ASTRAVERGE / PGDEVELOPER

Генерация кода

AstraVerge PGDeveloper 0.7.0

Практическое руководство · 11 ноября 2026

Рабочая среда для PostgreSQL: от структуры базы до кода приложения.

Команды и имена файлов сохраняют техническое имя av-pgdeveloper. Параметры и границы применимости относятся к версии 0.7.0.

Содержание

Интерфейс v2 и файлы 3

Новые языки 4

Новые контракты 6

Protobuf: эволюция 6

Реестр и проверки v2 7

Использование 7

Типы и NULL 8

Код доступа 9

Ограничения и расширение 10

Воспроизведение проверок 11

Версия 2 расширяет существующую подсистему до **15 генераторов**: три генератора v1 и все 12 новых. MetadataExtractor и архитектура SQL Window / Object Browser / Result Grid сохранены. Генерация остаётся локальным детерминированным Java-кодом, без AI, внешних API и запуска целевых runtime из приложения.

Интерфейс v2 и файлы

В том же Generate Code Dialog выберите группу **Languages** или **Contracts**. Список Output содержит только результаты, поддерживаемые выбранным генератором. Контракты и TypeScript не предлагают CRUD. Поле Artifact выбирает файл для Preview, Copy и Open in Editor. Save для набора показывает список файлов и выбирает каталог; существующие файлы перечисляются перед явным подтверждением замены.

`GeneratedCode` сохраняет прежний четырёхаргументный конструктор и основной файл, а дополнительные артефакты доступны через `files()` / `additionalFiles()`. Файлы имеют безопасные плоские имена, без traversal, абсолютных путей и Windows device names. Экспорт проверяет весь список, запрещает symlink-назначения и записывает каждый файл через временный файл. Общая атомарная транзакция нескольких файлов не обещается: при файловой ошибке проверьте сообщение и сохранённый список.

Новые языки

Генератор	Результаты и зависимости	Представление типов и NULL	Проверка	
TypeScript	interface / type, DTO; tsc, серверного CRUD нет	bigint/numeric - string; enum - строковые enum; nullable - `T \	null; `Optional properties` независимо добавляет `?`; UUID/date/time - текст с PG-типом в описании	`tsc --noEmit --strict --target ES2020`
C#	Record / Class, Mapping / CRUD; Npgsql, C# 11+	long/int/short/bool, Guid, DateOnly, DateTime Unspecified, DateTimeOffset; decimal только при известном precision ≤ 28 и scale 0-28, иначе точная строка; NRT и nullable value types	dotnet build и реальный Npgsql CRUD	
PHP	DTO, typed properties, Mapping / CRUD; PDO_PGSQL, PHP 8.2+ syntax	bigint/numeric/UUID/time/JSON/arrays - точная строка; int/bool/float для соответствующих PG-типов; `?T`	php -l и PDO-тест при наличии runtime	
C++17	Struct, Mapping / CRUD; **libpq**	int64/int32/int16/bool/float/double, std::string для точных numeric/UUID/time/JSON/arrays/composite, std::optional; RAII для PGconn/PGresult	g++ -std=c++17 и реальный libpq CRUD	
Rust	Struct, Mapping / CRUD; **tokio-postgres 0.7**, uuid 1	i64/i32/i16/bool/f32/f64, uuid::Uuid, Option; numeric/JSON/arrays/date/time/composite - точная строка с исходным PG-типом	cargo check и live test при наличии runtime	
R	Schema / Model и Row Mapping / Data Loading; **DBI + RPostgres**	data.frame из точных text-проекций, SQL NULL - NA_character_, JSON null - строка `null`; bigint/numeric не проходят через double	parse(encoding="UTF-8"), DBI-тест при наличии пакетов	
MATLAB	Schema / Model, Data Loading / Analysis; **Database Toolbox**, native postgresql	MATLAB table с точными string-колонками и missing; JSON null - строка `null`; отдельный SQL NULL flag для каждой колонки сохраняет отличие от пустой строки	установленный R2025a, matlab -batch, реальный SELECT / статистика / ошибки	

Все перечисленные функции работают с метаданными таблицы, представления или пустого SQL Result. Для C#/PHP/C++/Rust Row Mapping ожидает проекцию колонок как `::text` в исходном порядке: собственный PostgreSQL-парсер выполняет проверенные конверсии primitive и не получает уже

округлённый numeric от драйвера. CRUD сам создаёт эту проекцию. Для произвольного SQL Result такую проекцию задаёт вызывающий код; несовместимый native row не считается корректным входом этого reader.

