации или Security Matchers доступна для Security Configuration из следующих мест:

- При создании Security Configuration, одним из шагов в диалоге создания.



- В инспекторе свойств для Security Configuration при нахождении курсора на методе *filterChain* в исходном коде.



Данные диалоги позволяют настроить разрешения *Permissions* для различных типов запросов *Matchers* к приложению. Подробности по настройке можно посмотреть в документации [Spring Security](#). Например, на диалогах выше, все запросы с путем `/api/**` доступны пользователям с ролью `USER`.

Результатом выполнения данных действий является обновление класса Security Configuration в соответствии с указанными правилами авторизации, а именно:

@Bean

```
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.headers(Customizer.withDefaults());
    http.sessionManagement(Customizer.withDefaults());
    http.formLogin(Customizer.withDefaults());
    http.anonymous(Customizer.withDefaults());
    http.csrf(Customizer.withDefaults());
    http.userDetailsService(inMemoryUserDetailsService());
    http.authorizeHttpRequests(authorizeHttpRequests -> authorizeHttpRequests
        .requestMatchers(...patterns: "/api/**").hasRole("USER")
        .anyRequest().authenticated());
    return http.build();
}
```

Настройки ролевого доступа в методах MVC/REST контроллеров

Для входных точек REST API возможна настройка ролевого доступа: указание списка ролей, для которых разрешено использовать текущий метод REST API. Настройка возможна как в классе Security Configuration (см. предыдущий раздел), так и непосредственно в классе REST контроллера при помощи механизма @Secured из Spring Security.

Для изменения ролевого доступа находим требуемый метод REST API контроллера и нажимаем за иконку “замка”.

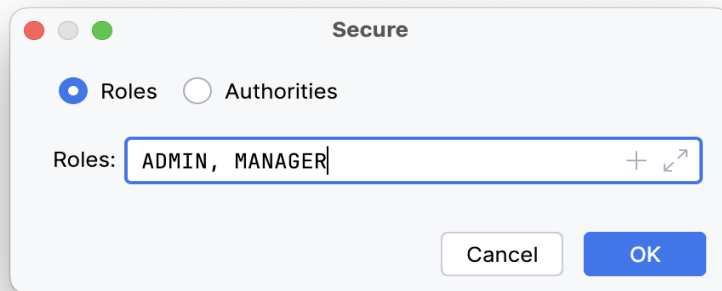


```
1  @GetMapping
2  public List<Customer> findAll() {
3      return customerRepository.findAll();
4  }
```


В открывшемся окне можно выполнить следующие действия:

- Add authorize rule. Добавляет REST API метод в правила авторизации для Security Configuration (см. предыдущий раздел).
- @Authenticated. Разрешает доступ только аутентифицированным пользователям к методу REST API
- @Secured. Настраивает ролевой доступ к методу REST API

При выборе @Secured предоставляется возможность указать список ролей, которым разрешен доступ к методу REST API.

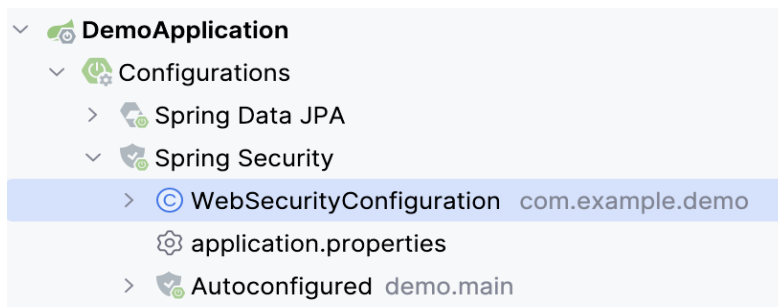


Результатом выполнения данных действий является обновление метода REST API: а именно добавление `@Secured` с указанием разрешенных для метода ролей:

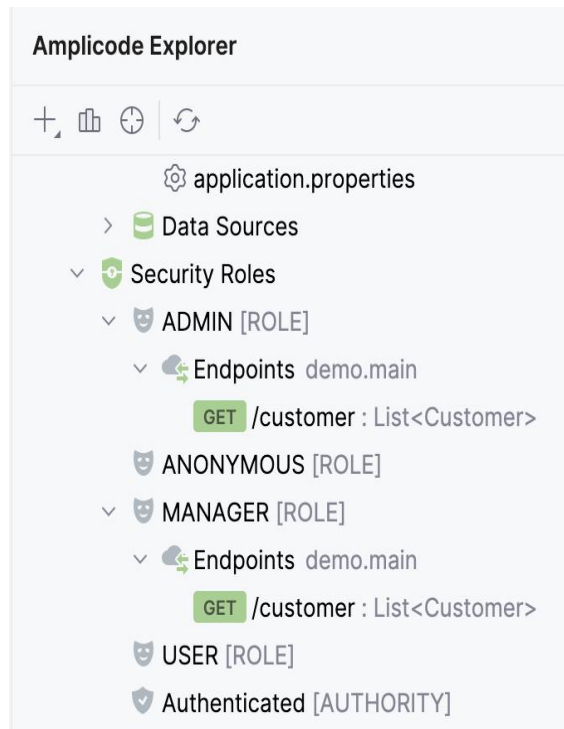
```
@GetMapping   
@Secured({"ROLE_ADMIN", "ROLE_MANAGER"})  
public List<Customer> findAll() {  
    return customerRepository.findAll();  
}
```

Отображение Spring Security в дереве Amplicode Explorer

Дерево Amplicode Explorer позволяет в структуре проекта быстро перейти к Security Configuration, через Configurations -> Spring Security.

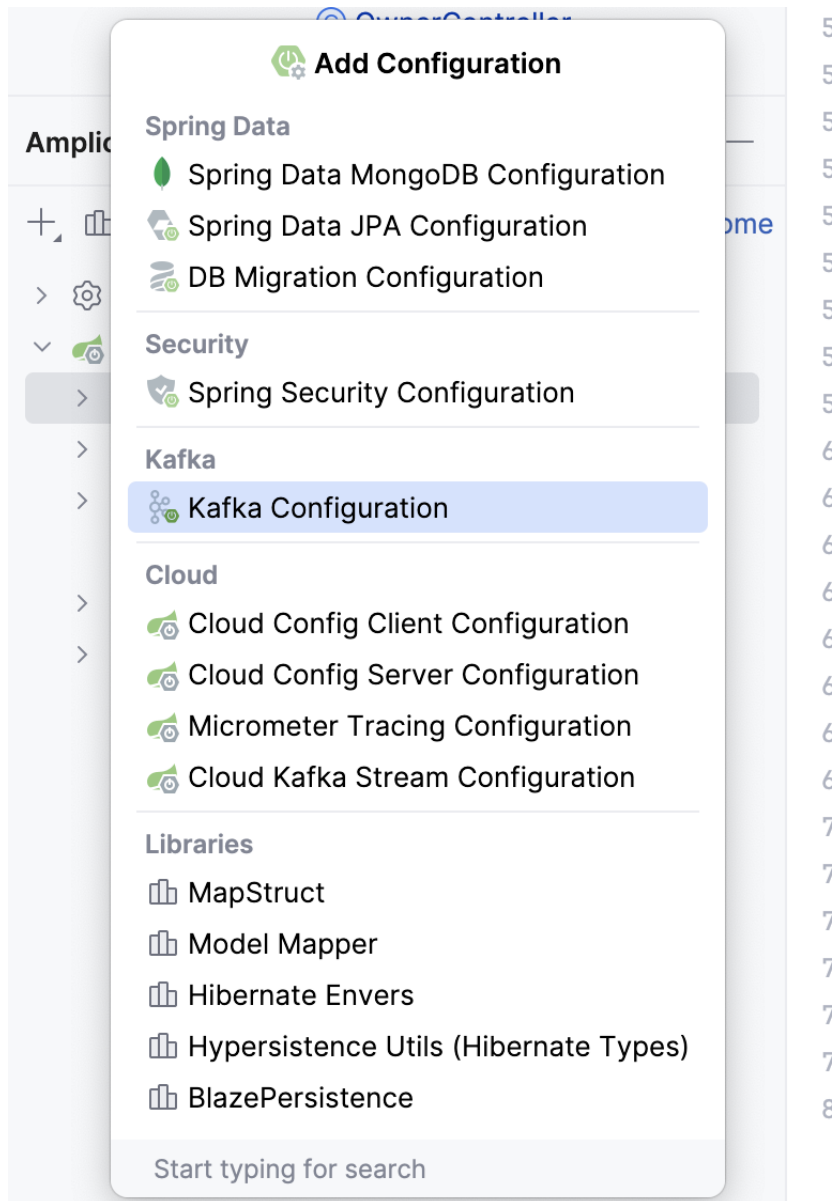


Дерево Amplicode Explorer позволяет просмотреть настроенные в проекте роли безопасности и связанные с ними REST API/WEB методы в разделе Security Roles.

