JoinedWorkz

Standardize persistence and APIs while keeping developers in full control.

JoinedWorkz is a model-driven toolchain that lets your teams describe persistence, APIs and controllers in a domain-specific language (DSL) and generate consistent, standard-compliant code – while keeping business logic fully in the hands of developers.

Model-driven generation for APIs and backend systems

THE PROBLEM

In larger organizations with several teams and services, the same issues keep appearing:

- **Architecture drift:**
  Each team interprets guidelines differently. Over time, patterns diverge and systems become harder to maintain.
- **Repetitive boilerplate:**
  Persistence, APIs and controller layers are implemented again and again, with only minor variations.
- **Guidelines not consistently applied:**
  Architecture rules live in Confluence or internal portals, but under time pressure they are partially ignored or interpreted loosely.
- **API-first initiatives that stall:**
  OpenAPI specs and documentation drift away from the implementation and lose trust.
- **Slow onboarding:**
  New developers struggle to understand how entities, APIs and conventions fit together across services.

WHAT JOINEDWORKZ IS

JoinedWorkz is a model-driven generation toolchain for APIs and backend systems with four main building blocks:

1. **Textual DSL**
   A domain-specific language to describe domain entities, persistence aspects, APIs and controllers in a Git-friendly, reviewable form.
2. **Standalone modeling studio**
   A dedicated modeling tool that runs alongside your IDE of choice, used to edit, navigate and validate models.
3. **Standard and custom generators**
   Generators that produce OpenAPI specs, persistence mappings, repositories, controllers and other infrastructure code. They can be used as-is or extended for your organization.
4. **Maven-based build integration**
   A Maven plugin that runs generators as part of your build, keeping model and code aligned without extra scripts.

Internally, the DSL is transformed into a simple, generator-friendly representation that avoids extra runtime framework dependencies.

BENEFITS FOR YOUR ORGANIZATION

- **Executable architecture decisions**
  Architecture rules move from documents into generators. Every new project and feature automatically applies the same patterns.
- **Consistent APIs and persistence**
  Shared models and generators ensure similar structures across services, making integration, operations and cross-team work more predictable.
- **Faster project start**
  New services start from a generated structure that already follows your standards, instead of a manually assembled boilerplate.
- **Better onboarding and transparency**
  The model gives a compact overview of entities and APIs. Because code is generated from this model, the overview stays aligned with the implementation.
- **Reduced risk of tool lock-in**
  Generated code is normal project code. It can be checked in, reviewed, refactored and maintained even if you decide to stop regenerating.

GOVERNANCE AND DEVELOPER AUTONOMY

JoinedWorkz is explicit about who owns what:

Generators typically own

- Persistence mappings and repository patterns
- API contracts and controller skeletons
- Integration points and cross-cutting concerns

Teams always own

- Business services and domain-specific logic
- Workflows, orchestration and project-specific refinements
- The decision when to regenerate and when to take over code manually

Generators standardize the repetitive, architecture-heavy parts. The domain logic where teams need freedom remains hand-written.

HOW IT WORKS (IN PRACTICE)

1. **Model:** Describe entities, persistence aspects, APIs and controllers in the DSL, using the modeling studio and your usual Git workflow.
2. **Transform & generate:** Transform the DSL into an internal representation and run generators that emit OpenAPI specs, repositories, controllers and supporting code.
3. **Build & evolve:** Integrate generation into your Maven build. Treat generators as versioned modules that evolve together with your architecture.

Typical adoption starts with a focused use case (e.g. one service), a first custom generator and a pilot project. From there, patterns and generators are rolled out to additional systems.

NEXT STEPS & CONTACT

If you want to explore model-driven generation for your APIs and backend systems:
1. Review the documentation and examples.
2. Identify a realistic pilot area (one service or domain).
3. Discuss how a first generator and project rollout could look in your context.

https://joinedsystems.eu karl.hoenninger@joinedsystems.eu https://joinedworkz.org