2nd Annual
MidPoint Community Meetup

# Rapid Connector Development using AI

Evolveum

Funded by the European Union
NextGenerationEU

RECOVERY AND RESILIENCE PLAN

**Tony Tkáčik**
Backend Technical Leader

# Agenda

- Introduction
- Motivation
- Our solution
- Demo
- Details & What's planned for 4.11

# Motivation

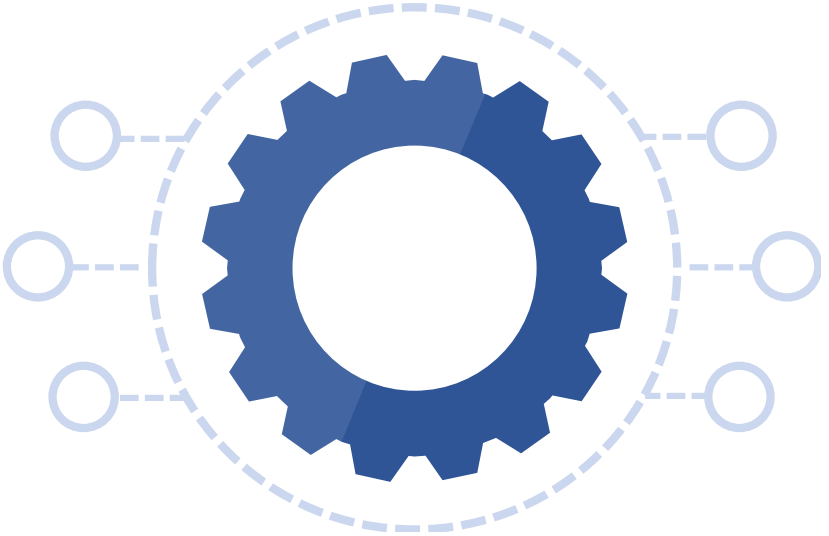
- Slow pace of connecting new applications & systems
  - Based on surveys
  - One of the pain-points of IDM / IGA solution implementation
- Lot of duplicate code across multiple connectors
  - Authentication
  - HTTP Handling
  - Data serialization

# Our Solution

- Methodology-driven approach for AI-assisted connector development with human validation at every step
- Decreasing workload for developing application specific connectors by having common framework

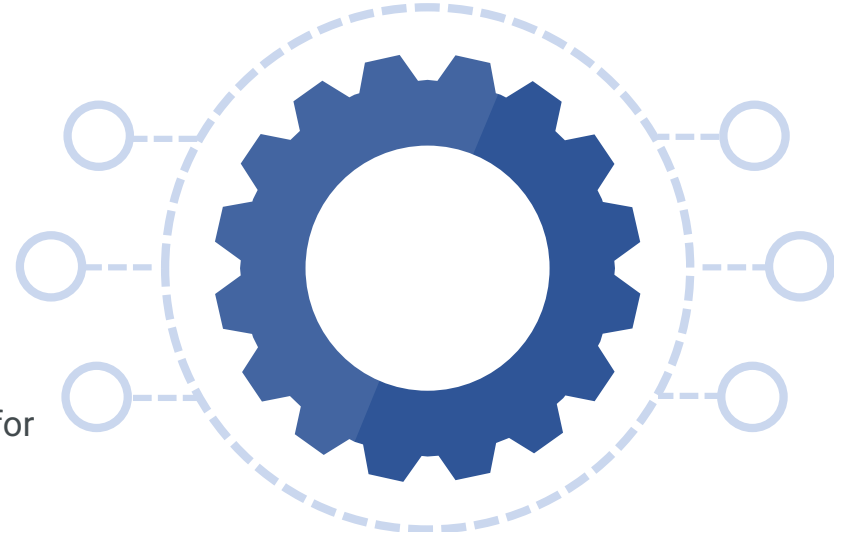
# What's new for 4.11?

- New **methodology** for **connector development**
  - Reduce the **effort** and **time**
  - Reduce the **amount of code**
  - Deploy and **test faster**
- New **AI-assisted wizard in MidPoint** for connector development



# What's new for 4.11?

- **Set of no-lode / low-code connector frameworks**
  - build on top **ConnId**
  - **Common functionality** for connectors
  - **SCIMREST Framework -** flexible and extensible foundation for REST & SCIM 2.0
  - **Mutitable (SQL) Connector** & Framework - flexible and extensible foundation for databases



# Methodology-based Approach to Connector Development

- Designed for quick wins
  - Providing value as soon as possible
  - Well-defined functionality chunks
- Uses AI recommendations when available
- Can be followed without AI (flexible)
- Multiple levels of capabilities
  - Read-Only
  - Full CRUD Support
  - Relationships (native associations)

# Methodology – Base Steps

- Identify application
- Gather necessary documentation
- Implement initial connection to application
- Implement reading support for user / account
  - Schema
  - List objects
  - Get objects

# Methodology – Advanced Steps

- Implement reading support for other entities
- Implement editing capabilities
  - **Create / Update / Delete**
- Implement relations support (native associations)