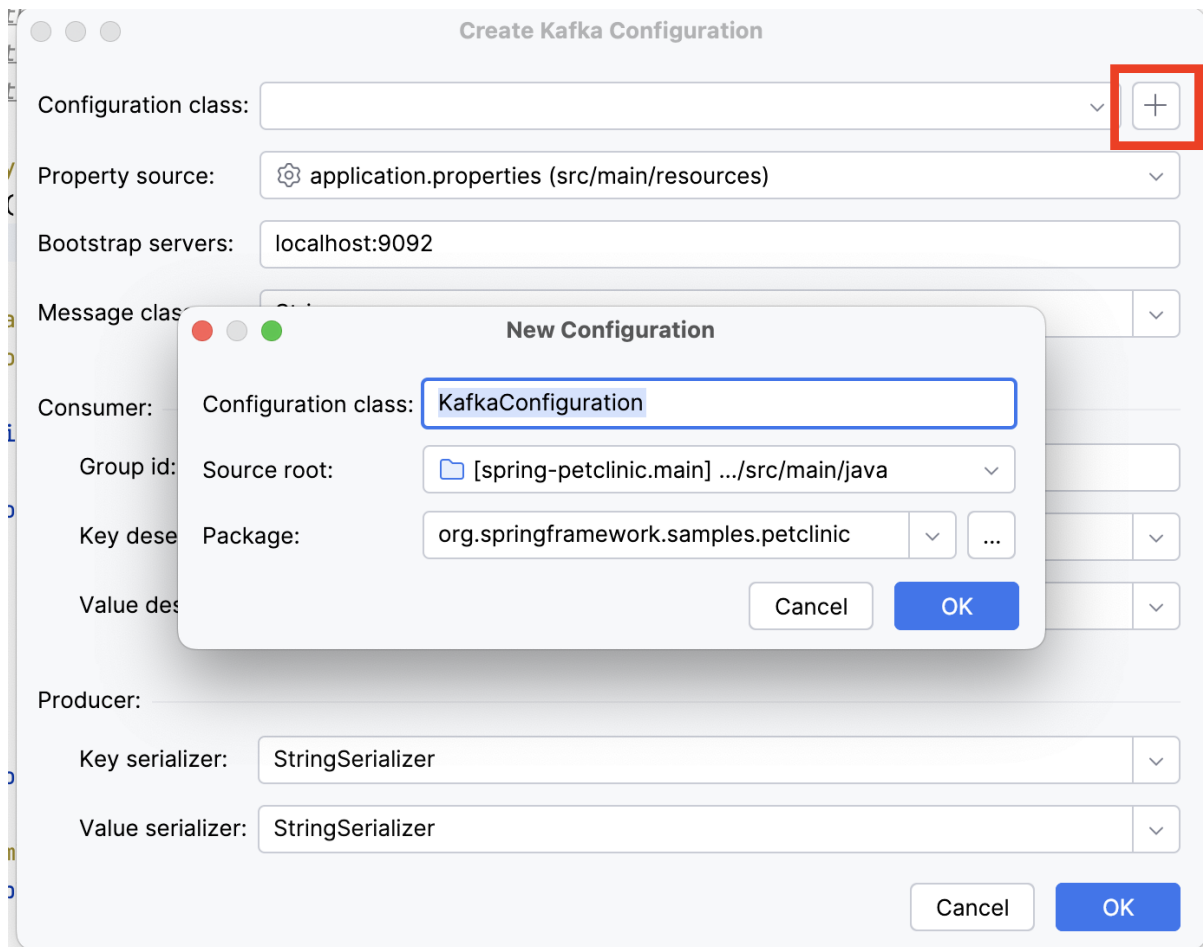


Kafka

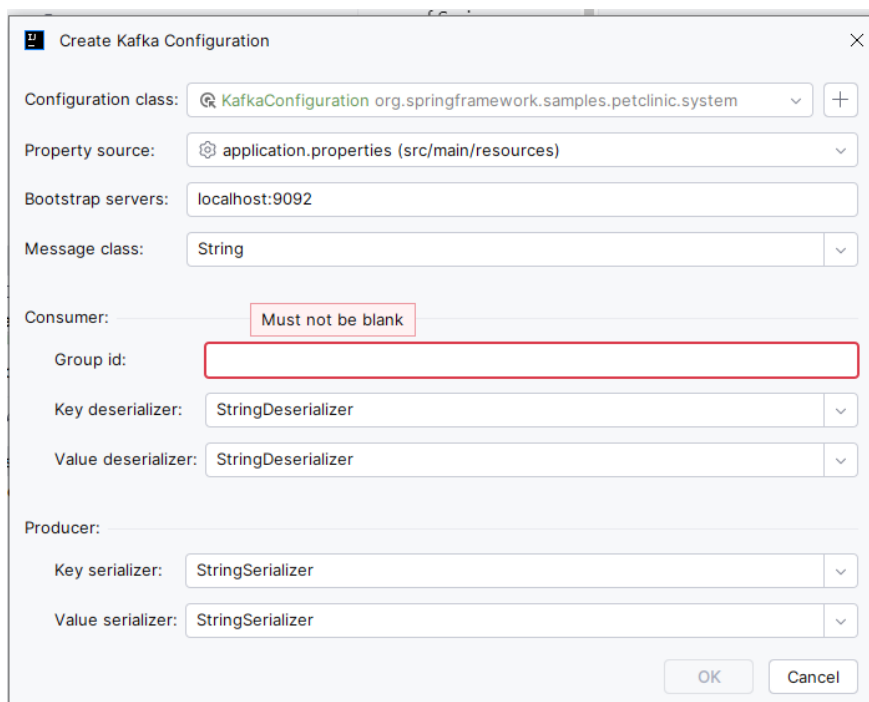
Amplicode поддерживает работу с брокером сообщений Kafka. Для того чтобы подключить Kafka конфигурацию необходимо воспользоваться Amplicode Explorer, нажав правой клавишей мыши на пункте Configurations и выбрав Add Configuration и затем Kafka Configuration.



Для создания нового конфигурационного класса для настроек, связанных с Kafka, можно с помощью всплывающего окна при нажатии на иконку плюса.



По умолчанию название класса `KafkaConfiguration`. После подтверждения создания конфигурация подставляется в поле `Configuration class`.



В диалоговом окне после заполнения обязательного поля Group id и нажатия на ОК создается конфигурационный класс, а также добавляется зависимость на spring-kafka и настраиваются значения в файле свойств application.properties, например:

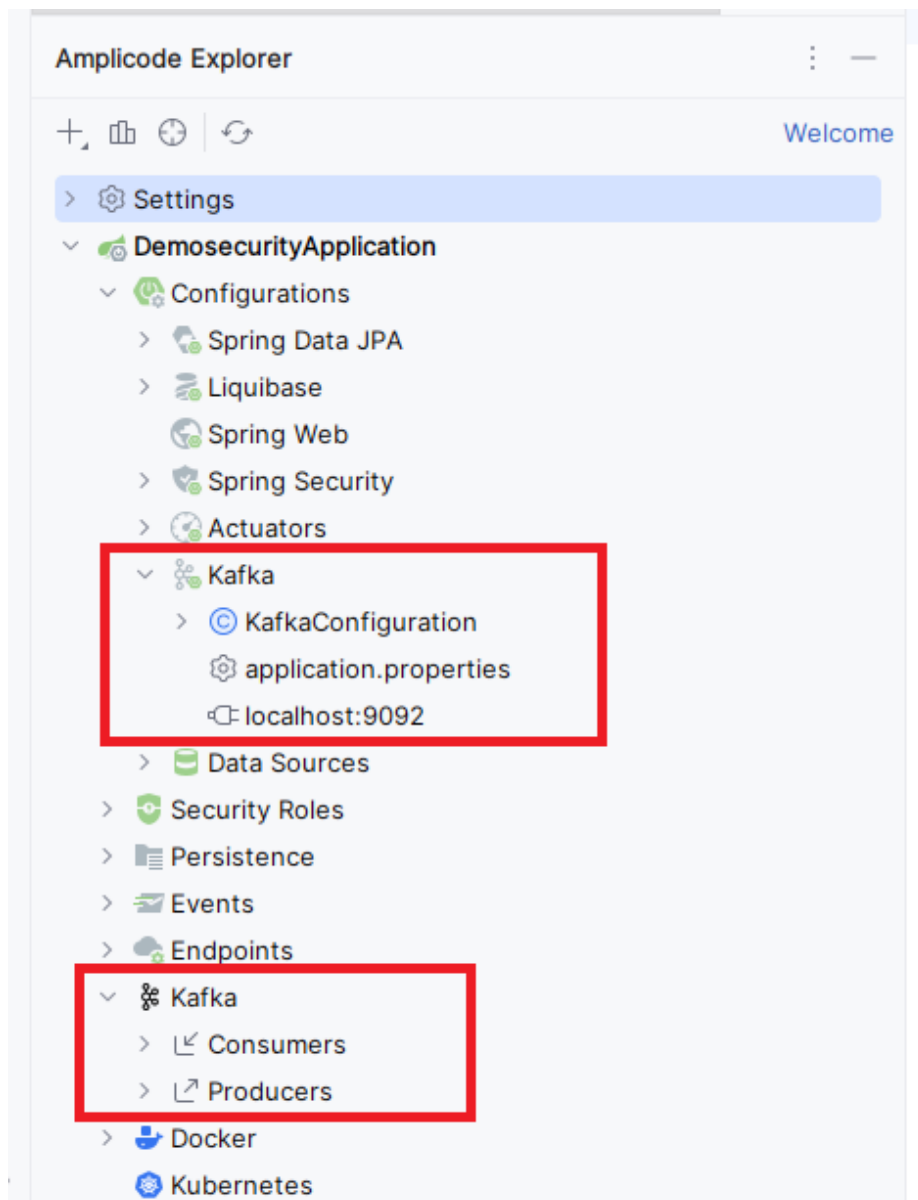
#Spring Kafka

```
spring.kafka.consumer.group-id=demo
```

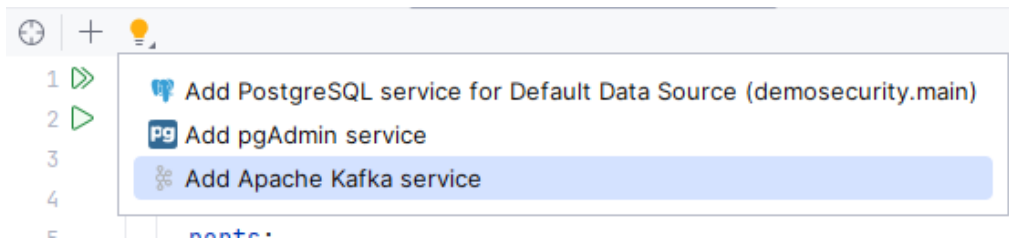
```
spring.kafka.bootstrap-servers=localhost:9092
```

В Amplicode Explorer становятся доступными категории:

- Kafka в Configurations со ссылкой на созданный класс и файл application.properties
- Kafka с доступными consumers и producers



Если в проекте уже есть файл docker-compose.yml, то с помощью подсказки (желтая лампочка) можно добавить Apache Kafka service



При клике на Add Apache Kafka service открывается диалоговое окно с автоматически заполненными полями

Add Apache Kafka to Docker Compose
✕

Service name:

Image: ↻ ▼

Depends on: + ↗ Used in: + ↗

Kafka Settings

Synchronized by: KRaft Zookeeper ?

Listeners

Listener	Host	Port	External port
PLAINTEXT	kafka	29092	29092
PLAINTEXT_HOST	localhost	9092	9092
CONTROLLER	kafka	9093	0

Persist data between re-deployments
The data will be stored on host machine folder

▼ **Advanced**

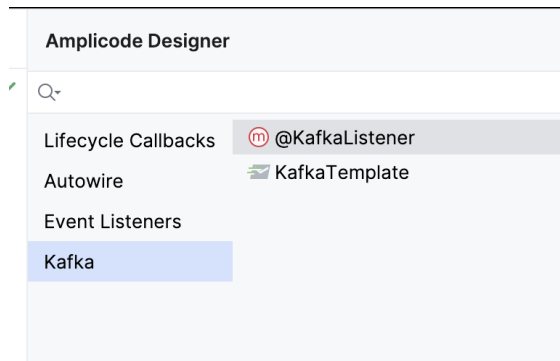
Restart policy: No Always On-failure Unless-stopped

Enable healthcheck

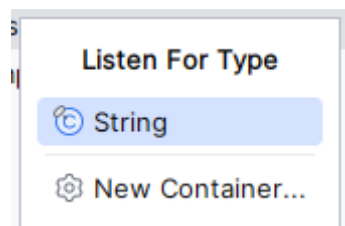
Создание через палитру

Механизм получения сообщения Spring для Kafka реализуется через слушателя. И Amplicode предоставляет такую возможность через Designer.

С помощью плагина Amplicode легко добавить листенер HibernateEventListener через Amplicode Explorer. На его примере можно посмотреть действия Kafka в Amplicode Designer.