Параметры C#/C++/Rust - `$1::text::schema.type`, PHP - `?::text::schema.type`. Значения передаются отдельно от SQL; типы и идентификаторы берутся из IM и экранируются. CRUD использует прежние правила PK, defaults/identity/generated, ONLY и RETURNING, не вызывает Commit/Rollback. Numeric NaN/Infinity, не помещающийся в System.Decimal, вызывает ошибку C# вместо округления; для этих данных используйте явную текстовую проекцию SQL Result.

C#/PHP/C++/Rust сохраняют неоднозначные массивы и composite точным PostgreSQL-текстом. TypeScript использует `PgArray<T>`: `dimensions`, `lowerBounds`, плоский `values: (T|null)[]` в row-major порядке. Это типизированный массив без потери нестандартных границ и NULL-элементов. JSON DTO использует `ExactJson {jsonText}`: SQL NULL - внешний null, JSON null - `jsonText: "null"`. Так JSON numeric не превращается неявно в небезопасный JavaScript number.

R не связан с AV-R/AV-Agent. Подключение открывается только вызовом сгенерированной функции `_connect`; `_load(connection, sql, params)` передаёт параметры DBI отдельно. Для дат/времени остаётся исходный PG-текст; as.Date / as.POSIXct вызываются явно с нужным timezone. `_analyze` предупреждает о намеренном преобразовании в double, возвращает матрицу и статистику, сохраняя исходный точный data.frame.

MATLAB создаёт основной `.m` и `_example.m`; вызов `api = SampleRow()` возвращает schema и функции connect/load/map/analyze. Подключение:

```
api = SampleRow();
conn = api.connect('127.0.0.1', port, databaseName, userName, password);
cleanup = onCleanup(@() close(conn));
data = api.load(conn);          % table точных строк, SQL NULL -> missing
stats = api.analyze(data, {'count','amount'}); % явный warning о double
```

Для SQL Result MATLAB map ожидает чередующиеся пары value `::text` / `IS NULL`. Даты можно явно преобразовать через datetime с InputFormat; timestamp без timezone и timestamptz с offset не считаются одним типом. AV-PGDeveloper не требует MATLAB или MATLAB Runtime для создания файлов. Для выполнения файлов требуется лицензия Database Toolbox. Конкретная фактическая приёмка R2025a приведена в отчёте.

Новые контракты

Формат	Правила
JSON Schema Draft 2020-12	`$schema`, title/type/properties/required/additionalProperties/description; все поля required независимо от SQL nullable; nullable - anyOf с null; exact numeric/bigint strings, UUID/date/date-time formats, локальный timestamp без ложного timezone; enum labels; массив с shape/bounds/values; `x-postgresql` сохраняет тип/OID/precision/scale/ordinal
OpenAPI 3.1	Полный документ с info, пустым paths и components.schemas; те же JSON Schema-правила; фиктивные endpoints и бизнес-примеры не создаются
Protocol Buffers proto3	Message, enum, optional presence, PgArray с repeated dimensions/bounds/PgElement; nullable-элемент имеет optional text; numeric/JSON/date/time/composite - exact text, bigint - int64; `.proto` и `.proto-map.json`
Apache Avro	Record, fields, enum с x-postgresql-labels, nullable union с default null; UUID/date/timestamp-micros/local-timestamp-micros; bounded numeric с допустимыми precision/scale - union decimal bytes / exact string, строковая ветка сохраняет specials; unbounded/negative scale - exact string; массив - record shape/bounds/values
GraphQL SDL	type или input, enum и scalar declarations, lists, `!` только для известного non-null; BigInt/Decimal - пользовательские точные scalars; ExactJSON - JSON text; локальный timestamp и instant раздельны; Query/Mutation/resolvers не создаются

Контракты содержат исходные имена и порядковые номера в описаниях даже при дедупликации идентификаторов. Enum-символы Proto/Avro/GraphQL имеют безопасные имена VALUE_N, а исходные PostgreSQL labels сохраняются в комментариях/описании/атрибутах. Composite остаётся текстом: достоверной структуры вложенных полей в текущем IM нет, поэтому фиктивные вложенные модели не создаются. Произвольные CHECK домена не интерпретируются как JSON/GraphQL-ограничения; достоверные base type и enum сохраняются.

