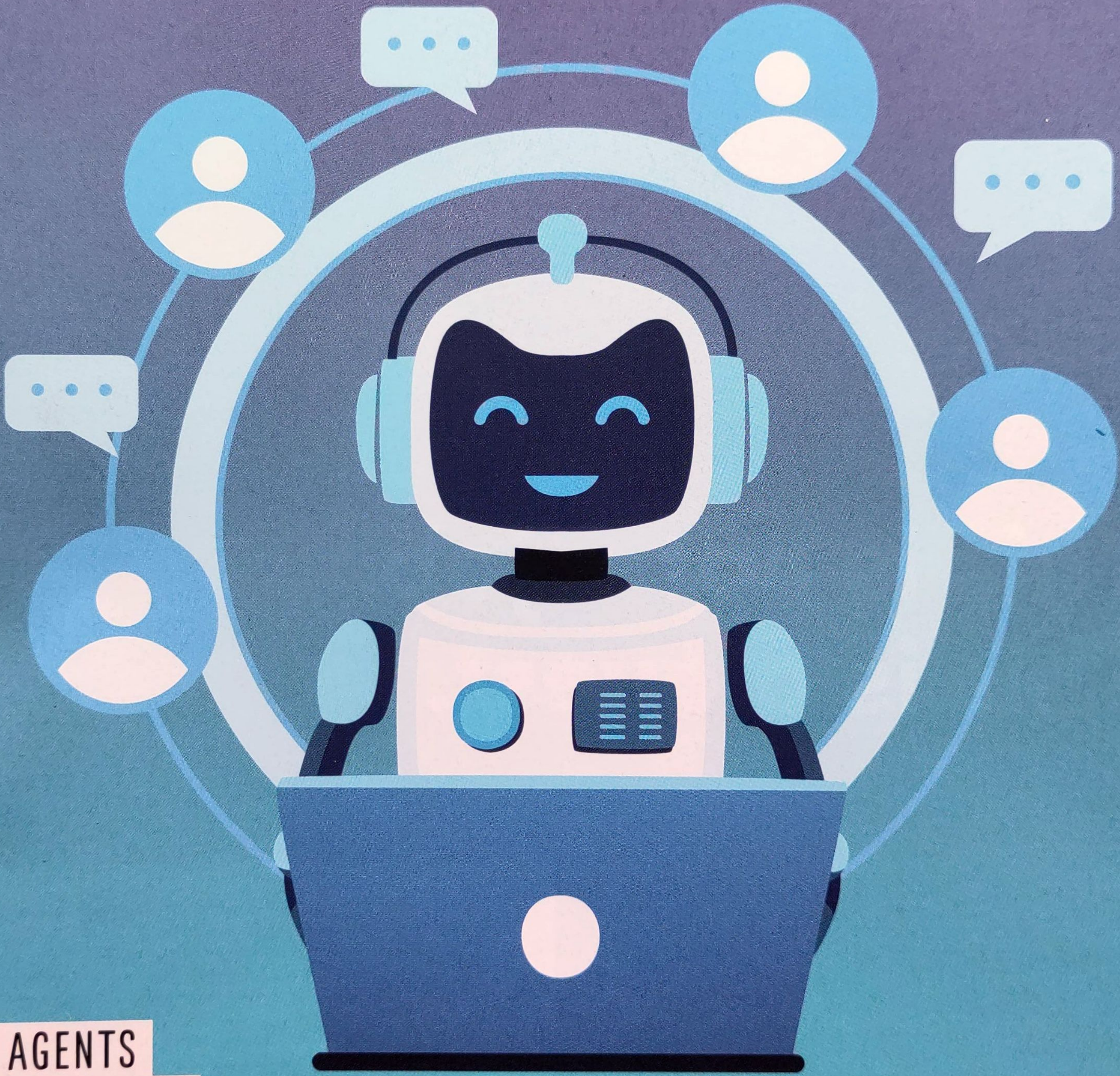
JAVA { MAGAZINE

nl.jug

01 > 2026

Onafhankelijk tijdschrift voor de Java-professional



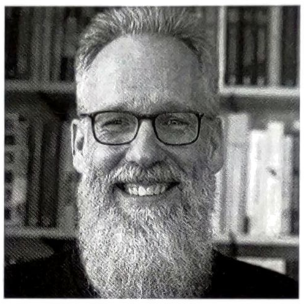
›AI AGENTS
WHEN YOUR CHATBOT CAN FINALLY DO THINGS

›WIRED 2.0
CREATE YOUR ULTIMATE LEARNING ENVIRONMENT

›QUARKUS
WHAT HAVE WE LEARNED?