При выборе `KafkaListener` в выпадающем списке есть варианты добавить `String` или `New Container`



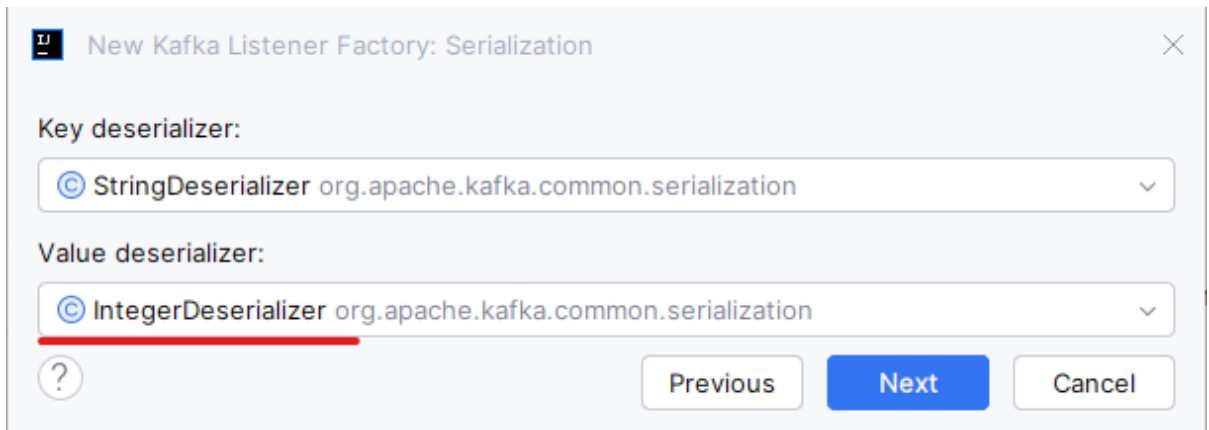
При выборе типа `String` в код добавляется метод с аннотацией `@KafkaListener` и дефолтный названием топика `"topic"`

```
@KafkaListener(topics = "topic", containerFactory = "stringListenerFactory")
public void consumeString(String string) {
}
```

При выборе `New Container` открывается диалоговое окно, где значение `Key` по умолчанию `String`, а значение `Value` необходимо заполнить



При заполнении `Value type` и нажатии `Next` в следующем окне `Value deserializer` автоматически проставляется, например `Value type = Integer`, `Value deserializer = IntegerDeserializer`



В последнем окне необходимо выбрать или создать конфигурационный класс и нажать Create. После выполненных действий в текущем классе также создается метод с аннотацией `@KafkaListener` и дефолтный названием топика “topic”

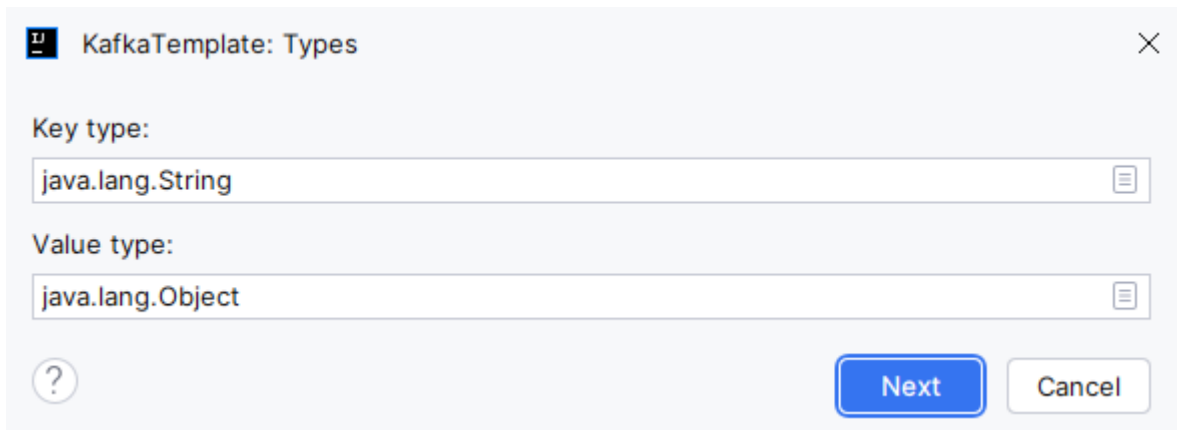
```
@KafkaListener(topics = "topic", containerFactory = "integerListenerFactory")
public void consumeInteger(Integer integer) {
}
```

А в конфигурационном классе добавляется примерно следующий код:

```
@Bean
public ConsumerFactory<String, Integer> integerConsumerFactory(KafkaProperties
kafkaProperties) {
    Map<String, Object> props = kafkaProperties.buildConsumerProperties(null);
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
IntegerDeserializer.class);
    return new DefaultKafkaConsumerFactory<>(props);
}
```

```
@Bean
public KafkaListenerContainerFactory<?>
integerListenerFactory(ConsumerFactory<String, Integer> integerConsumerFactory) {
    ConcurrentKafkaListenerContainerFactory<String, Integer> factory = new
ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(integerConsumerFactory);
    factory.setBatchListener(false);
    return factory;
}
```

При выборе `KafkaTemplate` открывается диалоговое окно “`KafkaTemplate: Types`”



После создания KafkaTemplate с дефолтными значениями добавляется следующий код в текущий класс:

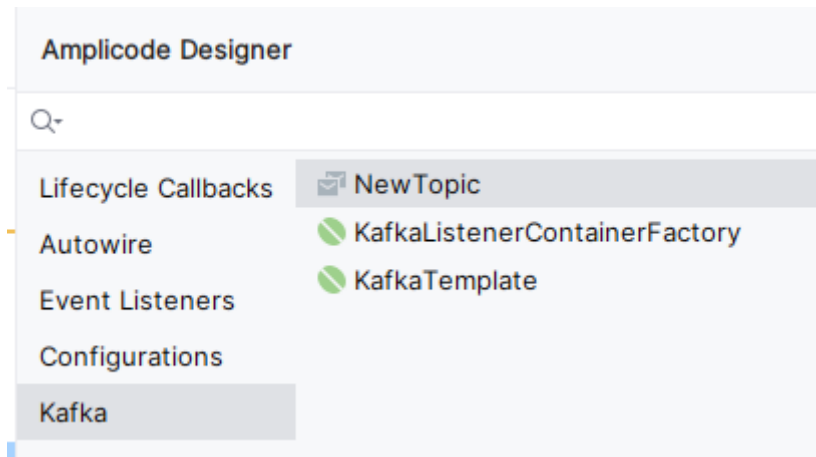
```
@Autowired
public void setKafkaTemplate(KafkaTemplate<String, Object> kafkaTemplate) {
    this.kafkaTemplate = kafkaTemplate;
}
```

А в конфигурационном классе добавляется примерно следующий код:

```
@Bean
DefaultKafkaProducerFactory<String, Object> objectProducerFactory(KafkaProperties
properties) {
    Map<String, Object> producerProperties = properties.buildProducerProperties(null);
    producerProperties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
    producerProperties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
    return new DefaultKafkaProducerFactory<>(producerProperties);
}
```

```
@Bean
KafkaTemplate<String, Object>
objectKafkaTemplate(DefaultKafkaProducerFactory<String, Object> objectProducerFactory)
{
    return new KafkaTemplate<>(objectProducerFactory);
}
```

В конфигурационных классах через Amplicode Designer есть возможность добавить NewTopic, KafkaListenerContainerFactory и KafkaTemplate.



При выборе NewTopic в класс добавляется новый метод с наименованием topic. После ввода названия топика в методе TopicBuilder.name(“”) он подставляется в название метода:

```

@Bean new *
NewTopic newTopic() {
    return TopicBuilder.name("new")
        .partitions(partitionCount: 1)
        .replicas(replicaCount: 1)
        .build();
}

```

При выборе KafkaListenerContainerFactory открывается диалоговое окно “New Kafka Listener Factory: Serialization”



После создания KafkaListenerContainerFactory с дефолтными значениями добавляется следующий код в текущий класс:

```

@Bean
public ConsumerFactory<String, String> stringConsumerFactory(KafkaProperties
kafkaProperties) {

```

```

Map<String, Object> props = kafkaProperties.buildConsumerProperties(null);
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
return new DefaultKafkaConsumerFactory<>(props);
}

```

@Bean

```

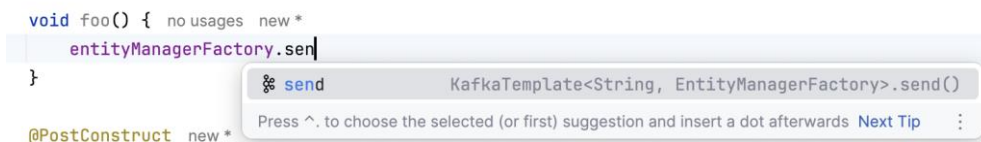
public KafkaListenerContainerFactory<?> stringListenerFactory(ConsumerFactory<String,
String> stringConsumerFactory1) {
ConcurrentKafkaListenerContainerFactory<String, String> factory = new
ConcurrentKafkaListenerContainerFactory<>();
factory.setConsumerFactory(stringConsumerFactory1);
factory.setBatchListener(false);
return factory;
}

```

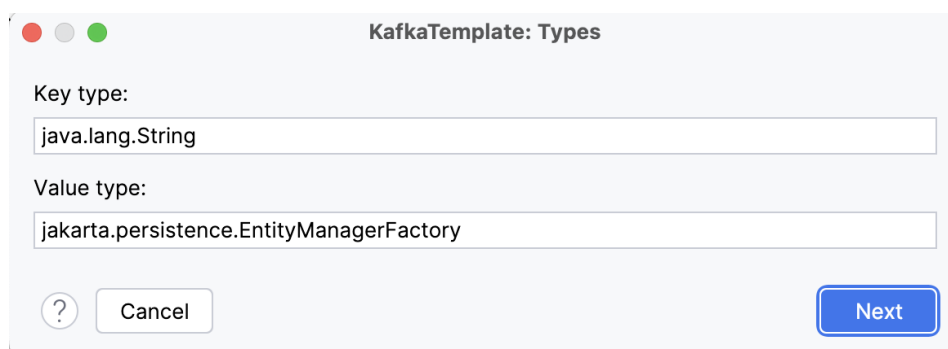
Постфикс send

Amplicode позволяет добавлять метод sendTo через умное автодополнение для того, чтобы отправлять сообщения в очередь.

Вернувшись в ранее созданный HibernateEventListener, посмотрим, как выглядит данная функциональность. Необходимо ввести в любом методе название сущности с маленькой буквы и начать писать send.



