Developers Conference 2026

# MAKING AN API FOR BUILDING APIs

JULY 2026

From First Line to Production: The Technical Journey
Behind Building a Complete API Platform

# CONTENTS

Founder & Lead Developer - Kinesis API

# Kishan Takoordyal

Professionally, I act as the Infrastructure and Support Team Leader at **Esokia**. I also develop games, APIs and websites during my free time.

I have authored a book on Unity, and have participated in numerous contests while contributing to Open Source.

*"Nothing is Impossible" - Unknown*

# WHAT I AM TRYING TO SOLVE?

Modern API development is fragmented and complex:

## Multiple Tools

Separate systems for API logic, databases, CMS, user management, localization...

## Slow Iteration

Backend changes require code compilation, testing, and redeployment

## External Dependencies

Reliance on third-party databases creates deployment complexity and failure points

## Steep Learning Curve

Teams need expertise across multiple technologies and frameworks

## Configuration Hell

Coordinating different services, versions, and configurations

# WHAT IS KINESIS API?

An all-in-one framework for building, managing, and scaling APIs

## X Engine

Build complete APIs without writing backend code through an intuitive flow-based system. Design custom workflows using drag-and-drop blocks, processors, and conditional logic. Create complex data transformations, validation rules, and API integrations visually—then deploy instantly. Perfect for rapid prototyping and production systems alike.

## Kinesis DB

A custom-built, high-performance ACID-compliant database engine developed from scratch in Rust. Features multiple storage engines (In-Memory, On-Disk, Hybrid), memory-mapped I/O for ultra-fast operations, and write-ahead logging for crash recovery. Zero external database dependencies means simplified deployment and reduced failure points.

## Complete Tooling

Production-ready capabilities built-in: full content management system with Markdown blogging, multi-language localization with variable interpolation, role-based user authentication with 2FA/OTP, media management, automated backups with compression, version control for all content, advanced search with custom query language, component analytics, ticketing system with email notifications, and interactive REPL for real-time testing.

# Technology Stack

## Backend

### Built for Performance & Safety

**Core Framework:**

- Rust - Memory-safe systems programming language
- Rocket - Type-safe, ergonomic web framework
- Tera - Powerful templating engine (Jinja2-inspired)

**Database & Storage:**

- Kinesis DB - Custom ACID-compliant embedded database
- Diesel ORM - Type-safe SQL query builder (for external DB support)
- Memory-mapped I/O - Ultra-fast data access

**API & Integration:**

- utoipa - OpenAPI documentation generation
- Lettre - Email delivery system
- MaxMind GeoIP2 - Geographic IP lookup
- Serde - High-performance serialization

**Infrastructure:**

- Tokio - Async runtime for concurrent operations
- Cryptography libraries - bcrypt, JWT, 2FA/OTP
- Compression - gzip, tar for backups

## Frontend

### Modern, Minimal, Fast

**Styling & UI:**

- Tailwind CSS - Utility-first CSS framework
- DaisyUI - Component library
- Remix Icon - Open-source icon set
- Jost & Montserrat - Clean, professional typography

**Content & Rendering:**

- Markdown-it - Extensible Markdown parser
- PrismJS - Lightweight syntax highlighting
- MermaidJS - Diagram and flowchart generation

**Data Visualization:**

- Chart.js - Interactive charts with Luxon adapter
- Leaflet - Map visualization
- JMESPath - JSON query language

# WHY RUST?

## SPEED

Zero-cost abstractions: No runtime overhead

Compiled to native code: Blazing fast execution

Memory-mapped I/O and direct system access

## CONCURRENCY

Fearless concurrency through ownership system

Tokio async runtime for handling thousands of connections

Thread-safe by default

## SAFETY

Memory safety without garbage collection

No null pointer exceptions, no data races

Compile-time error detection prevents runtime crashes

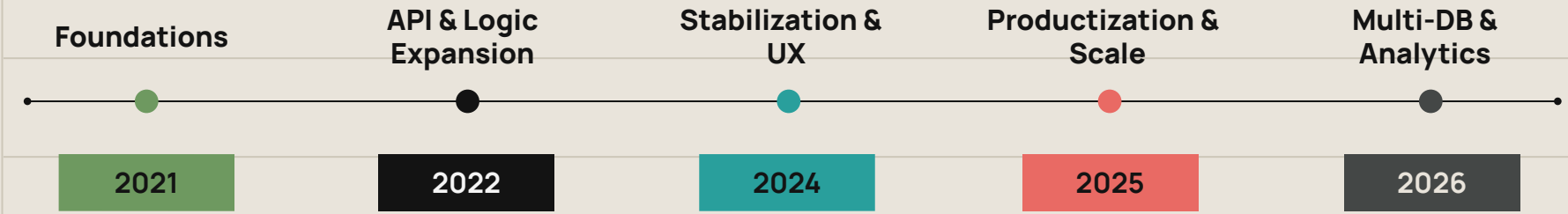
## REFACTORING

Compiler catches breaking changes before runtime

Type system ensures correctness across the entire codebase

Refactor with confidence—no hidden bugs or memory issues

# DEVELOPMENT TIMELINE




Idea phase — simple custom DB using .txt files

Turned into a real Rust API with auth, routes, and logic

Stabilized system + added web UI and better storage

Product phase — Kinesis DB, playground, docs, features

Scale phase — analytics, metrics, multi-database support

# FOUNDATIONS

2021

# Foundations: Core Components

Initially, the project was just to be a specialized & fast database in Rust and the API layer was meant to be coded in a framework on top of NodeJS.