OLD MAN YELLS AT CLOUD DB SCHEMA

In which I explain my personal rules for defining a schema in relational databases



Jan started programming when cassette tapes were still used for data storage and rotary phones were still a thing. Currently, he works for Yoink where he hopes to stay until he retires.

I have four rules that I follow when I define a relational database schema:

1. Implement Flyway from the start to avoid future regret
2. Slap a separate primary key field on every table
3. Yes, this includes join tables
4. Be consistent

These rules have served me well over the years. Yes, there are always exceptions to any rule, but I believe it's good to follow these unless you have a good reason not to.

Let me introduce Chad, who represents many of my past and current teammates. Chad is smart and very experienced with tools like Kafka, Redis and Cassandra, but hasn't worked with relational databases yet, because I'm old and they're not.

Of course Chad can totally pull up a *PostgreSQL* in `$preferred_cloud_provider` and make it go *brr*, but they don't have the painful experiences I've had. I've discussed these rules with three different Chads in the past year alone.

My rules are laughably easy to implement in a fresh database, but extremely difficult and expensive when the database has been running in prod for some time. So let's go over each one.

{ 1. IMPLEMENT FLYWAY FROM THE START }

Flyway is a tool that manages schema migrations. Migrations are when you want to make a change to an existing schema, like adding a field or a table. There are alternatives to Flyway, like *Liquibase*, which are all fine.

One time Chad argued that their microservice was very small and wouldn't ever change, but they were wrong. It's software, therefore it will change, and they need to prepare for that. Flyway is that preparation.

Without Flyway, when Chad has to make a change, they'll have to do it manually in test, accept, prod, and oh yeah, the entire team's local machines. Good luck keeping everything in sync Chad.

If Chad decides to add Flyway later, they're in for a lot of pain too, because Flyway will not be able to see any pre-existing schemas. I once had to add Flyway to an existing database. It took me weeks.

{ 2. SLAP A SEPARATE PRIMARY KEY FIELD ON EVERY TABLE }

Another time, Chad chose a phone number field to be the primary key. (This is a *natural ID*.) I told them to add a separate field whose only function is to be a primary key (a *surrogate ID*).

I know, *natural ID* sounds wholesome and cuddly, while *surrogate ID* sounds cold and, well, unnatural. Still, surrogate IDs are better.

Why? Because it's not just software that changes; the world changes too. A phone number seems unique enough, but what if the owner switches providers and the number gets reassigned to someone else? Or worse: I remember when the GDPR was introduced. We had to find a way to delete the phone number but keep the rest of the data. Think about all the foreign keys that point to this phone number field!

Better to just have an auto-incremented number (don't forget to give it its own dedicated sequence!) or a generated UUID, right?

{ 3. YES, THIS INCLUDES JOIN TABLES }

This also applies to join tables (with a caveat).

People may start using your software in unexpected ways, and what once was a humble join table could become an entity in its

own right. If it does, you'll be glad if it already has its own surrogate ID.

For example, a link between Employees and Departments might become its own entity, like a Project or Assignment, in version 2.0. Better to have a surrogate ID field ready to go!

The caveat is that JPA doesn't handle link tables with an explicit ID field very well: the join table must become an `@Entity` and you can't use `@ManyToMany`. If you must use JPA, you can skip this rule.

(This is a consequence of ORM being a leaky abstraction, but that's a topic for another rant. Take a look at jOOQ or Jdbi, or if it's a small service, just rawdog the JDBC, it's not that hard!)

{ 4. BE CONSISTENT }
Sometimes when you have multiple tables, it can be useful to write some generic code that works across all tables. But you can only do that if all tables work the same way.

For example, if you need to set up an audit trail, it's really annoying if all your tables have a `created_at` field except the Account table where it's called `create_time` because it was written by someone who didn't pay attention to the other tables that already existed in the database.

I recently inherited a microservice from Chad when they left the company, where every table had a string natural ID except for one which had an auto-incrementing integer surrogate ID. I had to special-case all my shared integration test logic for that one entity and it made me sad and angry.

{ CONCLUSION }
These are my personal rules for defining a schema for a relational database. I hope they can help you and your team (I'm looking at you, Chad!) make good decisions. Of course you don't have to follow them, but if you don't, I hope it's because you disagree with them and not because you weren't aware. ◂

Space constraints in this issue meant some of the story had to be shortened. The story as originally intended is available at jqno.nl/blog.

{ LINKS }

Flyway: **https://github.com/flyway/flyway**
Liquibase: Flyway alternative: **https://www.liquibase.com/**
jOOQ: **https://www.jooq.org/**
Jdbi: **https://jdbi.org/**