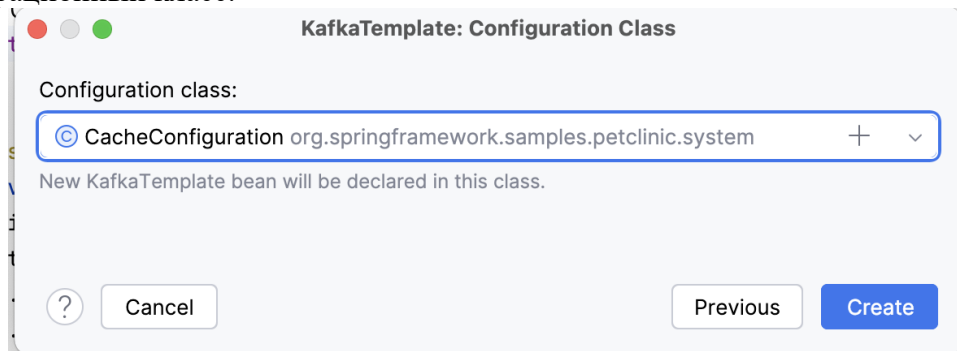
Как видно, Amplicode сам предоставит возможность выбрать метод send. Нажав Enter, появляется диалоговое окно “KafkaTemplate: Types”, где в Value type автоматически выбрана сущность.



После нажатия Next в следующем окне можно оставить сериализаторы для ключа и значения оставить по умолчанию.



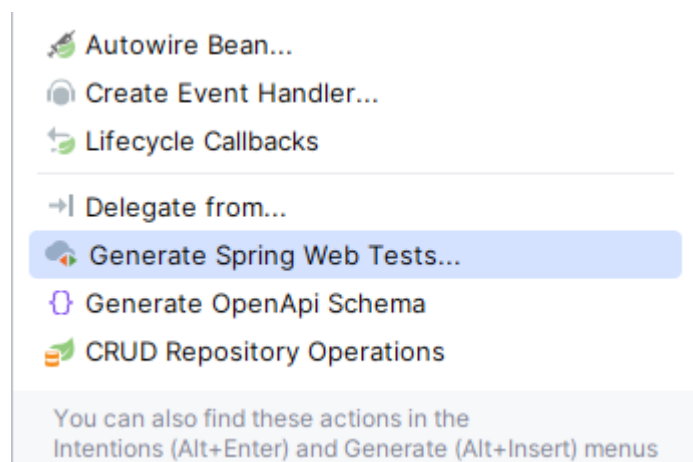
Нажав еще раз Next, открывается последнее окно, где необходимо выбрать конфигурационный класс.



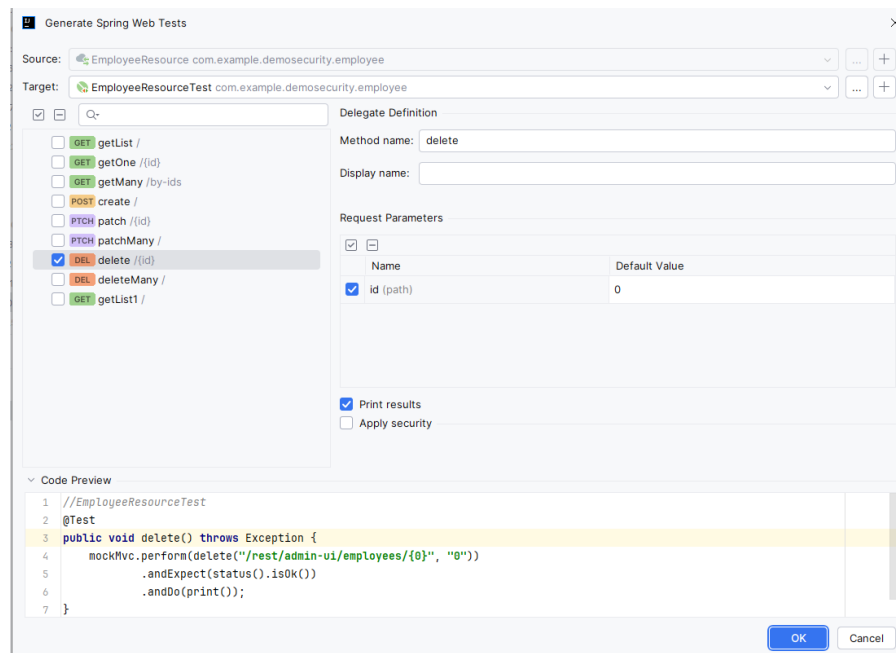
Тесты

Существует много способов протестировать работоспособность кода, а именно эндпоинтов. Можно обращаться к ним через стороннего HTTP клиента вроде Postman, через браузер или через curl команды прямо из терминала. Но Amplicode дает возможность реализовать проверку наиболее быстрым способом. А именно, используя @WebMvcTest.

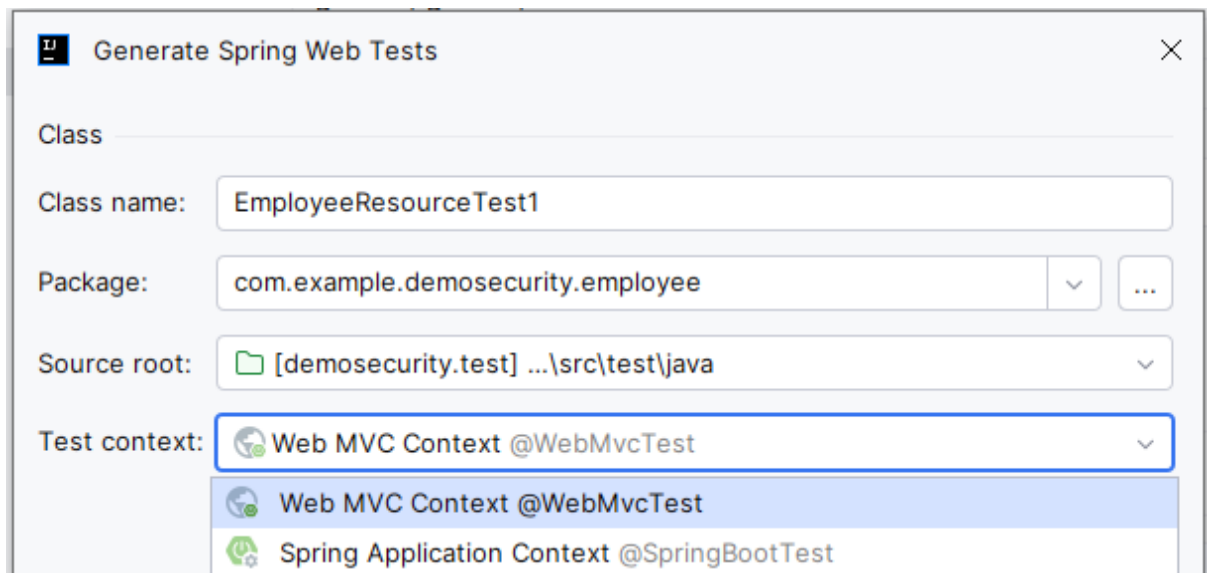
Чтобы сгенерировать тесты, необходимо в любом контроллере с аннотацией @Controller или @RestController вызвать Show Bean Actions, либо поставить курсор на необходимый метод и вызвать действие через Generate меню



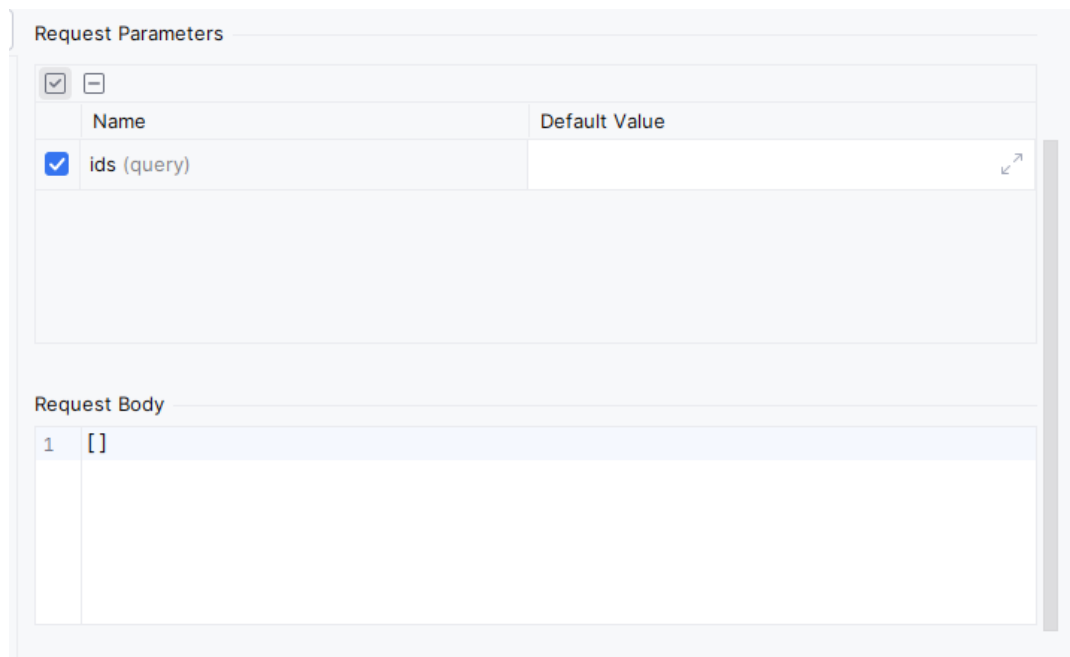
В появившемся окне нужно нажать Generate Spring Web Tests. После выполнения действия появляется диалоговое окно “Generate Spring Web Tests” с выбранным методом, если ранее действие вызывалось через Generate меню.



При нажатии Create Web Test (+) в поле Target есть возможность создать тестовый класс с изменением Test context в выпадающем списке: Web MVC Context или Spring Application Context. В Source root по умолчанию выбрана тестовая директория. Для создания класса просто нажмите ОК после выбора тестового контекста.