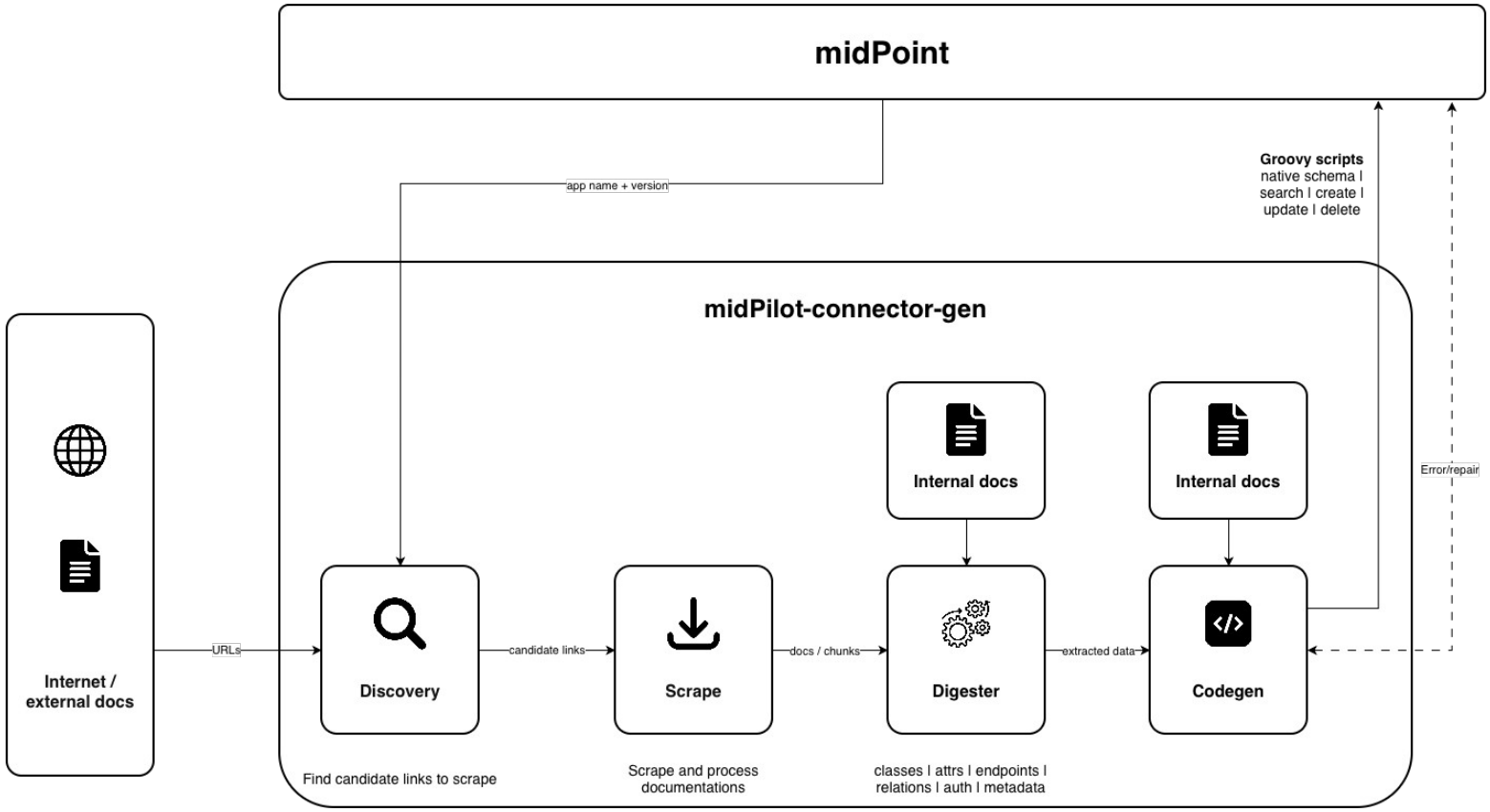
# How AI fits into this picture?

- Methodology allowed for structured workflow
- AI pipeline & functionality for
  - Documentation selection
  - Structured Information Extraction
  - Definition & Code Generation
- Step-by-step interactive wizard is better UX then prompt window
- Proposals can be validated more easily

# Wizard over prompts

- Allow to optimize and standardize UX
  - Display information / data in most-suitable forms
- Guides you throught the process
- **Human-in-the-loop**
  - Scripts are executed only when approved
  - Allows you to test early
- Support for multi-session development

# Architecture Overview




Evolveum

2nd Annual
MidPoint Community Meetup

# Demo

# Connector Generator Wizard

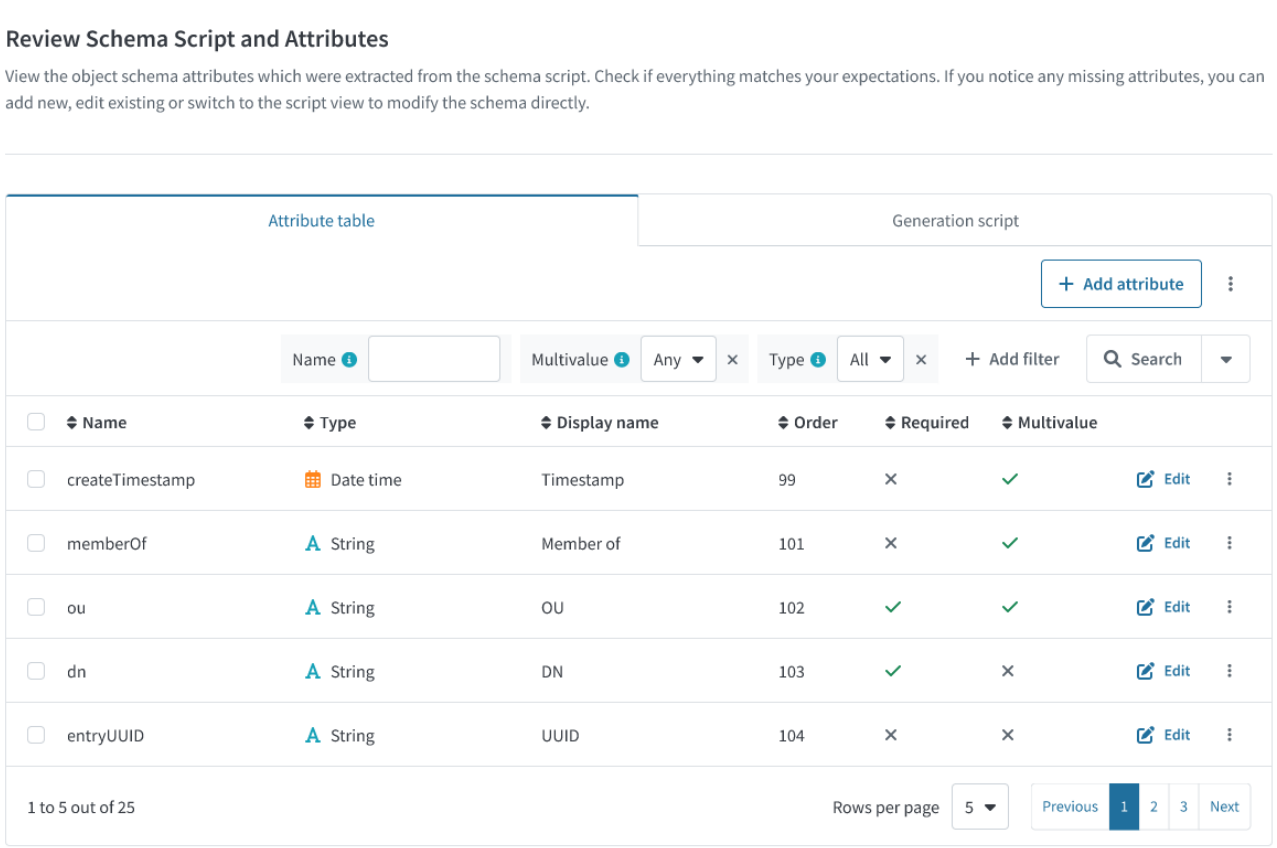
# Connector Generator Wizard

- Technically most complex wizard in midPoint
- Designed for multi-session workflow