GraphQL scalar declarations требуют codecs в целевом приложении. JSON/Avro/Proto контракты не являются готовым автоматическим сериализатором PostgreSQL: приложение должно соблюдать описанное переносимое представление. Особые значения времени infinity/BC/out-of-range не преобразуются в несуществующее нативное представление: выберите точный текст или отклоняйте их в целевом serializer. Avro logical date/time не заявляет представление PostgreSQL infinity.

Protobuf: эволюция

Начальные номера вычисляются по SHA-256 исходного ключа колонки (имя и номер повторения), а не по позиции. Перестановка уникальных колонок не меняет номера даже без mapping. Диапазон 19000-19999 исключён; коллизия вызывает явный отказ. Опубликованный `.proto-map.json` нужно хранить вместе со схемой и загружать через Load Protobuf Mapping перед повторной генерацией: он сохраняет wire names, типы/presence и tombstones. Удалённые поля получают reserved number/name и не используются повторно. Изменение несовместимого типа/presence отклоняется.

Если одновременно исчезли и появились поля, генератор требует явное сопоставление в Protobuf evolution, например `note#1=renamed#1`. Номер и wire name сохраняются; новое исходное имя фиксируется в ledger. Эволюция повторяющихся SQL Result aliases требует уникальных псевдонимов: их идентичность нельзя доказать перестановкой. Для необычных имён с `=` или `;` поле также принимает JSON-объект сопоставлений. Не редактируйте опубликованные номера вслепую. Открытие редактора mapping создаёт копию для просмотра/сохранения; новый mapping применяется только после загрузки.

Реестр и проверки v2

`CodeGenerator` расширен обратно совместимыми category / outputs / styles. Реестр содержит 15 явных экземпляров, без динамического class loading. Новый генератор использует тот же `CodeModel` и не меняет MetadataExtractor или SQL Result. Дополнительные параметры находятся в `GenerationRequest.Options`; старый девятиаргументный конструктор v1 сохранён.

Примеры всех новых форматов: `examples/code-generation/v2`. Поддерживаемый стенд: `tools/verify-code-generation-v2.ps1`; регрессия v1 - прежний `tools/verify-code-generation.ps1`. Проверки используют отдельный loopback PostgreSQL 18.x. Отсутствующий runtime отмечается **NOT RUN**, а не PASS. Установки SDK или облачного сервиса для самой генерации нет. Для отдельного compiler/runtime-прогона PHP и Rust есть `tools/verify-generated-php-rust.ps1` с явными путями `-Php`, `-Cargo`, `-Rustc`. Приёмка выполнена на временно распакованных PHP 8.5.12 и Rust 1.99.1; после успешных проверок runtime, архивы и Cargo-кэши удалены, свидетельства сохранены.

Ниже сохранено подробное описание существующих генераторов v1.

Встроенная детерминированная подсистема Java 21 создаёт исходный код по структуре PostgreSQL. Ядро `av.pgdeveloper.codegen` независимо от Swing. Генерация не вызывает AI, Ollama, LLM, внешние API, процессы, AV-Agent или IPC. Компиляторы целевых языков используются только тестами, отдельно от приложения.

Использование

В Object Browser: таблица или представление → **Generate** → **Code…**. Существующие команды SELECT / INSERT / UPDATE / DELETE / MERGE / DDL находятся в **Generate** → **SQL**. В SQL Window команда **Generate Code…** использует выделенный оператор или оператор под курсором. В меню Result Grid она использует метаданные выбранного результата.

Выберите язык, результат Model / Row Mapping / CRUD, вид модели, Naming, Nullable handling, пакет и имя модели. Предпросмотр обновляется после изменения параметров; чтение каталога и генерация выполняются в SwingWorker. Generate перечитывает каталог таблицы/представления. Copy копирует показанный код, Save to File сохраняет те же байты в UTF-8, Open in Editor открывает отдельный редактор исходного кода с подсветкой и действиями Copy / Save.

Новые строки и контекстные подсказки поддерживают русский и английский языки. Код и имена объектов не переводятся.