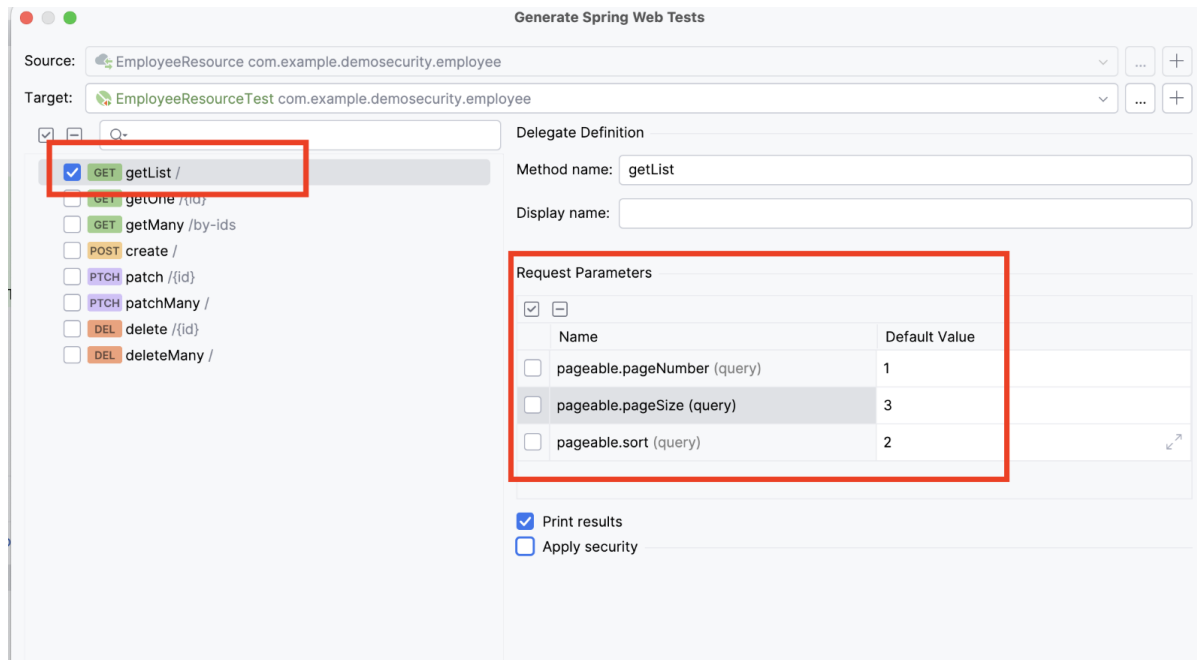


В зависимости от передаваемых параметров у метода контроллера будут отображаться Request Parameteres и Request Body в диалоговом окне.



В диалоговом окне поля Method name, Display name, Default value у Request Parameters и Request Body являются редактируемыми.

Пометив метод, как тестируемый, можно приступить к настройке входных параметров для метода. На примере метода getList, возвращающего всех сотрудников, посмотрим как это выглядит на картинке:



Опция Apply Security позволяет настроить доступ, где вводятся имя пользователя, пароль, роль. Также, есть опция CSRF, которую можно выбрать. При необходимости все поля можно заполнить:

Print results
 Apply security
 Anonymous access

User name: Password:

Roles: + ↗
Comma separated list

Apply CSRF ?

The following libraries will be added to the project: org.springframework.security:spring-security-test

Code preview отобразит актуальные изменения диалогового окна:

Print results
 Apply security
 Anonymous access

User name: Password:

Roles: + ↗
Comma separated list

Apply CSRF ?

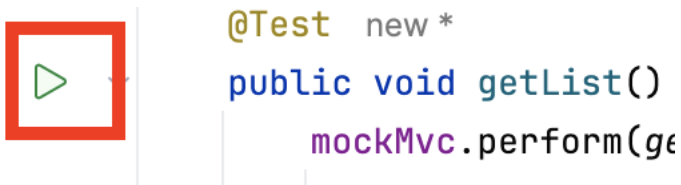
The following libraries will be added to the project: org.springframework.security:spring-security-test

После нажатия ОК метод добавится в тестовый класс:

```

6   @WebMvcTest({EmployeeResource.class}) new *
7   public class EmployeeResourceTest {
8
9       @Autowired
10      private MockMvc mockMvc;
11
12
13      @Test new *
14      public void getList() throws Exception {
15          mockMvc.perform(get( urlTemplate: "/rest/admin-ui/employees")
16                          .with(SecurityMockMvcRequestPostProcessors.csrf())
17                          .with(SecurityMockMvcRequestPostProcessors
18                              .user( username: "user")
19                              .password("user")
20                              .roles("USER")))
21              .andExpect(status().isOk())
22              .andDo(print());
23      }
24  }
  
```

Чтобы запустить тесты, воспользуйтесь специальной иконкой в исходном коде тестового класса:



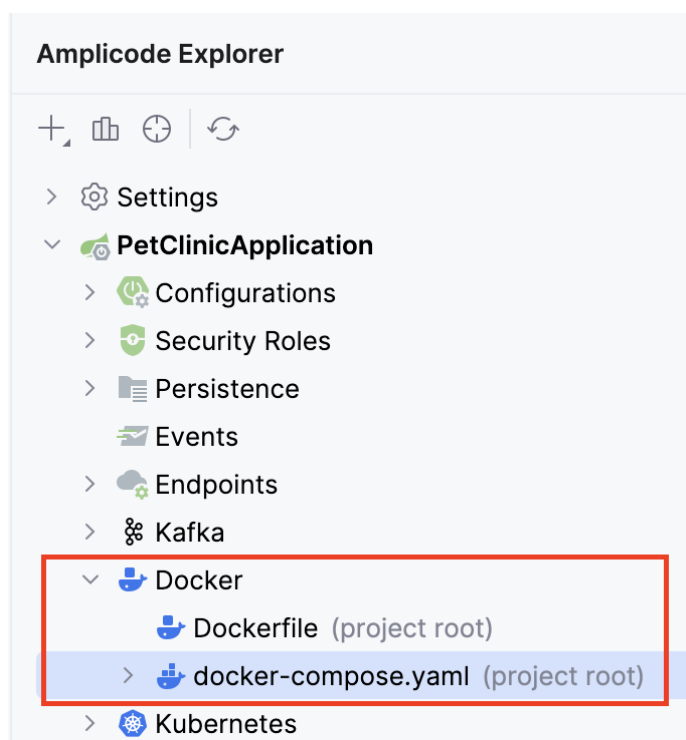
Если метод написан правильно, то тест прогонится успешно.

Docker

Amplicode предоставляет помощь с технологией контейнеризации Docker при работе со Spring Boot приложением, которая включает в себя:

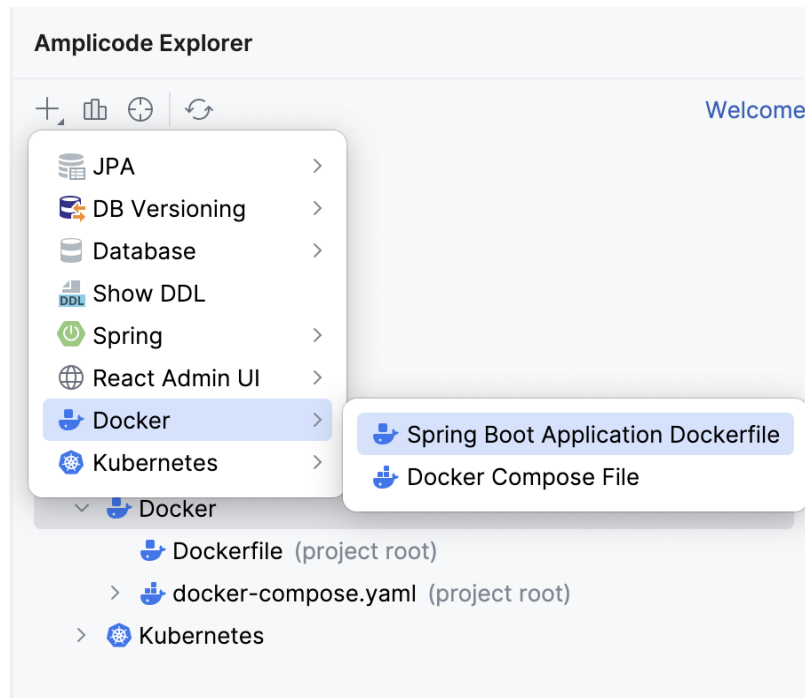
- Создание Docker файла для сборки приложения Spring Boot в Docker image.
- Помощь при создании docker-compose файлов и добавления связанных сервисов.

Дерево компонентов Amplicode Explorer содержит элемент Docker, в котором сгруппированы Docker/Docker Compose файлы проекта, с возможностью быстрой навигации к файлам, а также выполнения дополнительных действий, например сборки Docker Image.



Создание Dockerfile

Amplicode позволяет создать правильный Docker файл для Spring Boot приложения. Для создания Docker файла воспользуйтесь действием «Плюс» в Amplicode Explorer и выберите Docker -> Spring Boot Application Docker File.



В открывшемся диалоговом окне возможно указать:

- Имя Docker файла
- Директорию, в которой будет расположен Docker файл
- Базовый Docker Image, который будет использоваться при создании Docker Image для приложения
- Возможность использовать Spring layers для ускорения создания Docker Image
- Включение шага сборки Spring Boot приложения перед созданием Docker Image

New Spring Boot Application Dockerfile

Application:

Name:

Directory:

Image Settings

Base image:

Use Spring Boot JAR layers

Run as non-root user Username:

Include application build stage

Gradle build settings

Distribution: Wrapper Docker image

Builder image:

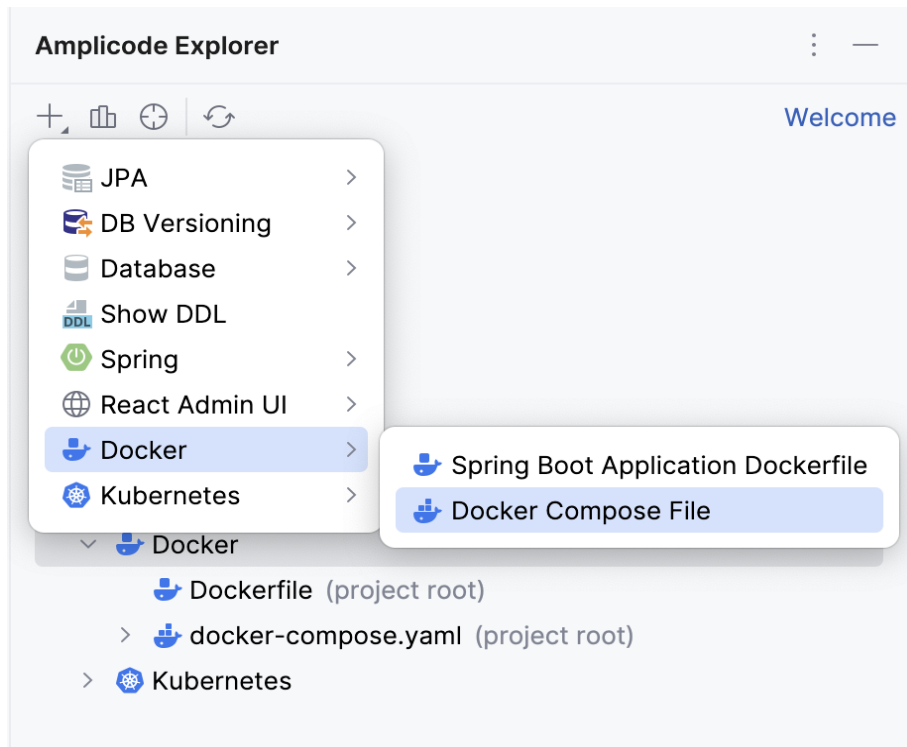
Build command:

Working directory:

По нажатию кнопки ОК создается Dockerfile в выбранной директории. В дальнейшем для сборки Docker Image, в дереве Amplicode Explorer для выделенного Docker файла, воспользуйтесь действием Build из контекстного меню. Параметры диалога сборки Docker файла, позволяют указать правильный Image Tag.