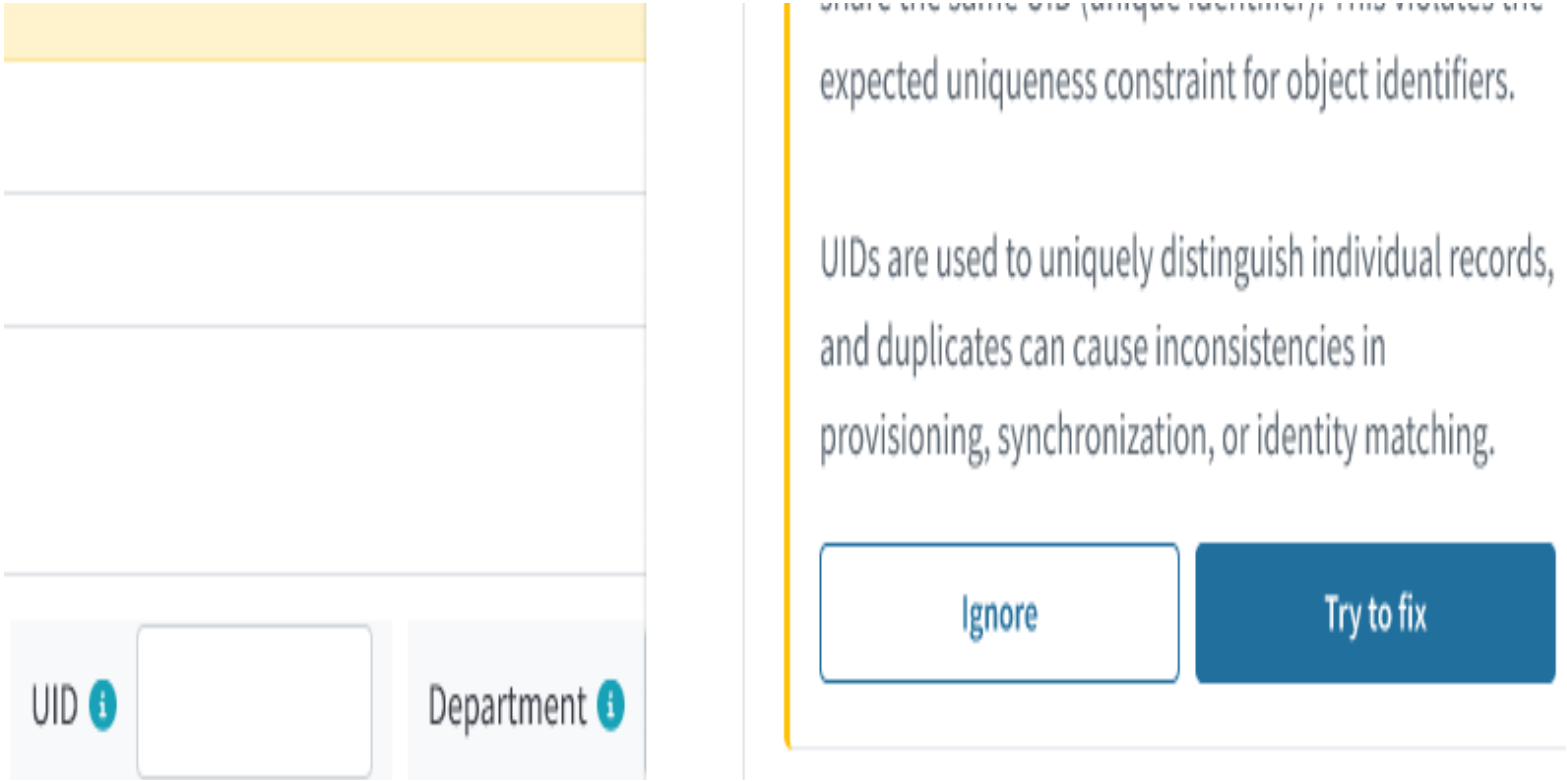
# Features planned for 4.11

- Forms over code
- More Testing & Acceptance Views
  - Filter Support
  - Create, Update, Delete
- Better Visibility for testing
  - Logs in GUI
  - Proactive Warnings
  - Error Reporting