The first core components of the project were created to represent data.

```rust
#[derive(Default, Debug, Clone)]
pub struct Mapping {
    id: String,
    file_name: String,
}
```

```rust
#[derive(Default, Debug, Clone)]
pub struct Config {
    pub name: String,
    pub value: String,
}
```

```rust
#[derive(Default, Debug, Clone)]
pub struct CustomStructure {
    pub id: String,
    name: String,
    structures: Vec<Structure>,
}
```

```rust
#[derive(Debug, Clone)]
pub enum Role {
    ROOT,
    ADMIN,
    AUTHOR,
}

impl Default for Role {
    fn default() -> Self {
        Role::AUTHOR
    }
}

#[derive(Default, Debug, Clone)]
pub struct User {
    pub id: String,
    first_name: String,
    last_name: String,
    username: String,
    email: String,
    password: String,
    role: Role,
}
```

```rust
#[derive(Default, Debug, Clone)]
pub struct Project {
    id: String,
    name: String,
    description: String,
    api_path: String,
}
```

```rust
#[derive(Default, Debug, Clone)]
pub struct Collection {
    id: String,
    project_id: String,
    name: String,
    description: String,
    structures: Vec<Structure>,
    custom_structures: Vec<CustomStructure>,
}
```

```rust
#[derive(Default, Clone, Debug)]
pub struct EncryptionKey(pub String);
```

```rust
#[derive(Debug, Clone)]
pub enum Type {
    TEXT,
    EMAIL,
    PASSWORD,
    RICHTEXT,
    NUMBER,
    ENUM,
    DATE,
    MEDIA,
    BOOLEAN,
    UID,
    JSON,
    CUSTOM(String),
}

impl Default for Type {
    fn default() -> Self {
        Type::TEXT
    }
}

#[derive(Default, Debug, Clone)]
pub struct Structure {
    pub id: String,
    name: String,
    stype: Type,
    default: String,
    min: usize,
    max: usize,
    encrypted: bool,
    unique: bool,
    regex_pattern: String,
    array: bool,
}
```

# Foundations: Mappings

Mappings were a special type of component that served as a K/V store representation of components and the .txt file that was used for data storage.

All Components had logic to add separators between each field and an associated fetch_all_xyz and save_all_xyz functions.

```rust
pub fn fetch_all_configs(path: String, encryption_key: &String) -> Vec<Config> {
    let all_configs_raw = fetch_file(path.clone(), encryption_key);

    let individual_configs = all_configs_raw
        .split("\n")
        .filter(|line| line.chars().count() >= 3);

    let mut final_configs: Vec<Config> = Vec::<Config>::new();

    for config in individual_configs {
        let current_config = config.split("|").collect::<Vec<&str>>();

        let tmp_config = Config::create_no_check(current_config[0], current_config[1]);
        final_configs.push(tmp_config);
    }

    final_configs
}
```

```rust
pub fn save_all_configs(configs: &Vec<Config>, path: String, encryption_key: &String) {
    let mut stringified_configs = String::new();

    for config in configs {
        stringified_configs = format!(
            "{}{}{}|{}",
            stringified_configs,
            if stringified_configs.chars().count() > 1 {
                "\n"
            } else {
                ""
            },
            config.name,
            config.value,
        );
    }

    save_file(path, stringified_configs, encryption_key);
    println!("Configs saved!");
}
```

# Foundations: Redeeming Features

- Unit tests since the beginning for all components.
- Password hashing for users.
- Regexes in use for validating email addresses.
- Initializers for creating sample data on boot.
- Introduction of a CI/CD pipeline to test and deploy after each new commit.
- Encrypting sensitive data such as config values.

```rust
    pub fn encrypt(data: String, key: &str) -> String {
        let mc = new_magic_crypt!(key, 256);
        let ciphertext = mc.encrypt_str_to_base64(data);

        ciphertext
    }

    pub fn decrypt(data: String, key: &str) -> Result<EncryptionKey, String> {
        let mc = new_magic_crypt!(key, 256);
        let original_data = mc.decrypt_base64_to_string(&data);

        if let Err(e) = original_data {
            return Err(e.to_string());
        }

        Ok(EncryptionKey(original_data.unwrap()))
    }
}
```

```rust
pub fn initialize_encryption_key(
    mappings: &Vec<Mapping>,
    password: &str,
) -> Result<String, String> {
    let encryption_key_path = get_file_name("encryption_key", mappings);
    let encryption_key: Result<String, String>;

    if let Err(e) = encryption_key_path {
        return Err(e);
    }

    encryption_key = fetch_encryption_key(encryption_key_path.clone().unwrap(), password);

    if let Err(_) = encryption_key {
        // Encryption key most likely doesn't exist yet
        let new_encryption_key = EncryptionKey::generate(20);
        let saved_encryption_key = save_encryption_key(
            new_encryption_key.0.clone(),
            password,
            &*encryption_key_path.unwrap(),
        );

        if let Err(f) = saved_encryption_key {
            return Err(String::from(f));
        }

        println!("Encryption Key Saved!");

        return Ok(new_encryption_key.0);
    }

    Ok(encryption_key.unwrap())
}
```

# Foundations: WASM

The API layer was meant to interact with Kinesis DB using WASM.

All functions and components needed to have separate binding functions.

**Problem**: It just created yet another unneeded layer and unnecessary complexity. Bridging a JS backend to another Rust backend didn't feel noticeably faster either.

```
20    "dependencies": {
21      "kinesis-db": "file:../pkg"
22    },
```
package.json

```
1  import { fetch_users } from '../pkg/kinesis_db.js';
2
3  let users: any = fetch_users(
4    '/home/edgeking810/Documents/Rust/kinesis-db/data/users.txt',
5    'OhgENBmdz38JOCsRYehi53YCLQFbj99x6ua/dxI5kc8='
6  );
7  console.log(users);
```

```
45  #[wasm_bindgen]
46  pub fn create_config(
47      configs: &str,
48      name: &str,
49      value: &str,
50  ) -> String {
51      let mut all_configs = get_configs_from_str(configs);
52      let result = Config::create(&mut all_configs, name, value);
53
54      if let Err(e) = result {
55          return e;
56      }
57
58      String::from("OK")
59  }
60
61  #[wasm_bindgen]
62  pub fn update_config_value(
63      configs: &str,
64      name: &str,
65      value: &str,
66  ) -> String {
67      let mut all_configs = get_configs_from_str(configs);
68      let result = Config::update_value(&mut all_configs, name, value);
69
70      if let Err(e) = result {
71          return e;
72      }
73
74      String::from("OK")
75  }
76
77  #[wasm_bindgen]
78  pub fn delete_config(
79      configs: &str,
80      name: &str,
81  ) -> String {
82      let mut all_configs = get_configs_from_str(configs);
83      let result = Config::delete(&mut all_configs, name);
84
85      if let Err(e) = result {
86          return e;
87      }
88
89      String::from("OK")
90  }
```