Источник	Model / Row Mapping	CRUD
Table, включая partitioned table	Да	Только проверенный PK без отложенной проверки

Источник	Model / Row Mapping	CRUD
View / materialized view	Фактическая проекция представления	Нет
SQL Result	Сохранённый ResultSetMetaData, строки не нужны	Нет
SQL Statement	Доказанный существующим scanner полный SELECT * одной schema-qualified таблицы	Нет

SQL Statement никогда не выполняется автоматически и не подготавливается через JDBC ради определения типов. Для JOIN, выражений, агрегатов, CTE и других неподтверждённых структур нужно явно выполнить SELECT и выбрать SQL Result. Даже если в WHERE простого SELECT * есть функция, извлекается только каталог; условие и функция не выполняются. Неквалифицированное имя не разрешается через предположение о search_path: SQL Window может иметь собственные SET search_path и временные объекты. Укажите схему явно или используйте SQL Result.

SQL Result сохраняет порядок, алиасы, повторяющиеся подписи и типы выражений. Структура результата не заменяется структурой исходной таблицы. Например, `select e.employee_id,e.first_name,d.department_name,e.salary ...` создаёт четыре поля результата, даже если обе исходные таблицы содержат другие колонки. Повторяющиеся имена получают уникальные идентификаторы (`id`, `id_2`), а чтение идёт по порядковому номеру, независимо от неоднозначности подписей.

Типы и NULL

Метаданные каталога получаются через расширенный общий `db.Metadata`, включая schema/relation, порядок колонок, PostgreSQL type/OID, precision/scale, nullable, default, identity/generated, упорядоченные PK/FK/unique, происхождение, enum labels, domain base type и array element type. Общий `PgTypes` сохраняет исходную идентичность типа; отдельного дублирующего catalog resolver нет.

PostgreSQL	Go	Java	Python
bigint / integer / smallint	int64 / int32 / int16	long / int / short	int
boolean	bool	boolean	bool
numeric	Numeric (точная строка)	BigDecimal	Decimal
real / double precision	float32 / float64	float / double	float
text / varchar	string	String	str
uuid	UUID (строка)	java.util.UUID	uuid.UUID
date	time.Time	LocalDate	date
timestamp	time.Time	LocalDateTime	datetime без timezone
timestamptz	time.Time	OffsetDateTime	datetime с timezone
json / jsonb	json.RawMessage	исходный JSON String	JsonText

PostgreSQL	Go	Java	Python
bytea	[]byte	byte[]	bytes
массив	PGArray (строка)	PgArray(text)	PgArray(text)
enum	строка + исходный PgType	строка + исходный PgType	строка + исходный PgType
domain	тип базового значения + исходный PgType	то же	то же
composite / extension / прочие типы	текст PostgreSQL	текст PostgreSQL	текст PostgreSQL

Массивы в v1 представлены **точным текстом PostgreSQL**, а не одномерными коллекциями. Это сохраняет размерность, нижние границы, NULL-элементы, вложенные массивы и элементы пользовательских типов. Например, `[0:2]={1,NULL,3}` сохраняется целиком. Составные типы не разворачиваются в отдельные вложенные классы.

Nullable-поля Go используют указатели; Java использует объектные аналоги primitive; Python - Optional. JDBC unknown явно сохраняется как UNKNOWN и консервативно становится nullable. Стратегия Require known nullability отклоняет такой результат. Для JOIN даже объявленное NOT NULL исходной таблицы не считается достаточным доказательством non-null результата. Строки данных для определения типов не читаются.

JSON сохраняется без промежуточного преобразования чисел в float. В Python `JsonText("null")` отличается от SQL NULL (`None`), а `value()` разбирает дробные числа через Decimal. В Java можно отдельно разобрать JSON выбранной библиотекой. Часовой пояс timestamp и timestamptz различается. Значения PostgreSQL infinity, даты вне диапазона целевого runtime и специальные numeric NaN/Infinity не имеют универсального переносимого представления; для таких данных выбирайте явную текстовую проекцию SQL Result.

Код доступа

Go использует стандартный `database/sql` и **pgx/v5/stdlib** как драйвер. Зарегистрируйте его в приложении:

```
import _ "github.com/jackc/pgx/v5/stdlib"
```

Java создаёт record или POJO и работает через стандартные ResultSet / PreparedStatement с PostgreSQL JDBC. Python создаёт dataclass или Pydantic 2 модель и использует **psycopg 3**, без SQLAlchemy. Для Pydantic нужен пакет `pydantic>=2`; в приложение AV-PGDeveloper эти Python/Go зависимости не входят. Поле Package / Namespace задаёт package в Java/Go; для Python это целевой модуль/пакет приложения, указанный в заголовке файла. Файл сохраняется туда, куда пользователь укажет в Save.