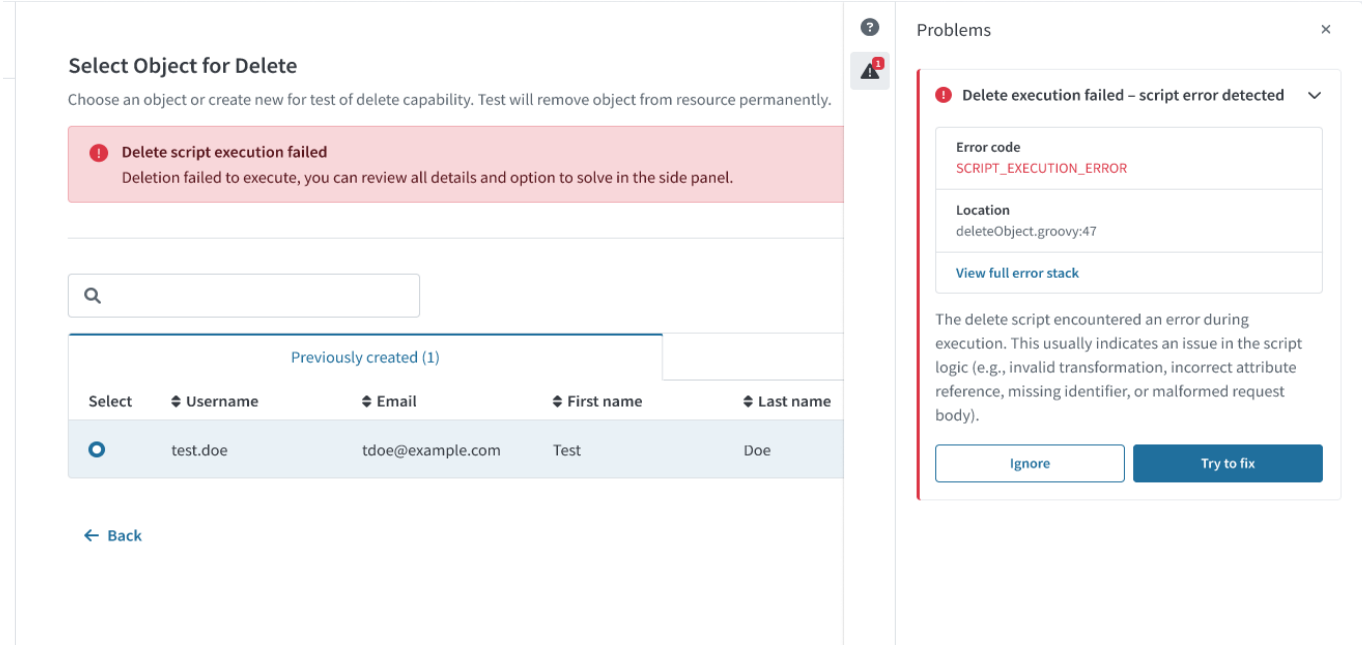
# Schema Editor View



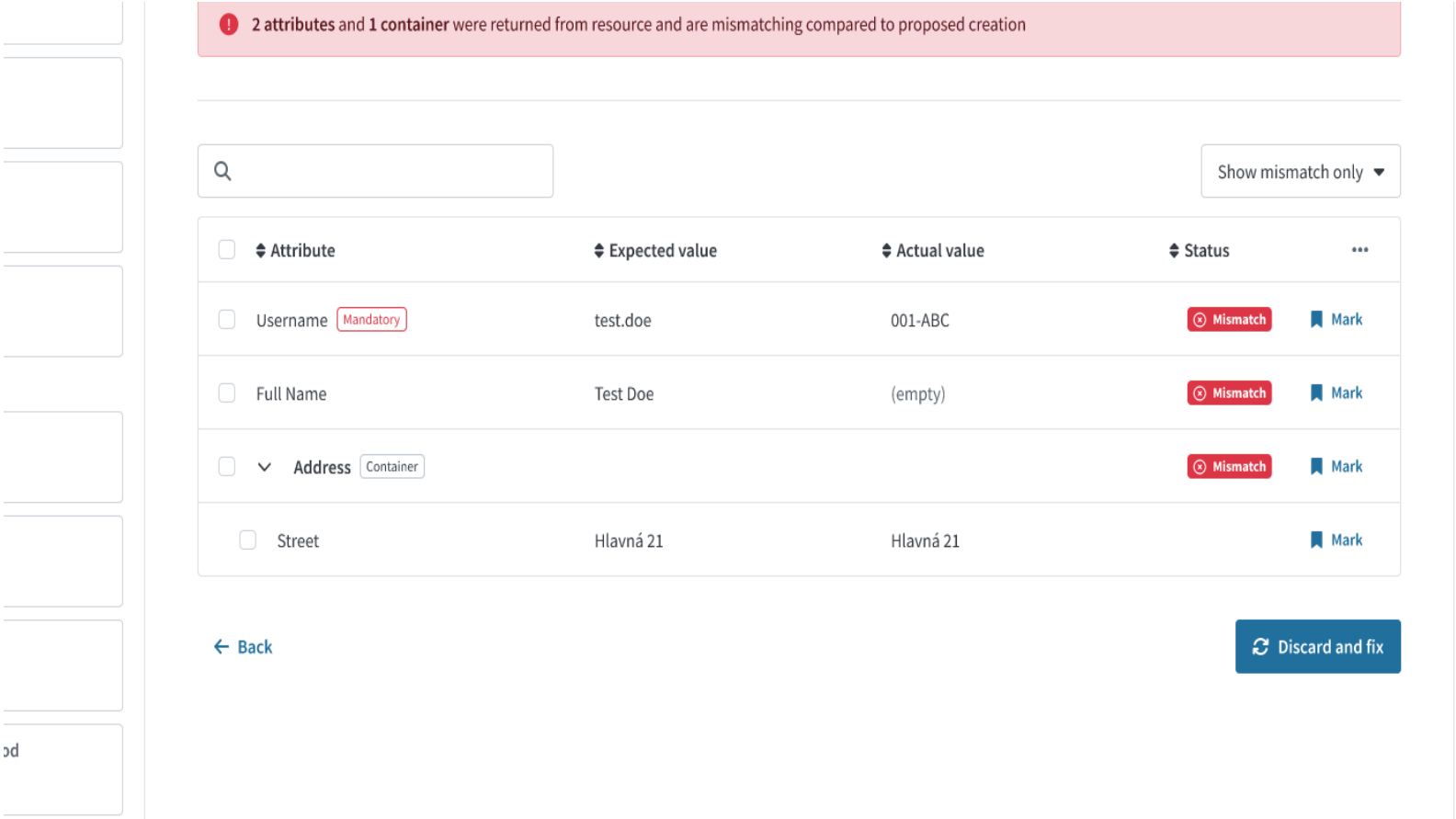
# Visibility of potential issues



# Error Visibility & Context



# Built-in Validation & Detection

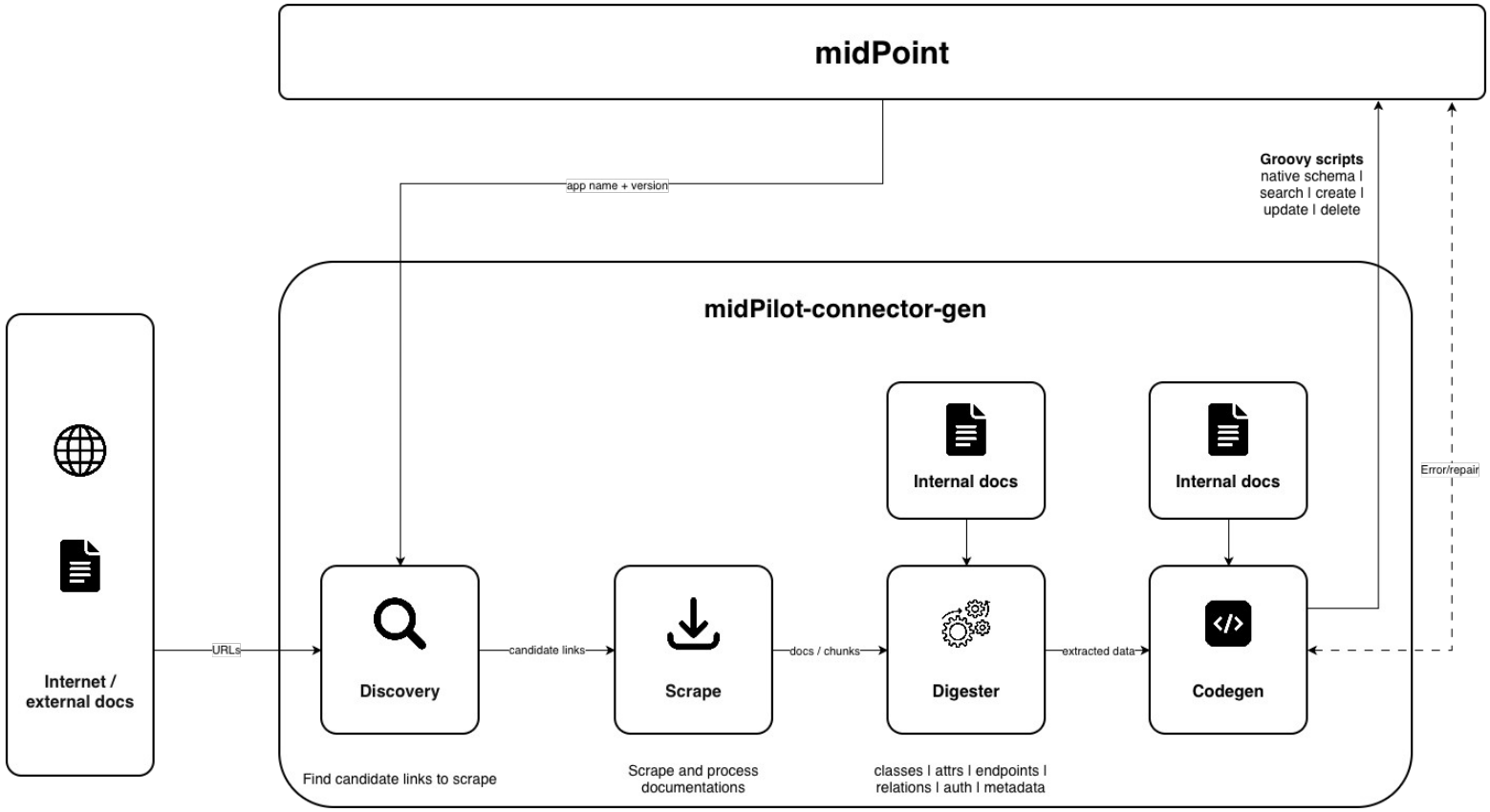


# MidPilot Connector Generator Service

# MidPilot Connector Generator Service

- An open-sourced **AI-assisted service** from Evolveum
  - **Automates creation** of midPoint **connectors from** API **documentation**
  - **Can be self-hosted**
- Automatically **discovers documentation**
- **Extracts Schemas** and Endpoints
- **Generates definitions & Groovy code** based on the frameworks
- Comunicates with LLM & MidPoint only

# Architecture Overview




Evolveum

2nd Annual
MidPoint Community Meetup

# Discovery

- Used in first steps in wizards
- Searches web for application API documentation
  - **Brave Search** or DDGS
  - Multiple queries with different keywords
- Uses LLM to filter out search results to best candidates

# Scraping

- Scrapes documentation from accepted search results
- Uses web browser for compatibility and scraping (playwright with Chromium)
- LLM
  - inspects links and marks them for download / scraping
  - tags chunks based on their content to optimize context building for actual operations

# Digest

- Multiple endpoints, prompts for specific functionality
- Uses only task-specific parts of documentation
- Extracts information in structured form
  - **Basic Information**
  - **Authentication Methods** and their quirks
  - **Object Classes** – structured list and basic properties of all object classes
  - Endpoints – endpoints and their classifications, use, documentation
  - **Attribute / Schema detection**
  - **Relations**