# API & LOGIC EXPLOSION

## 2022

# API & Logic Explosion: Rocket

- Why not build the API layer in Rust itself?
- Added serde for the [de]serialization of components.
- Introduced middlewares for creating and verifying JWTs.

```rust
pub fn create_jwt(mappings: &Vec<Mapping>, uid: String) -> Result<String, String> {
    let expire = match get_config_value(mappings, "JWT_EXPIRE", "60").parse::<i64>() {
        Ok(val) => val,
        _ => 60,
    };
    let secret = get_config_value(mappings, "TOKEN_KEY", "secret");

    let expiration = Utc::now()
        .checked_add_signed(chrono::Duration::seconds(expire))
        .expect("Valid Timestamp")
        .timestamp();

    let claims = Claims {
        uid: uid.to_owned(),
        exp: expiration as usize,
    };

    let header = Header::new(Algorithm::HS512);

    return match encode(
        &header,
        &claims,
        &EncodingKey::from_secret(secret.as_bytes()),
    ) {
        Ok(jwt) => Ok(jwt),
        Err(_) => Err(String::from("Error: Failed to create JWT")),
    };
}
```

```rust
pub async fn verify_jwt(uid: String, token: String) -> Result<String, (usize, String)> {
    let mappings = auto_fetch_all_mappings();
    let secret = get_config_value(&mappings, "TOKEN_KEY", "secret");
    let users = match auto_fetch_all_users(&mappings) {
        Ok(u) => u,
        _ => {
            return Err((500, String::from("Error: Failed fetching users")));
        }
    };

    if !User::exist(&users, &uid) {
        return Err((404, String::from("Error: Account with this uid not found")));
    }

    let mut proper_token = "";
    for v in token.split(" ") {
        proper_token = v;
    }

    let decoded = match decode::<Claims>(
        &proper_token,
        &DecodingKey::from_secret(secret.as_bytes()),
        &Validation::new(Algorithm::HS512),
    ) {
        Ok(dec) => dec,
        Err(e) => {
            return Err((
                500,
                format!("{} ({})", String::from("Error: Failed decoding JWT"), e),
            ));
        }
    };

    if decoded.claims.uid != uid {
        return Err((403, String::from("Error: Incorrect UID")));
    }

    Ok(String::from("Successfully Authenticated!"))
}
```

# API & Logic Explosion: API Routes

- Added routes for fetching, creating, updating and deleting core components.
- Make use of the lettre crate for sending emails (e.g. during user registration process).
- Custom pagination function for fetch routes with offset and limit.

```
55      let all_configs = match auto_fetch_all_configs(&mappings) {
56          Ok(u) => u,
57          _ => {
58              return json!({"status": 500, "message": "Error: Failed fetching configs"});
59          }
60      };
61      let amount = all_configs.len();
62      let processed_configs = paginate(all_configs, passed_limit, passed_offset);
63
64      return json!({"status": 200, "message": "Configs successfully fetched!", "configs": processed_configs, "amount": amount});
65  }
```

# API & Logic Explosion: New Components

New components were introduced along with tests, routes and utils.

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize)]
pub struct Constraint {
    pub component_name: String,
    pub properties: Vec<ConstraintProperty>,
}
```

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize)]
pub struct ConstraintProperty {
    pub property_name: String,
    pub is_alphabetic: bool,
    pub is_numeric: bool,
    pub min: usize,
    pub max: usize,
    pub not_allowed: Vec<char>,
    pub additional_allowed: Vec<char>,
}
```

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize)]
pub struct Data {
    pub id: String,
    pub project_id: String,
    pub collection_id: String,
    pub pairs: Vec<DataPair>,
    pub published: bool,
}
```

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize)]
pub struct DataPair {
    pub id: String,
    pub structure_id: String,
    pub custom_structure_id: String,
    pub value: String,
    pub dtype: String,
}
```

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, PartialEq)]
pub struct RawPair {
    pub data_id: String,
    pub structures: Vec<StructurePair>,
    pub custom_structures: Vec<CustomStructurePair>,
    pub published: bool,
}

#[derive(Default, Debug, Clone, Serialize, Deserialize, PartialEq)]
pub struct StructurePair {
    pub id: String,
    pub value: String,
    pub rtype: String,
}

#[derive(Default, Debug, Clone, Serialize, Deserialize, PartialEq)]
pub struct CustomStructurePair {
    pub id: String,
    pub structures: Vec<StructurePair>,
}
```

```rust
#[derive(Default, Debug, Clone, Deserialize, Serialize, PartialEq)]
pub struct Event {
    pub id: String,
    pub event_type: String,
    pub description: String,
    pub timestamp: String,
    pub redirect: String,
}
```

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize)]
pub struct Media {
    pub id: String,
    name: String,
}
```

# API & Logic Explosion: Notable Changes

- Remove wasm since it was too buggy, had an overhead and KDB was behaving like an API for another Backend.
- Codebase restructuring: splitting files into smaller specialized files and group similar files in 'categories' (e.g. components/).
- Introduction of utils: utility functions to simplify recurring operations (such as fetching all 'records' of a component).
- Addition of the 'VIEWER' role for users.
- Custom Dockerfile and enhanced CI/CD for packaging, pushing and deploying to a K8s cluster.
- .lock files for reading/writing from/to disk (I/O) and several utility functions (e.g. creating nested directories).
- New fields for existing core components.
- Caching data using Redis as a POC.
- KDL as a language to represent routing objects.
- REPL as a POC for selecting data.

```rust
use rocket::fairing::{Fairing, Info, Kind};
use rocket::http::Header;
use rocket::{Request, Response};

pub struct CORS;

#[rocket::async_trait]
impl Fairing for CORS {
    fn info(&self) -> Info {
        Info {
            name: "Attaching CORS headers to responses",
            kind: Kind::Response,
        }
    }

    async fn on_response<'r>(&self, _request: &'r Request<'_>, response: &mut Response<'r>) {
        response.set_header(Header::new("Access-Control-Allow-Origin", "*"));
        response.set_header(Header::new(
            "Access-Control-Allow-Methods",
            "GET, POST, OPTIONS",
        ));
        response.set_header(Header::new("Access-Control-Allow-Headers", "*"));
        response.set_header(Header::new("Access-Control-Allow-Credentials", "true"));
    }
}
```

Added a CORS Middleware too although it wasn't amazing.