Для Python Row Mapping перед выполнением SELECT вызовите сгенерированную `configure_connection(connection)`: она устанавливает точные text loaders JSON/массивов для этого соединения. OID массивов разрешаются по квалифицированным именам в целевой БД, а не переносятся из БД генерации. Альтернатива - явно проецировать массивы/JSON как текст. Уже преобразованные драйвером списки/float не принимаются как точное исходное значение.

CRUD создаёт готовые Get / Insert / InsertExplicit / Update / Delete и reader результата RETURNING. Все пользовательские значения передаются отдельно: `$1` в Go, `?` в Java и `%s` в psycopg. Имена объектов экранированы, casts типов квалифицированы; символы `%` в идентификаторах Python удвоены.

* Get / Update / Delete используют **все** колонки PK в порядке ограничения. У обычных таблиц используется ONLY, чтобы не затронуть строки наследников; для partitioned table доступ идёт через родителя партиций. * Insert пропускает identity/generated и колонки с default, включая serial. InsertExplicit позволяет передать значения колонок с default, но также пропускает identity/generated. Если значение default нужно заменить SQL NULL, используйте InsertExplicit и nullable-поле с null/None. * Update исключает PK, identity и generated. Неизменяемые таблицы без полей для UPDATE получают объяснение ограничения всего режима CRUD. * RETURNING возвращает фактические identity/default/generated. Для Insert передайте экземпляр модели с исходными значениями; значения пропускаемых колонок в этом экземпляре игнорируются. * Транзакция принадлежит вызывающему приложению. Сгенерированный код не делает Commit/Rollback. Go принимает как `*sql.DB`, так и `*sql.Tx`. * При отсутствии строки возвращается ошибка/LookupError/sql.ErrNoRows. UPDATE/DELETE без PK, по ctid или по произвольным полям не создаются. * Optimistic concurrency автоматически не выводится. Между Get и Update другой клиент может изменить строку; бизнес-правило блокировок/версий задаёт приложение.

Пример Java после генерации:

```
try (var connection = dataSource.getConnection()) {
  connection.setAutoCommit(false);
  var saved = SampleRow.insert(connection, draft);
  var same = SampleRow.get(connection, saved);
  connection.commit();
}
```

Пример Python:

```
import psycopg
from models import configure_connection, insert

with psycopg.connect(dsn) as connection:
    configure_connection(connection)
    saved = insert(connection, draft)
```

Полные исходники, полученные тем же реестром, находятся в `examples/code-generation`. Это примеры схемы тестового окружения, не подключение к пользовательской БД.

Ограничения и расширение

Режим CRUD требует PK, validated и non-deferrable; unique без PK пока не заменяет это требование. Представления и произвольные SQL Result допускают только Model / Row Mapping. Неподдерживаемые комбинации очищают предпросмотр и отключают экспорт. Java-модели ограничены пределом параметров конструктора JVM; слишком широкая проекция отклоняется. Лимит IM - 1600 колонок, лимит исходника - 2 MiB. Необычные имена и ключевые слова преобразуются в легальные уникальные идентификаторы, даже в режиме исходных имён.

Публичный `CodeGenerator` имеет id / displayName / supports / generate и limitation для объяснения отказа. `GeneratorRegistry` использует явный список GoGenerator / JavaGenerator / PythonGenerator. Чтобы добавить генератор, реализуйте контракт, зарегистрируйте экземпляр в списке и добавьте вариант выбора в UI. MetadataExtractor и модель SQL Result менять не требуется. Динамической загрузки сторонних классов нет. TypeScript, C++ и JSON Schema добавлены в v2 вместе с остальными новыми форматами. Spark/PySpark остаётся возможным последующим расширением.

Воспроизведение проверок

`gradlew.bat test` проверяет детерминированность, ограничения, точный экспорт и компиляцию Java. `tools/verify-code-generation.ps1` создаёт **новый изолированный loopback-кластер PostgreSQL**, запускает Java/Go/Python CRUD и Swing-сценарии, проверяет регрессию пользовательских типов и останавливает только свой кластер. Нужны существующие PostgreSQL binaries, JDK 21, Go, Python 3.12 с psycopg 3 и Pydantic 2. Пути задаются параметрами скрипта; SDK автоматически не устанавливаются. Go-проверка использует локальный module cache с pgx v5.7.6 и GOPROXY=off. Артефакты и логи сохраняются вне репозитория в Codex-Shared.