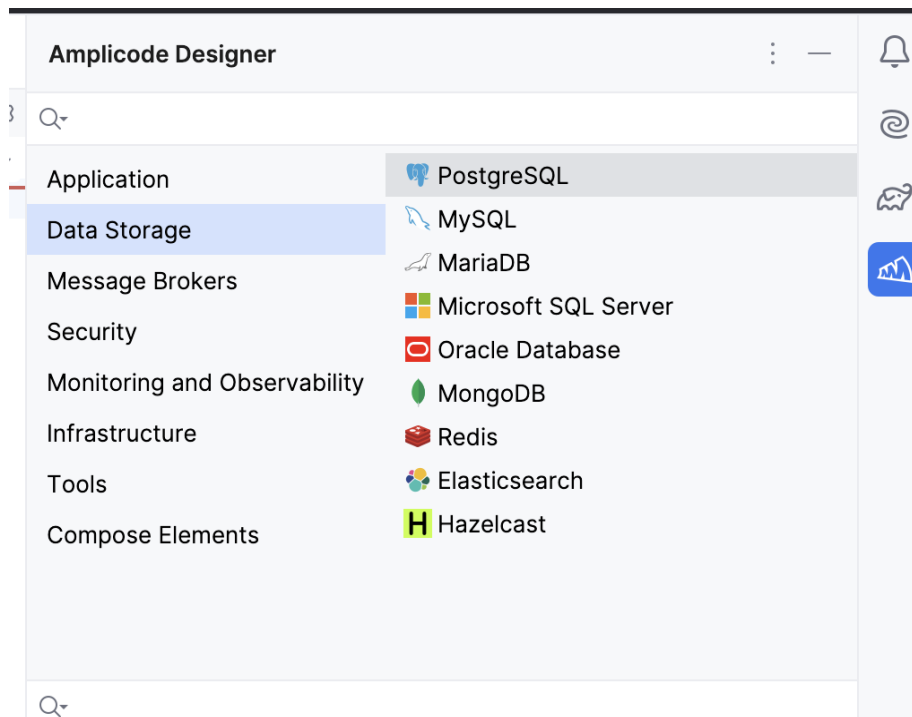
Создание Docker Compose файла

Amplicode позволяет создать Docker Compose файл и упростить добавление в него сервисов, связанных со Spring Boot приложением. Для создания Docker Compose файла воспользуйтесь действием «Плюс» в Amplicode Explorer и выберите Docker -> Docker Compose File.

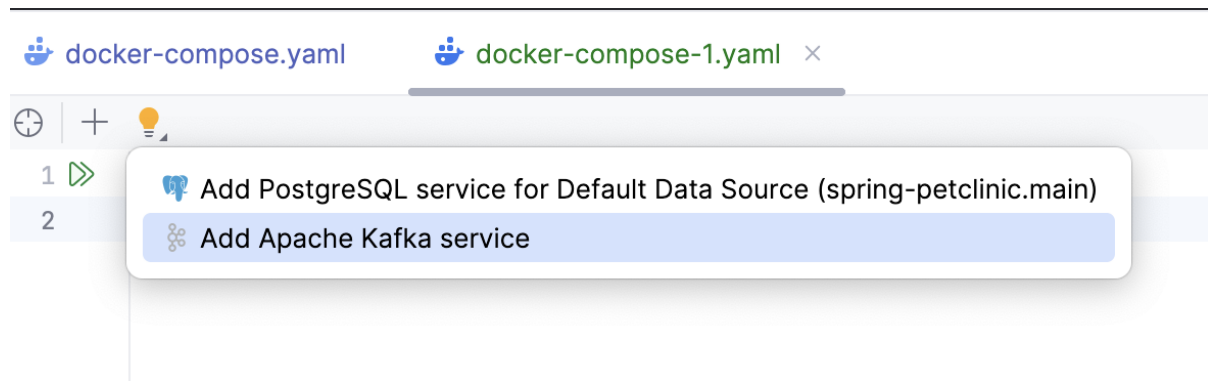


В открывшемся диалоговом окне укажите имя файла, директорию размещения и нажмите ОК.

В открывшемся файле доступна палитра компонентов с возможностью добавлять различные сервисы, такие как базы данных, очереди сообщений, системы мониторинга, сервис для Spring Boot приложения и прочие. При добавлении сервисов отображаются диалоговые окна для редактирования основных параметров сервиса.



В дополнение к выше сказанному, анализирует структуру и зависимости проекта и делаются предложения по сервисам, которые стоит использовать. Например в проекте используется БД Postgres или Kafka и предлагается добавить БД Postgres и Kafka в виде сервисов Docker. Список предложений по добавлению сервисов Docker отображается в виде “лампочки”.



При выделении любой строки Docker сервиса в файле отображается инспектор свойств для редактирования основных параметров сервиса.

postgres		PostgreSQL
Q-		
General		
Image	postgres:16.4	
Restart policy	No	
Build	<input type="checkbox"/>	
Depends on	Edit	
Profiles		
Env files		
PostgreSQL settings		
Database name	spring-petclinic	
User	root	
Password	root	
Volumes		
Data		
Container path	/var/lib/postgresql/data	
Source	postgres_data	
Read-only access	<input type="checkbox"/>	
Init scripts		
Container path	/docker-entrypoint-initdb.d	
Source		
Published ports		
Postgres (5432) <input checked="" type="checkbox"/>		
Host port	5432	
Host IP	0.0.0.0	
Protocol	tcp	
Networking		
Mode	Use Docker networking	
Connected networks	Edit	

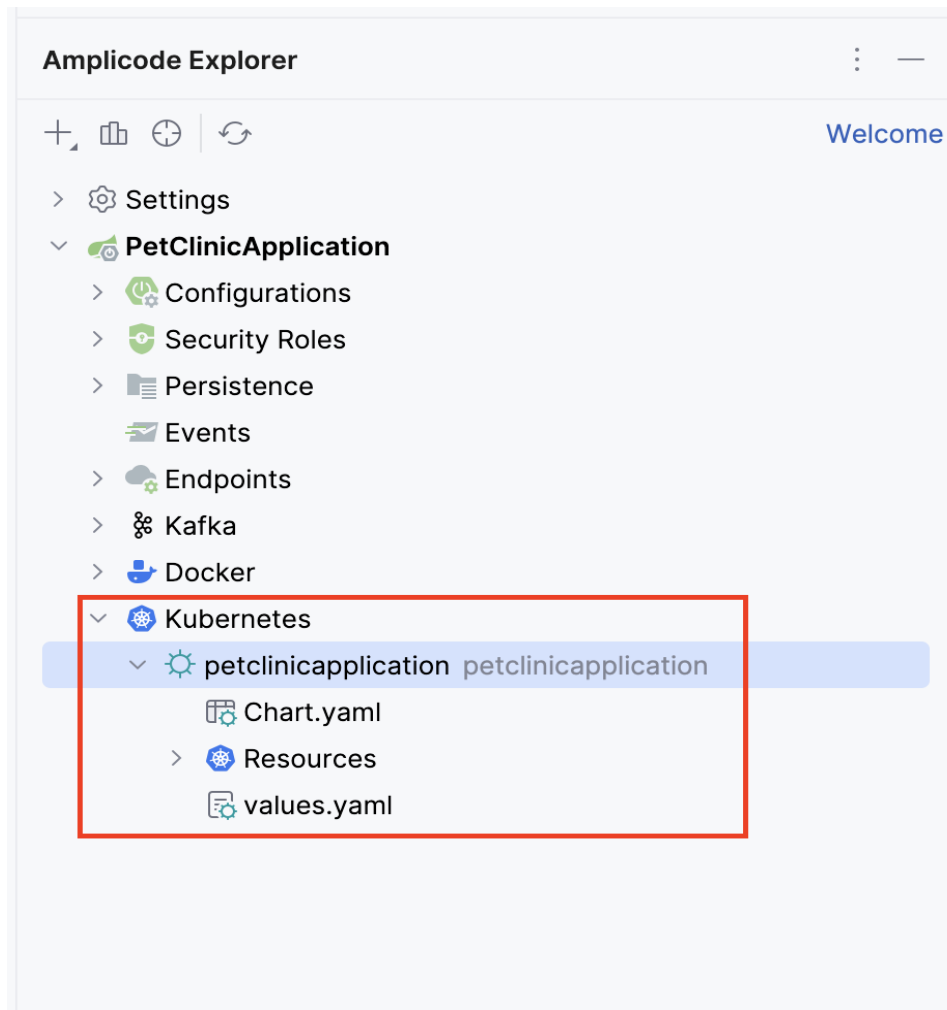
После того как Docker Compose файл создан и настроен необходимо развернуть Docker сервисы. Запуск Docker Compose выполняется с использованием действия “play”.

```
1 >> services:
2   postgres:
3     image: postgres:16.4
4     restart: "no"
5     ports:
6     - "5432:5432"
7     volumes:
8     - postgres_data:/var/lib/postgresql/data
9     environment:
10    POSTGRES_USER: root
11    POSTGRES_PASSWORD: root
12    POSTGRES_DB: spring-petclinic
```

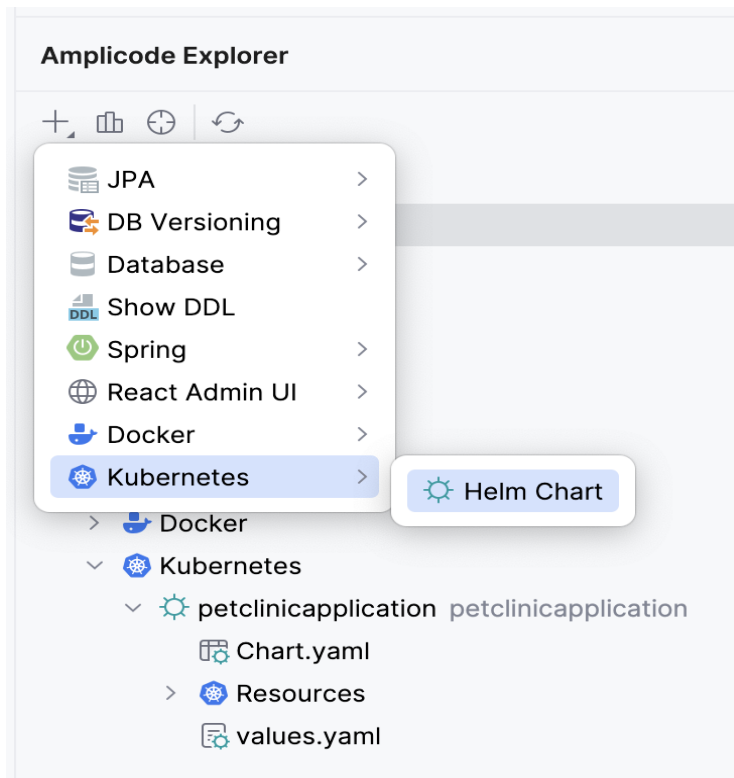
Kubernetes и Helm чарты

Amplicode предоставляет помощь с технологией Kubernetes и Helm при работе со Spring Boot приложением.