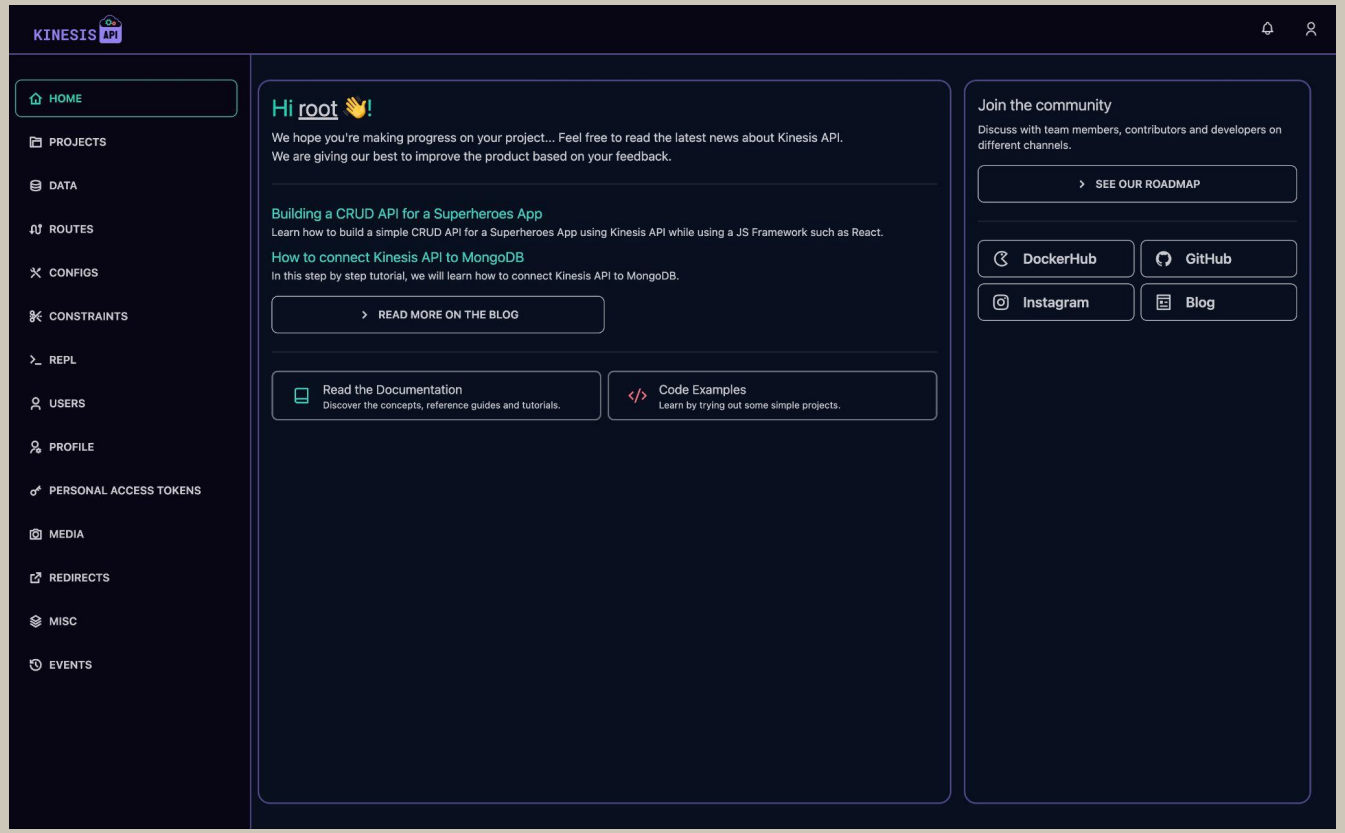
# API & Logic Explosion: Complications

- Complicated logic in I/O to handle file and directory operations.
- Make sure data pairs respect constraints of structures when they're created.
- Logic to delete linked components (e.g. Project → Collections + Structures + Custom Structures).
- Similar code for updating all components when a linked field changes (e.g. collection_id).

```
281  -          if !String::from(description)
282  -              .chars()
283  -              .all(|c| c != ';' && c != '>' && c != '#')
284  -          {
285  -              return Err((
286  -                  400,
287  -                  String::from("Error: description contains an invalid character"),
288  -              ));
289  -          }
290  -
291  -          if String::from(description.trim()).len() < 1 {
292  -              return Err((
293  -                  400,
294  -                  String::from("Error: description does not contain enough characters"),
295  -              ));
296  -          } else if String::from(description.trim()).len() > 400 {
297  -              return Err((
298  -                  400,
299  -                  String::from("Error: description contains too many characters"),
300  -              ));
301  -          }
254  +          let mappings = auto_fetch_all_mappings();
255  +          let all_constraints = match auto_fetch_all_constraints(&mappings) {
256  +              Ok(c) => c,
257  +              Err(e) => return Err((500, e)),
258  +          };
259  +          let final_value = match ConstraintProperty::validate(&all_constraints, "collection", "description", description) {
260  +              Ok(v) => v,
261  +              Err(e) => return Err(e),
262  +          };
```

```
78      match Project::delete(&mut all_projects, &passed_project_id) {
79          Err(e) => return json!({"status": e.0, "message": e.1}),
80          _ => {}
81      }
82
83      let mut all_collections = match auto_fetch_all_collections(&mappings) {
84          Ok(u) => u,
85          _ => {
86              return json!({"status": 500, "message": "Error: Failed fetching collections"});
87          }
88      };
89
90      Collection::delete_by_project(&mut all_collections, &passed_project_id);
91
92      match auto_save_all_collections(&mappings, &all_collections) {
93          Err(e) => return json!({"status": 500, "message": e}),
94          _ => {}
95      }
96
97      if let Err(e) = auto_create_event(
98          &mappings,
99          "project_delete",
100         format!(
101             "The project <{}> was deleted by usr[{}]",
102             project.name, &passed_uid
103         ),
104         format!("/projects"),
105     ) {
106         return json!({"status": e.0, "message": e.1});
107     }
108
109     match auto_save_all_projects(&mappings, &all_projects) {
110         Ok(_) => return json!({"status": 200, "message": "Project successfully deleted!"}),
111         Err(e) => {
112             json!({"status": 500, "message": e})
113         }
114     }
115 }
```