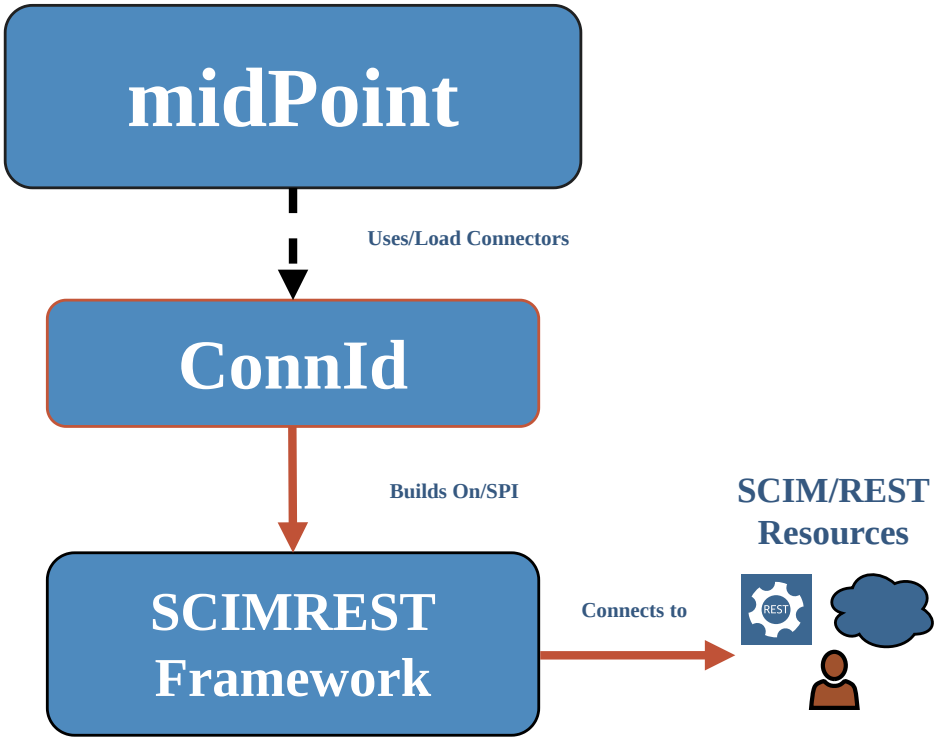
# Codegen

- Uses Evolveum documentation for frameworks
- Uses **only relevant context** for the current task: schema, endpoints, auth, relations
- Small, operation-focused parts instead of one large connector

# Connector Frameworks

# No-Code / Low Code Connector Frameworks

- **SCIMREST**
- **Multitable**
- **Declaration** over code
  - YAML, Groovy
- Common functionality built-in
  - handlers and strategies for operation support
  - Reduces amount of code needed to be generated
- Object Class specific handlers are first-class citizen
- Allows incremental development



# Why Groovy DSL

- Lean syntax **reduces boilerplate**
- Declarative-style builders
- **Reduced** code **verbosity**
  - **Faster** development
  - **Fewer** bugs
- Suitable **for more customized solutions**
  - **Advanced use-cases** and customization

```
objectClass( className: "Group") {
    create {
        scim {
            supportedAttributes  ...attributes: "displayName", "externalId"
            // AWS supports adding max 100 members during create
            supportedAttribute( attributeName: "members") {
                limitations {
                    maxPerRequest  maximum: 100
                }
            }
        }
    }

    update {
        scim {
            patch {
                supportedAttributes  ...attributes: "displayName", "externalI

                supportedAttribute( attributeName: "members") {
```

# Why YAML

- Purely **declarative**, mostly **static data only**
- Well known by system **Administrators,** non-developers
- **Readability:** easier to review and understand
- Groovy scripts only at specific well-defined
  - Easier **visual control**
- Best **for simple implementations, well-defined objects** and application supporting SCIM 2.0

```yaml
objectClasses:
  User:
    attributes:
      email:
        description: User email address
        required: true
        json:
          path: extras.emails
          type: string
```

# Why not pure Java?

- More verbose
- Less suitable for non-developers
- Requires recompile & package cycle
- Harder to review
- In order to inspect connector you need source code or decompile it
- Harder to automate review using heuristics

# Common Built-In Functionality

- Transformation of protocol specific types to MidPoint / ConnId types
- Support for multi-operation Create, Update strategies
- Attribute Resolution
- Relationships

# SCIMREST Framework

- **No-code / Low-code** approach.for HTTP / REST based APIs
- Common Model for **Authorization** and **Authentication**
- **REST and SCIM support**
  - **Support** for handling non-standard cases
  - Automatic **SCIM** schema discovery

# REST

- REST is architectural / design style for HTTP based APIs
- APIs have different structure between applications
- OpenAPI specifications have different modeling approaches

# SCIM 2.0

- **SCIM v2** (System for Cross-domain Identity Management)
  - IETF **standard** - defines **common JSON schemas** and an REST API for provisioning and synchronization
  - **Reduces custom connector work** by using a common schema
  - Still REST based protocol

# Isn’t SCIM 2.0 supposed to be universal?

- **Yes**, but still application specific connectors makes sense:
  - Not all features supported across multiple vendors
    - Different limitations, requirements
  - Custom additional resources
    - Relationships are not mapped uniformly
  - Not all functionality exposed as SCIM Resources

# SCIM vs REST diferences

- With REST you need to **declare the structure of application APIs**
- With SCIM you **declare the additions and exceptions from** the standard.
- Less verbose, automatic and implicit handling
- **Customizable overrides** in case of special handling and differences from the standards.

# Authorization and Authentication handling

- Pre-defined & built-in support for
  - **Basic, Token, Api Keys, JWT, OAuth2**
- Common Authentication and Authorization strategies
  - **Connection** set-up
  - **Error handling**
- Customizable via scripting for appliction specific quirks

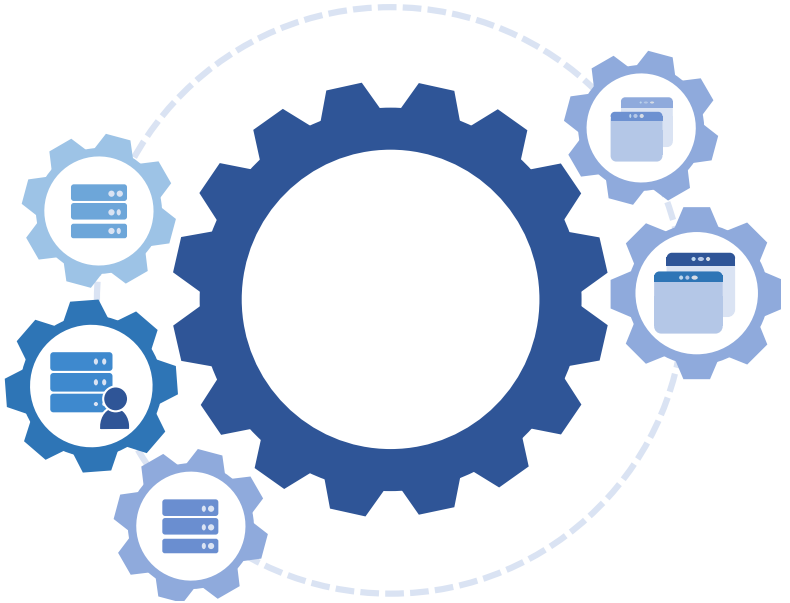
```
authentication {
    rest {
        // Sets the preference of Authentication mechanisms
        preference tokenBased, apiKey, basic

        apiKey {
            header "X-API-KEY"
        }

        tokenBased {
            implementation {
                request.header( name: "Authorization", "toke
            }
        }
    }
```