Дерево компонентов Amplicode Explorer содержит элемент Kubernetes, в котором сгруппированы Helm чарты проекта, с возможностью просмотра структуры Helm чарта и быстрой навигации к файлам, а также выполнения дополнительных действий, например деплоя чарта в Kubernetes кластер.



Для создания Helm чарта воспользуйтесь действием «Плюс» в Amplicode Explorer и выберите Kubernetes -> Helm Chart.



При создании Helm чарта предложено будет указать имя чарта, директорию расположения и шаблон, по которому будет создан чарт. Для создания Helm чарта для Spring Boot приложения выбираем шаблон Spring Boot, а для произвольного чарта выбираем шаблон Default. В случае Spring Boot чарта можем указать дополнительные параметры такие как:

- Имя Docker образа
- Порт, который будет использовать Spring Boot приложение
- Пробы Kubernetes для проверки работоспособности Pod
- Включать Ingress для приложения и его параметры.

New Helm Chart

Name:

Directory:

Template:

Settings

Application:

Image name:

Server port:

Probes

Readiness probe path:

Liveness probe path:

Enable ingress

Host:

Context path:

По нажатию ОК будет создан Helm чарт.

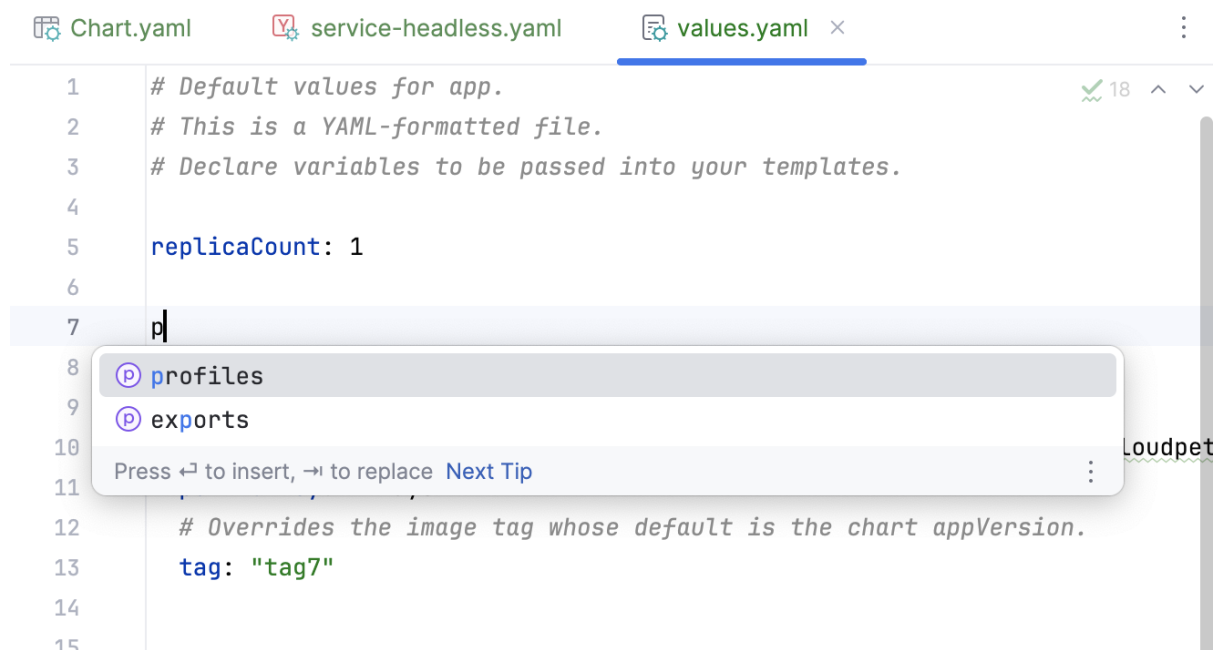
После создания чарта, возможно потребуется дополнительная настройка. Для этого в чарте в файле Chart.yaml доступна палитра инструментов Amplicode с возможностью добавления дополнительных сервисов, таких как базы данных или очереди сообщений в виде зависимостей.

```

1  apiVersion: v2
2  name: petclinicapplication
3  description: A Helm chart for deploying spring-petclinic to Kubernetes
4
5  # A chart can be either an 'application' or a 'library' chart.
6  #
7  # Application charts are a collection of templates that can be packaged into
8  # to be deployed.
9  #
10 # Library charts provide useful utilities or functions for the chart develop
11 # a dependency of application charts to inject those utilities and functions
12 # pipeline. Library charts do not define any templates and therefore cannot
13 type: application

```

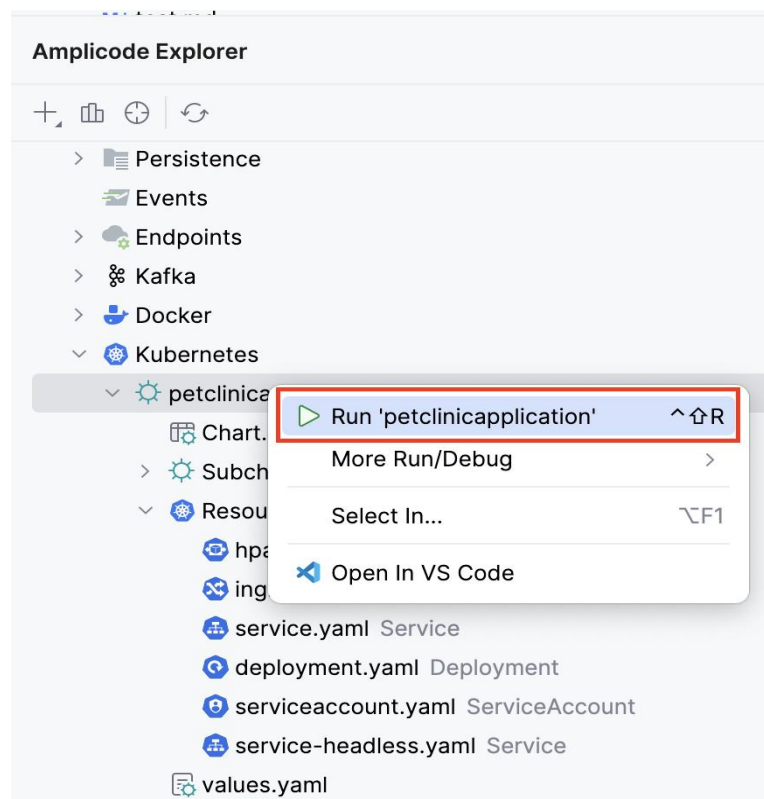
Так же всех файлов чарта доступна подсветка кода, включая дескрипторы Kubernetes и в файле values.yaml доступна подсказка имен переменных используемых в Helm чарте для эффективного заполнения значений переменных.



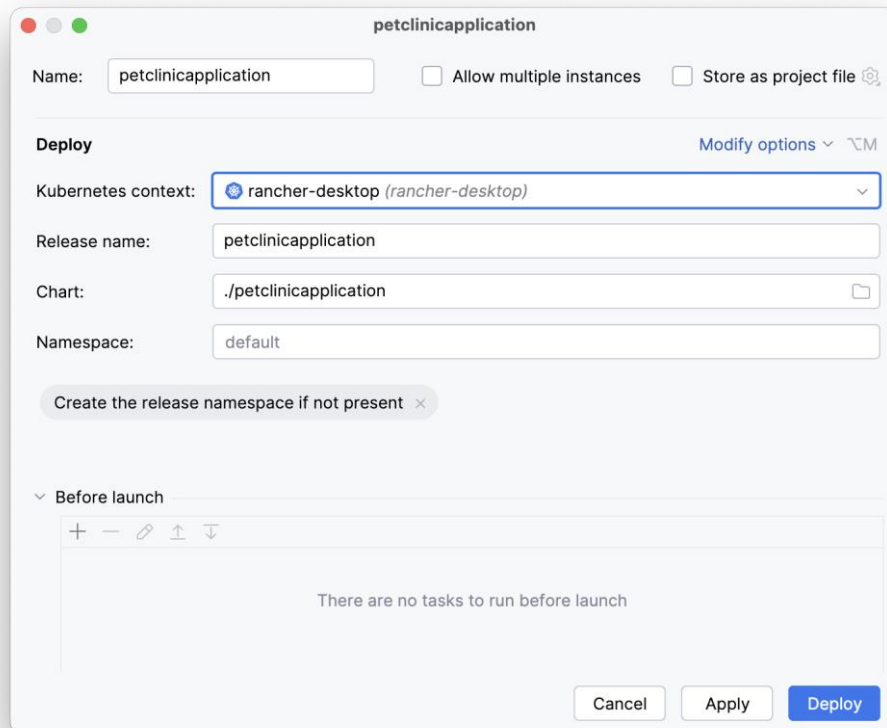
```
1 # Default values for app.
2 # This is a YAML-formatted file.
3 # Declare variables to be passed into your templates.
4
5 replicaCount: 1
6
7 pl
8
9
10
11
12 # Overrides the image tag whose default is the chart appVersion.
13 tag: "tag7"
14
15
```

The screenshot shows a code editor with three tabs: Chart.yaml, service-headless.yaml, and values.yaml. The values.yaml file is active, showing a list of default values. A dropdown menu is open at line 7, suggesting variables like 'profiles' and 'exports'. A tip box is also visible, stating 'Press ↵ to insert, → to replace Next Tip'.

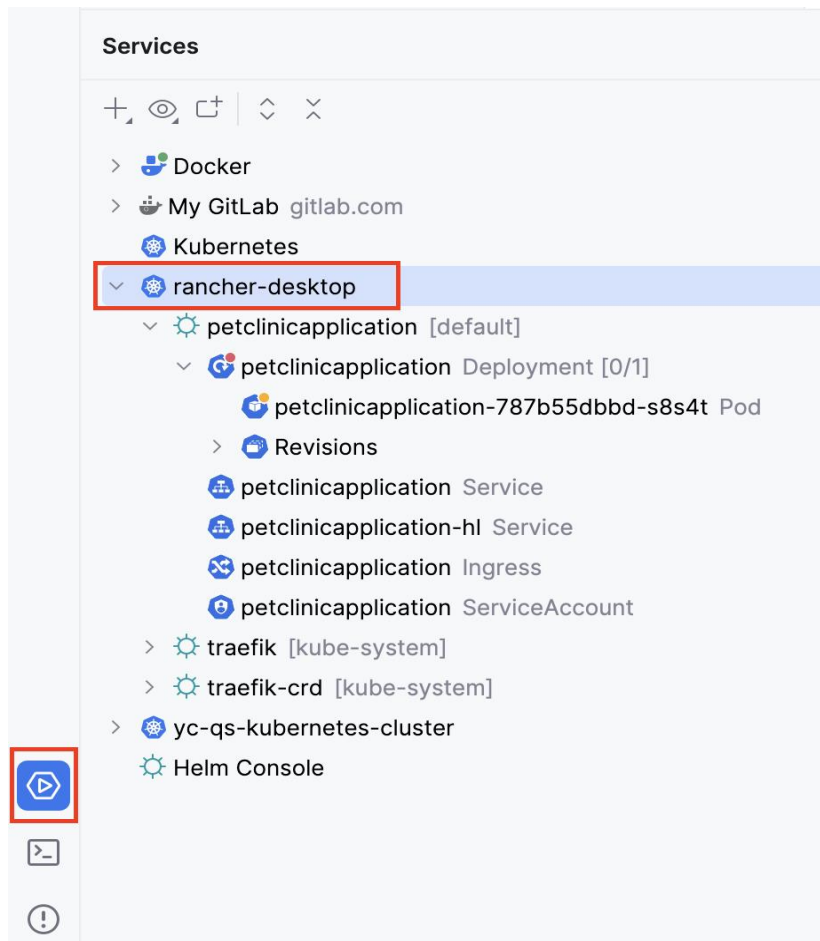
После окончательной настройки Helm чарта Amplicode позволяет его развернуть в Kubernetes кластере. Для этого в дереве Amplicode Explorer находим Helm чарт и по контекстному меню вызываем команду Run “имя чарта”.