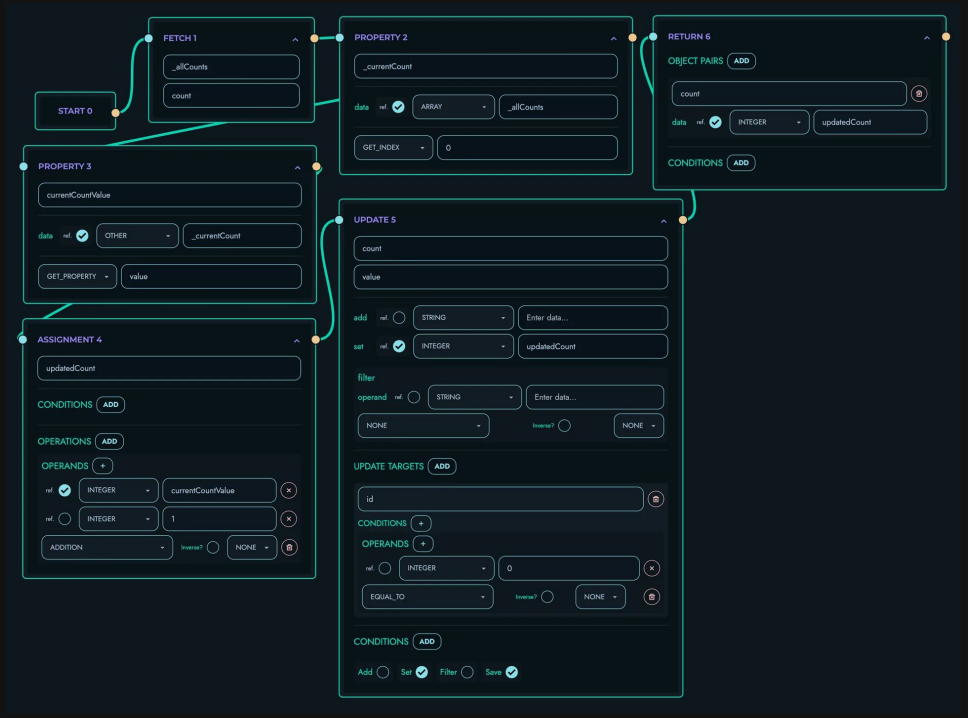
# Kinesis API Portal

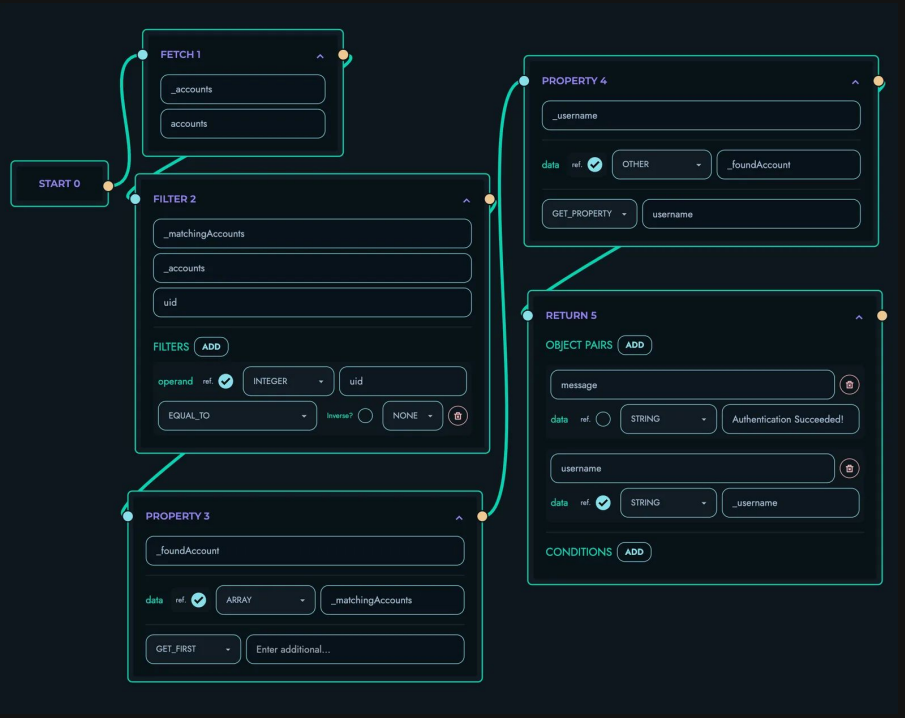



- ReactJS
- Ugly af
- Scaling limitations
- Different codebase
- Replaced by Tera

# The X Engine

The magic behind "the API to make an API".



# X Engine : Components

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, PartialEq)]
pub struct RouteComponent {
    pub route_id: String,
    pub route_path: String,
    pub project_id: String,
    pub auth_jwt: Option<AuthJWT>,
    pub body: Vec<BodyData>,
    pub params: Option<ParamData>,
    pub flow: RouteFlow,
}
```

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, PartialEq)]
pub struct AuthJWT {
    pub active: bool,
    pub field: String,
    pub ref_col: String,
}
```

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, PartialEq)]
pub struct BodyData {
    pub id: String,
    pub bdtype: BodyDataType,
}
```

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, PartialEq)]
pub struct ParamData {
    pub delimiter: String,
    pub pairs: Vec<BodyData>,
}
```