# Object Class Schema

- **Define** managed **Object Classes**
- Map **Native** attributes to **ConnId** Attributes
  - Simple and Complex
  - Built-in ConnId attributes
- Define **properties** and **limitation** for Attributes and Object Classes
- Define **custom relations** and **handling** for attributes
- Allows attribute renames



# Object Class Schema: Attribute Names

- Attribute can have multiple different names based on point of view
  - MidPoint / ConnId – name used to export to midPoint
  - Protocol Name – name used in API definitions / procotol
  - Application Name – name actually used in application
- Defaults: all  attribute names are same

```
attribute("username") {
    connId {
        name "NAME" // Built-in required ConnId Attribute
    }
    json {
        // In core REST API it is called login
        // for backwards API compatibility
        name "login"
        type "string"
    }
    description "the user's username";
}
```

# Attribute Serialization and De-serialization

- **Implicit**, based on computed protocol mappings
- **Explicit**, implemented via Groovy
  - Protocol **specific format** (i.e. other than a flat structure of attributes)
  - **Different endpoints** have to be accessed
  - Other **edge cases** …

# REST: Explicitly Declared Schema Mapping

- **Declare Schema** (custom Groovy DSL)
  - Object classes and attributes
- Declare **relationships**
- Handle **complex attributes**
- Handle **special cases**

# SCIM: Dynamic Schema Mapping

- **Dynamic Discovery** of schema
  - During the "**schema()**" operation
  - Using "/ResourceTypes" and "/Schemas" endpoints
  - Generates Schema objects
    - Names, Types (including complex) and flags
  - Includes common and extension schemas (Unique Namespaces)
- **Automatic** object **translation**
  - Based on pre-build SCIM strategies
  - Schema Registry build during Schema Discovery

# SCIM: Overriden Schema Mapping

- Allows to override / disable automated mapping
- Attributes can be mapped to specific paths
  - Some applications supports only single value for emails / photos
- Assign aliases to URN extension names
- Attribute names can be changed to use application names instead of protocol names

```
objectClass("User") {
    scim {
        // Only listed attributes are supported
        onlyExplicitlyListed true
        extension("slack", "urn:ietf:params:scim:schemas:extension:slack:profile:2.0:
        extension("enterprise", "urn:ietf:params:scim:schemas:extension:enterprise:2.
    }

    attribute("userName")
    attribute("nickName")
    attribute("displayName")
    attribute("title")
    attribute("email") {
        scim {
            path attribute("emails").firstValue().child("value")
        }
    }
    attribute("profile_photo") {
        scim {
            path attribute("photos").firstValue().child("values")
        }
    }
```

# Operation Implementations

- Frameworks allows you to implement operation support per object class using declarative style and built-in strategies
  - **Search** – and its special cases **get, list**
  - **Create**
  - **Update**
  - **Delete**

# Custom Implemementation

- Sometimes only custom implementation is possible to support operation
  - API does not have coresponding endpoints
  - Endpoint uses different / custom format from the REST
- E.g. objects without list all endpoints (Team in **Gitea**, **Forgejo, Codeberg**)

# REST: Endpoint-based declarations

- Allows to declare HTTP method, URI pattern
- Supported features
- Parsing and body extractions
  - Defaults to parsing based on schema, can be overriden

```
objectClass("User") {
    update {
        endpoint(PATCH, "/users/{id}") {
            request {
                contentType APPLICATION_JSON
            }
            supportedAttributes "admin", "email","login", "language",'
        }

        endpoint(POST, "/users/{id}/lock") {
            request {
                body EMPTY
            }
            supportedAttribute("status") {
                value "locked"
            }
        }

        endpoint(DELETE, "/users/{id}/lock") {
            request {
                body EMPTY
            }
            supportedAttribute("status") {
```

# Search operations

- **list -**list all objects, it is special case of search without filter specified
- **get** – get one object it is special case of search with id filter
- **search** – support for search with filters support

# REST: Search Filter supports

- Allows to use multiple endpoints for different filters
- Correct endpoint is selected based on incoming filter and supported filter declaration

```
objectClass("User") {
    search {

        endpoint("/users/search") {
            emptyFilterSupported true

            supportedFilter(attribute("id").eq().anySingleValue()) {...}

            supportedFilter(attribute("login").contains().anySingleValue
                request.queryParameter("q", value)
            }
        }
        endpoint("orgs/{org}/members") {
            responseFormat JSON_ARRAY
            pagingSupport {...}
            supportedFilter(attribute("organization").eq().anySingleValu
                request.pathParameter("org", value.value.uid)
            }
        }

        endpoint("teams/{id}/members") {
            responseFormat JSON_ARRAY
```

# SCIM: Get, List, Search

- Defaults: built-in as defined in **SCIMv2 specifications**
- **Limitations** have to be **declared**
  - Applications may supports only subset of filter

```
objectClass("User") {
    create {...}
    update {...}

    search {
        scim {
            limitations {
                supportedFilter attribute("externalId").eq().anySingl
                supportedFilter attribute("userName").eq().anySingleV
                supportedFilter attribute("groups").child("value").eq
            }
        }
    }
}
```

# REST: Custom List All Implementation

```
objectClass("Team") {
    search {
        custom {
            emptyFilterSupported true
            implementation {
                def orgs : RestSearchOperationBuilder = objectClass("Organization").search()
                for (def org :orgs) {
                    def teamFilter : EqualsFilter = attributeFilter("organization.name").eq(org.name.nameValue)
                    objectClass("Team").search(teamFilter, resultHandler)
                }
            }
        }
    }
}
```

# REST: Create Operation

- Declared using **create** keyword
- Endpoint-based style
- Endpoint selected on supported attributes
- Defaults
  - Serialized using definition in schema
  - Createable attributes are supported
  - other updateable attributes will be handled by follow-up updates

```
objectClass("User") {
    create {
        endpoint(POST, "/users") {
            request {

            }
        }
    }
}
```