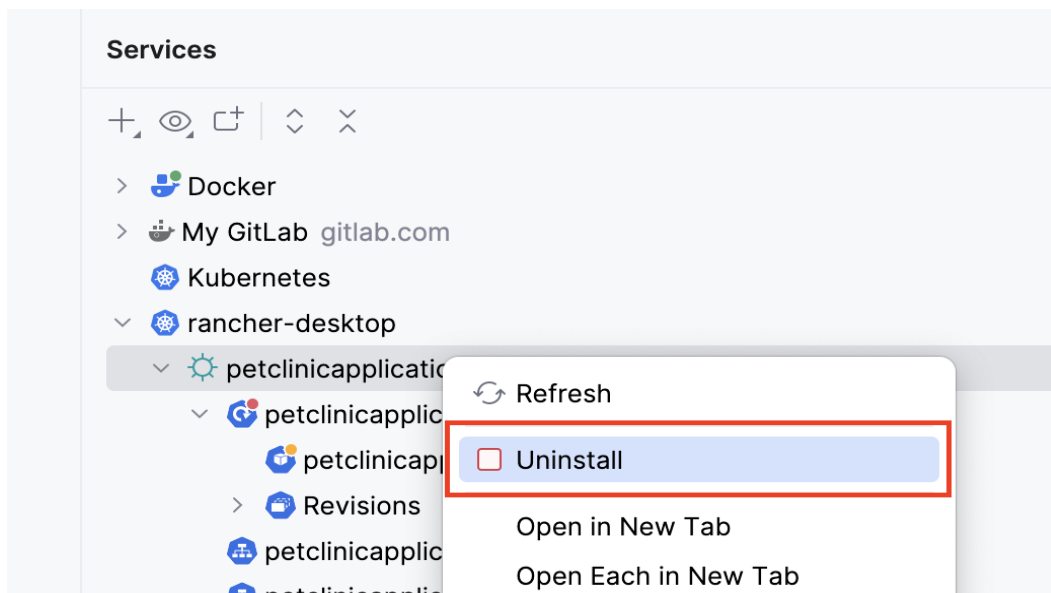
В открывшемся диалоговом окне выбираем Kubernetes Context и нажимаем Deploy. Приложение будет развернуто в Kubernetes кластере, определяемом переменной Kubernetes Context.



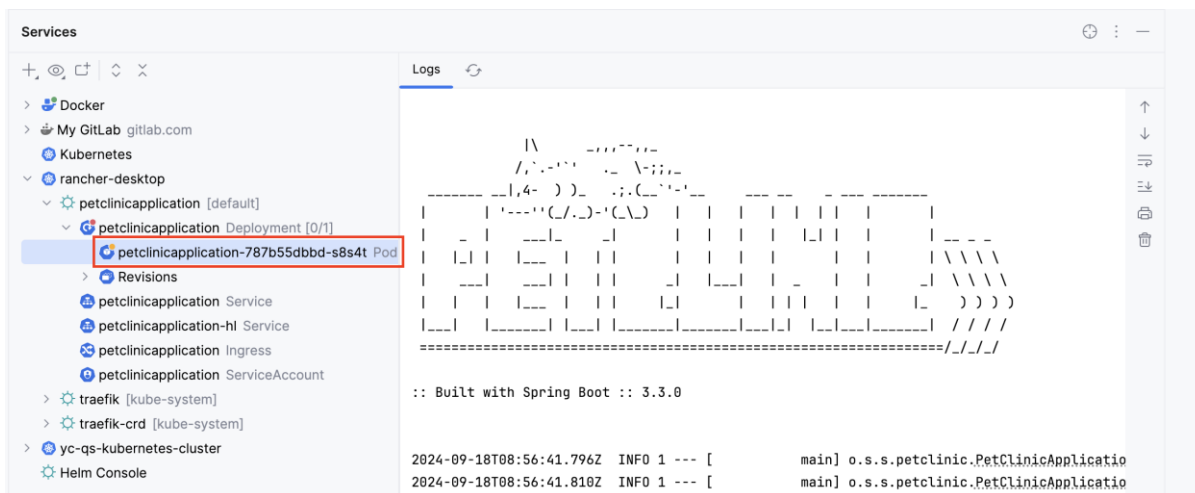
Amplіcode на вкладке Services отображает Kubernetes кластеры (контексты) и позволяет просмотреть список запущенных приложений.



Запущенное приложение можно удалить из Kubernetes кластера, воспользовавшись действием Uninstall в контекстном меню.



Для рабочих нагрузок Pod доступен просмотр логов, для этого необходимо выделить Pod и справа для него отобразятся логи.

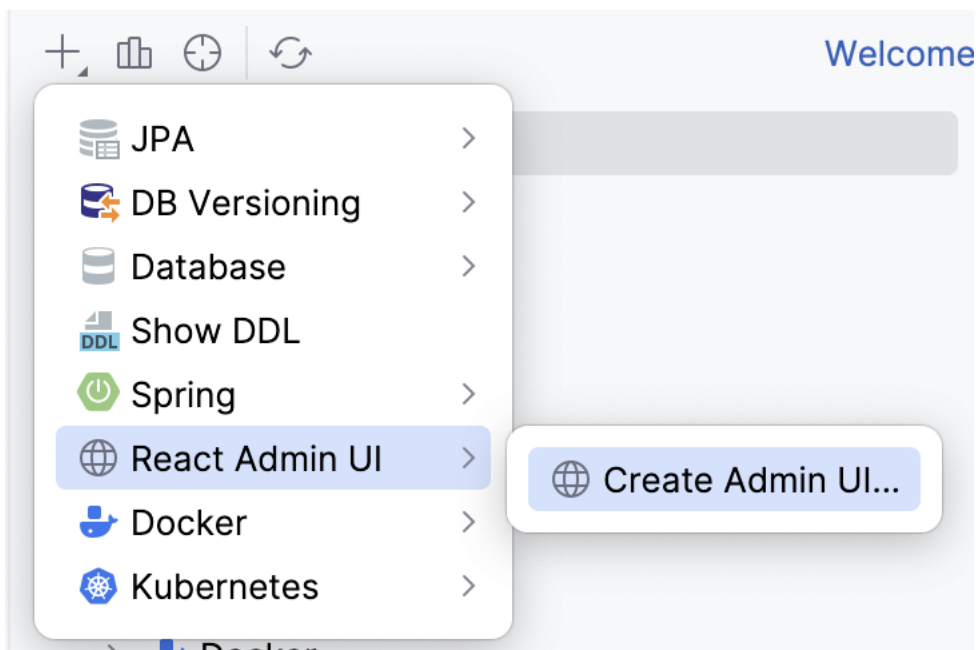


Создание React Admin модуля

Amplicode позволяет создавать административный пользовательский интерфейс на React Admin для взаимодействия с фронтендом.

Перед созданием модуля на ПК следует установить Node.js и npm.

Для создания админки необходимо в Amplicode Explorer вызвать действие Create (+) и выбрать опцию React Admin UI – Create Admin UI



После вызова действия Create Admin UI запускается процесс проверки версии npm (npm check). После завершения проверки открывается диалоговое окно для создания админ модуля.

Create React Admin UI

Spring Boot Application:

Project name:

Backend host:

API root path:

Add react admin starter
Add [Spring Utils Starter](#) to simplify work with Admin UI

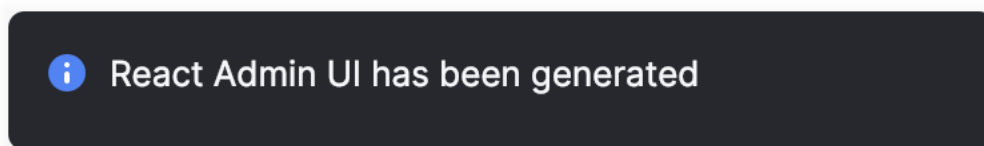
Npx executable:

The npm version must be greater than 10.*.* and the node version must be greater than 18.*.*

В диалоговом окне находятся следующие поля:

- Выпадающий список Spring Boot Application (Предвыбран текущий back-end)
- Поле ввода Project name (Предзаполнено названием проекта)
- Поле Backend host (Предзаполнено дефолтным localhost:8080)
- Поле API root path (Предзаполнено как /rest/admin-ui)
- Чекбокс Add react admin starter. Опция добавляет библиотеку, которая предоставляет несколько полезных утилит, помогающих реализовать REST-контроллеры для интерфейса на основе React Admin.
- Поле Npx executable (Предзаполнено путем к версии npx, так же можно выбрать произвольную версию)

После нажатия ОК генерируется реакт админ модуль и появляется уведомление “React Admin UI has been generated”.



Открывается README.md файл с подсказками от Amplicode.

pet-clinic

Admin UI application powered by ReactAdmin and created by Amplicode.

Next steps

Install the application dependencies:

```
npm install
```

Install Amplicode Frontend VS Code extension

[Installation instruction](#)

Or install via terminal command:

Only Amplicode Frontend

```
code --install-extension haulmont-tech-ltd.amplicode-frontend
```

Amplicode Fullstack (best for fullstack developers)











```
code --install-extension haulmont-tech-ltd.amplicode-fullstack
```

Create backend logic

You can create special Spring CRUD Controllers, that are perfectly fit to [project dataProvider](#). Each controller corresponds to Resource in ReactAdmin terms.

Amplicode Explorer

В корне проекта создается папка webapp со всеми необходимыми файлами для работы с фронтендом.

- ▼  webapp
 - >  .vscode
 - >  public
 - ▼  src
 - >  themes
 - ≡ App.tsx
 - ≡ dataProvider.ts
 - ≡ index.tsx
 - ≡ springDataProvider.ts
 - ≡ vite-env.d.ts
 - ≡ .env
 -  .eslintrc.json
 -  .gitignore
 -  .prettierrc.json
 - <> index.html
 -  package.json
 - M↓ README.md
 -  tsconfig.json
 - ≡ vite.config.ts