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, PartialEq)]
pub struct RouteFlow {
    pub fetchers: Vec<FetchBlock>,
    pub assignments: Vec<AssignmentBlock>,
    pub templates: Vec<TemplateBlock>,
    pub conditions: Vec<ConditionBlock>,
    pub loops: Vec<LoopBlock>,
    pub filters: Vec<FilterBlock>,
    pub properties: Vec<PropertyBlock>,
    pub functions: Vec<FunctionBlock>,
    pub objects: Vec<ObjectBlock>,
    pub updates: Vec<UpdateBlock>,
    pub creates: Vec<CreateBlock>,
    pub returns: Vec<ReturnBlock>,
}
```

Several modules and submodules required for representing and storing routes created by users.

- body_data_type.rs
- condition_action.rs
- condition_plain.rs
- condition_type.rs
- condition.rs
- fail_obj.rs
- filter.rs
- function_list.rs
- function.rs
- next_condition_type.rs
- object_pair.rs
- operation_type.rs
- operation.rs
- property_apply.rs
- property.rs
- ref_data.rs
- submodules.rs
- update_target.rs

# X Engine : The /x route

- A generic, configurable API gateway used across projects.
- Supports GET, POST, PUT, PATCH, DELETE, HEAD and OPTIONS verbs.
- Centralized 'handle_request' method for orchestrating processing.
- Finds out which project's API path and route is after the "/x/".
- Fetches and caches all relevant data pairs for the target project.
- Validates URL parameters and body parameters that have been passed.
- Checks the JWT authentication mechanism, if enforced.
- GlobalBlockOrder: Obtains an ordered list of blocks for the current route.
- LoopProcessor: Checks for loops and processes them.
- DefinitionStore: Adds definitions for all blocks during block processing.
- Resolvers: Resolve conditions, operations and referenced data during processing.
- SignalProcessor: Returns and parses signals after processing blocks.

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct DefinitionStore {
    pub name: String,
    pub global_index: usize,
    pub block_index: usize,
    pub data: DefinitionData,
}

#[derive(Debug, Clone, Serialize, Deserialize, PartialEq)]
pub enum DefinitionData {
    NULL,
    UNDEFINED,
    BOOLEAN(bool),
    STRING(String),
    INTEGER(isize),
    FLOAT(f64),
    ARRAY(Vec<DefinitionData>),
    DATA(RawPair),
    NoChange,
}
```

```rust
pub struct GlobalBlockOrder {
    pub index: usize,
    pub block_index: usize,
    pub global_index: usize,
    pub name: String,
    pub ref_name: String,
}
```

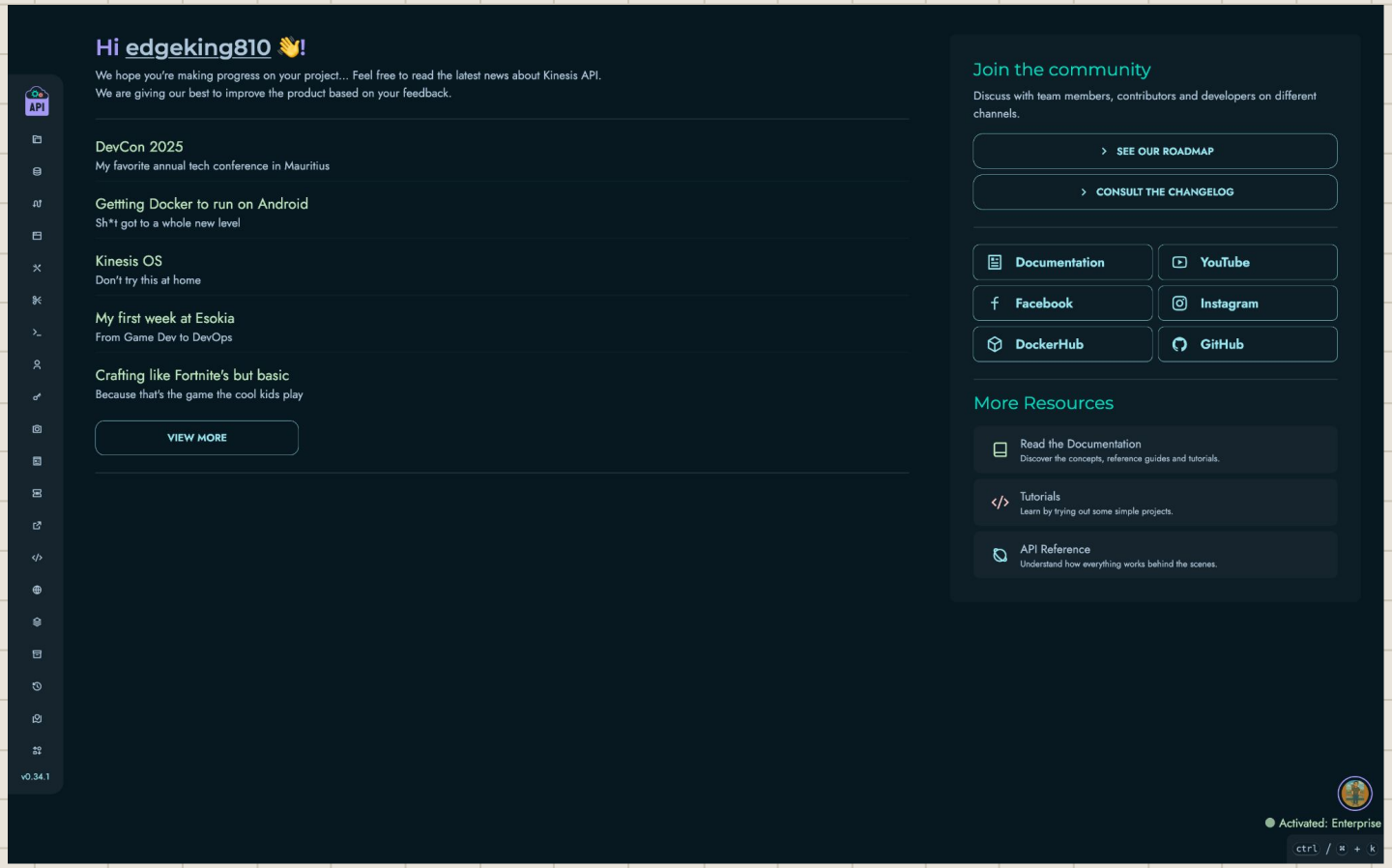
```rust
pub struct LoopObject {
    pub start_index: usize,
    pub end_index: usize,
    pub loop_index: usize,
    pub ref_var: String,
    pub parent: Option<String>,
}
```

```rust
pub enum Signal {
    NONE,
    BREAK,
    CONTINUE,
    FAIL(usize, String),
    RETURN(Value),
}
```

# STABILIZATION & UX

## 2024

# Stabilization & UX: Web UI




New Web UI built using Tera templates along with DaisyUI.