# Update Operation

- Declared using **update** keyword
- Built-in strategies for
  - absolute & relative changes
  - decomposing into multiple partial requests if necessary
- Supports custom partial handlers

# REST: Update Operation

- Defaults:
  - PATCH method assumes relative updates
  - PUT method assumes absolute updates
- Supports using multiple endpoints per update
  - Based on changed attributes and attributes supported by endpoint

```
objectClass("User") {
    update {
        endpoint(PATCH, "/users/{id}") {
            request {
                contentType APPLICATION_JSON
            }
            supportedAttributes "admin", "email","login", "language","lastName"
        }

        endpoint(POST, "/users/{id}/lock") {
            request {
                body EMPTY
            }
            supportedAttribute("status") {
                value "locked"
            }
        }

        endpoint(DELETE, "/users/{id}/lock") {
            request {
                body EMPTY
            }
            supportedAttribute("status") {
```

# SCIM: Update Operation

- Defaults assume full support for SCIM patch and put operations
- Allows to specify vendor limititations
  - relative updates only for subset of attributes
  - limits per changes of multi-value attributes

```
objectClass("User") {
    create {...}
    update {
        scim {
            patch {
                supportedAttributes "externalId", "displayName", "nickName", "
                        "preferredLanguage", "locale", "timezone", "name", "en

                supportedAttribute("userName") {
                    limitations {
                        operations ADD, REPLACE
                        maxPerRequest 1
                    }
                }

                supportedAttribute("active") {
                    limitations {
                        operations ADD, REPLACE
                        maxPerRequest 1
                    }
                }

            }
```

# Delete Operation

- Deletes object, declared using **delete** keyword
- **REST**: Usually HTTP method delete
- **SCIM**: built-in, some customization flags for vendor specific quirks
  - soft delete instead of actual delete

# Relationship

- Defines relationship between objects
- **Subject** & **Object** of the **relationship**
  - Define relationship object classes for both
  - The **reference attributes**
    - Added **automatically to** the **schema**
    - Allows for explicit **Serialization** & De-**serialization** if needed
    - Supports **attribute resolver, custom implementation**

```
/** Custom association between User and Office (custom SCIM type)
relationship("OfficeEmployment") {
    subject("User") {
        attribute("office") {
            multiValued true
            // Emulated true is implied by presence of resolver
            resolver {
                resolutionType PER_OBJECT
                // If not specified, we can assume search is perfo
                search {
                    attributeFilter("employees").eq(value)
                }
            }
        }
    }

    object("Office") {
        attribute("employees") {
            scim {
                implementation {
```

# SCIM (Relationships)

- The framework **automatically** detects SCIM **relationships** from base standard
  - User **“groups”** attribute for relationship to Group objects
  - Group “**members**” attribute as a relationship to User objects
- **Customization** is possible, similar as with REST examples
  - **Customize** the **default behaviour**
  - Add **relationships** between **custom object classes**

# Multitable Connector Framework

- Planed for 4.11
- For direct Database Connections
- Core concepts common with SCIMREST
  - Similarities to SCIM 2.0 flows
- Auto discovery of database schema from JDBC metadata
- SQL dialect abstraction
  - PostgreSQL, MySQL, Oracle, SQLite
- Built-in support for CRUD operations
  - Allows for same level of cusomization as SCIMREST

# Conclusion

- New **methodology** for **connector development**
- New **AI-assisted wizard in MidPoint** for connector development
- **Set of no-lode / low-code connector frameworks**
  - **SCIMREST Framework -** flexible and extensible foundation for REST & SCIM 2.0
  - **Mutitable (SQL) Connector** & Framework - flexible and extensible foundation for databases

# Conclusion

- Evolveum **Github**
  - **midpoint**
  - **Midpilot-connector-gen**
  - **connector-scimrest**
- Evolveum Docs
  - **SCIMREST Framework**
  - **List of Identity Connectors**
  - ConnId 1.x **Connector Development Guide**
- Any connector **contributions** are welcome

# Thank you for your attention

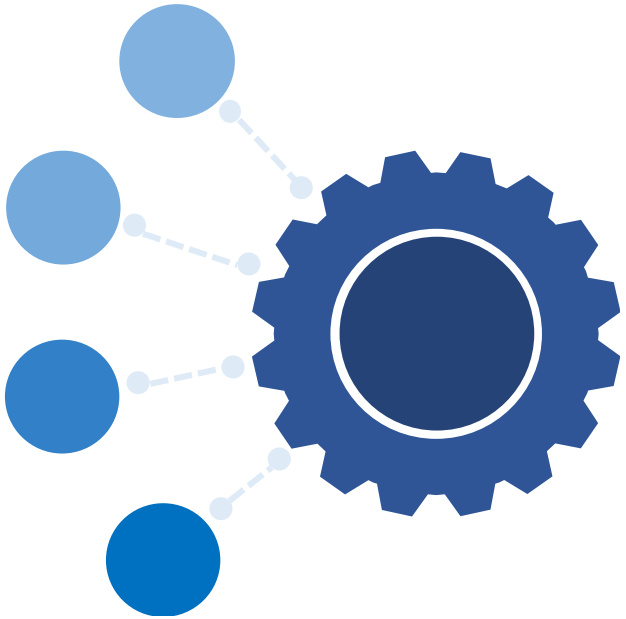
**Feel free to ask your questions now!**

# Isn’t Database Schema Enough?

- Yes, for simple applications
- Database schema and data in database may not capture all possible states and semantics

# Can I use other AI tools for connector development?

- Yes, this methodology does not preclude from your approach for connector development.
- But be careful about some pitfalls:
  - API Documentation may contain imprecisions
  - Test against real applications, not just mocks / AI generated test cases



# 2nd Annual
## MidPoint Community Meetup

Evolveum

Funded by the European Union
NextGenerationEU

RECOVERY AND RESILIENCE PLAN

# Why did you not used Claude Code / Codex Approach?

- Our Development started in April 2025
- Claude Code started being usable in October – November 2025
- But mainly:
  - Claude Code is good for ad-hoc development
  - Pipeline / structured approach is great for repetitive tasks
  - Validations / test suites can be defined before