All components could now be managed from the Web UI itself.

# Stabilization & UX: I/O Performance

```rust
pub fn construct_block(properties: HashMap<String, String>) -> String {
    let mut block = String::new();

    for (k, v) in properties {
        let new_value = v.replace("\n", "0x0a").replace("==========", "");
        block.push_str(&format!("${}={}\n", k, new_value));
    }

    block
}

pub fn construct_blocks(blocks: Vec<String>, property: Option<String>) -> String {
    let mut final_block = String::new();
    let _property = match property {
        Some(p) => p.clone(),
        None => String::new(),
    };

    if _property.len() > 0 {
        final_block.push_str(&format!("<{}=\n", _property));
    }

    for (i, b) in blocks.iter().enumerate() {
        final_block.push_str(&format!("{}", b));
        if i < blocks.len() - 1 {
            if _property.len() > 0 {
                final_block.push_str(&format!("__{}__\n", _property));
            } else {
                final_block.push_str("$==================$\n");
            }
        }
    }

    if _property.len() > 0 {
        final_block.push_str(&format!("{}>\n", _property));
    }

    final_block
}

// <property=
// __property__
// property>
```

Replace Redis POC by a custom built Memory DB: Kasper V2 Engine.

Add I/O BlockEngine for saving components in a more generic format.

```rust
pub fn get_memory_db(key: String) -> Option<String> {
    unsafe {
        match memory_db(key, None) {
            Some(v) => Some(v),
            None => None,
        }
    }
}

pub fn set_memory_db(key: String, value: String) -> String {
    unsafe {
        match memory_db(key, Some(value)) {
            Some(v) => v,
            None => String::new(),
        }
    }
}
```

```rust
pub unsafe fn memory_db(key: String, value: Option<String>) -> Option<String> {
    static mut MEMORY_DB: OnceLock<Mutex<HashMap<String, String>>> = OnceLock::new();

    let handle =
        thread::spawn(
            move || match MEMORY_DB.get_or_init(|| Mutex::new(HashMap::new())).lock() {
                Ok(mut guard) => {
                    let mem_db = guard.deref_mut();

                    match value {
                        Some(v) => {
                            mem_db.insert(key, v);
                            mem_db.insert("_return".to_string(), String::from("OK!"));
                        }
                        None => match mem_db.get(&key) {
                            Some(v) => {
                                mem_db.insert("_return".to_string(), v.clone());
                            }
                            None => {
                                mem_db.insert("_return".to_string(), String::new());
                            }
                        },
                    }
                }
                _ => println!("Failed acquiring memory_db lock."),
            },
        );

    match handle.join() {
        Ok(_) => match MEMORY_DB
            .get()
            .unwrap()
            .lock()
            .unwrap()
            .borrow()
            .get("_return")
        {
            Some(v) => {
                if v == "" {
                    return None;
                }
                return Some(v.clone());
            }
            None => None,
        },
        Err(_) => {
            println!("Failed to join memory_db thread.");
            return None;
        }
    }
}
```

# Stabilization & UX: New Components

New components: UUID, PersonalAccessToken, Redirect & Snippet

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, PartialEq, ToSchema)]
pub struct PersonalAccessToken {
    /// A unique auto-generated identifier.
    #[schema(example = "39b6f437-7f7b-44d5-8960-5e76be09400e")]
    pub id: String,
    /// An associated name or message with the personal access token.
    #[schema(example = "test pat")]
    name: String,
    /// The owner of the personal access token (user.id).
    #[schema(example = "15a70484-6684-4eb0-a6d9-6266b2719261")]
    pub owner_id: String,
    /// The timestamp that defines since when the personal access token is valid.
    #[schema(example = "2024-07-21T18:43:00+04:00")]
    valid_from: String,
    /// The timestamp that defines until when the personal access token is valid.
    #[schema(example = "2024-07-21T18:44:00+04:00")]
    pub valid_to: String,
    /// The personal access token itself (hashed).
    #[schema(
        example = "$argon2id$v=19$m=19456,t=2,p=1$yvQn/ekFRoOiom/t1qUEYg$Of8Iy2qagcMM6cxQRbUiSSKBilZ/+tPbJhU3IrLKaNY"
    )]
    pub token: String,
    /// The rights or permissions associated with the personal access token.
    pub rights: Vec<TokenRight>,
}

#[allow(non_camel_case_types)]
#[derive(Debug, Clone, Serialize, Deserialize, PartialEq, ToSchema)]
pub enum TokenRight {
    USER_FETCH,              // ROOT or owner_id == update_id
    USER_CREATE,             // ROOT
    USER_UPDATE,             // owner_id == update_id
    USER_UPDATE_ROLE,        // ROOT
    USER_DELETE,             // ROOT
    CONSTRAINT_FETCH,        // Any
    CONSTRAINT_UPDATE,       // ROOT
    CONFIG_FETCH,            // ROOT
    CONFIG_CREATE,           // ROOT
    CONFIG_UPDATE,           // ROOT
    CONFIG_DELETE,           // ROOT
```

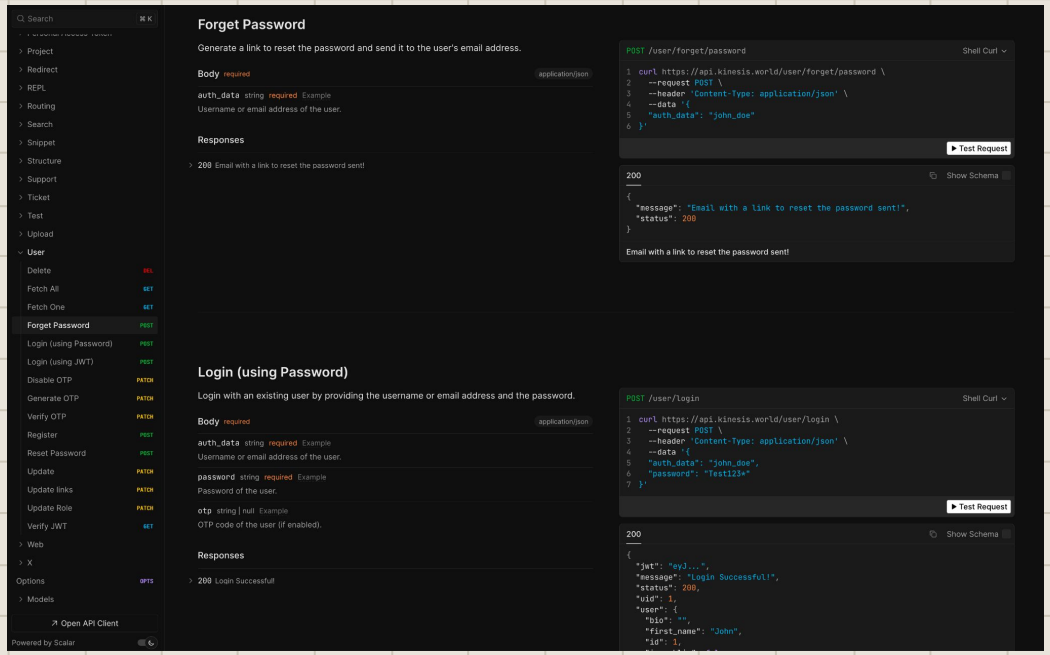
```rust
pub struct Redirect {
    /// The id of the redirect.
    #[schema(example = "abc")]
    pub id: String,
    /// The url of the redirect.
    #[schema(example = "https://api.kinesis.world")]
    pub url: String,
}
```

```rust
pub struct Snippet {
    /// The id of the snippet.
    #[schema(example = "abc")]
    pub id: String,
    /// The uid of the user who created the snippet.
    #[schema(example = "15a70484-6684-4eb0-a6d9-6266b2719261")]
    pub uid: String,
    /// The name of the snippet.
    #[schema(example = "test_snippet")]
    pub name: String,
    /// A description for the snippet.
    #[schema(example = "Just a snippet for testing purposes.")]
    pub description: String,
    /// Whether the snippet should be publicly visible or not.
    #[schema(example = false)]
    pub is_public: bool,
    /// A timestamp that defines the lifetime of the snippet.
    #[schema(example = "2024-07-21T18:44:00+04:00")]
    pub expiry: Option<String>,
    /// The actual content of the snippet.
    #[schema(example = "# Hello World!")]
    pub content: String,
}
```

# Stabilization & UX: General Improvements

- Move to gitea instead of gitlab.
- Improve docker build and release process.
- X Engine refactoring.
- Utilization of cookies for the /web endpoint.
- Removal of the REPL feature.
- Improve SMTP logic.
- Add /metrics route for prometheus.
- Initialize the roadmap page on the Web UI.
- Introduction of load testing using k6.
- Add pagination for components on the Web UI.
- New route for resetting user passwords.

- New middleware for validating rights through PAT and JWT.
- API Documentation using utoipa and Scalar.
- Reduce calls to auto_fetch_xxx functions.
- Use create_no_check functions when reading existing data.
- Bug fixes.

# Stabilization & UX : X Engine

- Add support for resolving multiple operands in conditions and operations.
- Redesign functionality of the DefinitionStore.
- Updated block: LoopBlock
- New block: EndLoopBlock

```rust
pub struct LoopBlock {
    pub global_index: u32,
    pub block_index: u32,
    pub local_name: String,
    pub start: RefData,
    pub end: RefData,
    pub step: Option<RefData>,
    pub include_last: bool,
}
```

```rust
pub struct EndLoopBlock {
    pub global_index: u32,
    pub block_index: u32,
    pub local_name: String,
}
```

```rust
pub fn detect_loops(
    global_blocks: &Vec<GlobalBlockOrder>,
    current_block: &GlobalBlockOrder,
) -> Result<Vec<LoopObject>, (usize, String)> {
    let mut loop_objects = Vec::<LoopObject>::new();

    let block_index = current_block.block_index;
    let mut start_index = 0;
    let mut is_processing_block = false;
    let mut loop_var = String::new();

    // Loop through all the blocks in global blocks
    for (i, block) in global_blocks.iter().enumerate() {
        // If a loop is being processed, and another loop is found or another block_index has
        // been reached, the current loop is completed.
        if is_processing_block && (block.name == "LOOP" || block.block_index > block_index) {
            loop_objects.push(LoopObject {
                start_index: start_index,
                end_index: i,
                ref_var: loop_var.clone(),
            });
            is_processing_block = false;
            start_index = 0;
        }

        // If a loop is found, set is_processing to true in order to process it and whatever
        // other blocks are found in it.
        if block.block_index == block_index && block.name == "LOOP" {
            is_processing_block = true;
            start_index = i;
            loop_var = block.ref_name.clone();
        }
    }

    // If is_processing is true, it means that the last loop was not closed yet, and since
    // the last index of global_blocks has been reached, it means that the last loop should
    // end at the end of the global_blocks.
    if is_processing_block {
        loop_objects.push(LoopObject {
            start_index: start_index,
            end_index: global_blocks.len(),
            ref_var: loop_var,
        });
    }

    Ok(loop_objects)
}
```

# PRODUCTIZATION & SCALE

2025

# Productization & Scale : Kinesis DB

## Core Capabilities

- Multiple Storage Engines: In-Memory (volatile/fast), On-Disk (persistent/durable), and Hybrid (balanced performance with persistence).
- Full ACID Compliance: Atomicity, Consistency, Isolation (4 levels), and Durability with Write-Ahead Logging (WAL).
- Dynamic Schema Management: Runtime schema creation/modification with strong typing, constraints (required, unique, patterns, min/max), and versioning.
- Robust Transactions: Multi-level isolation control, deadlock detection/prevention, and automatic crash recovery.
- Performance Optimizations: LRU buffer pool, configurable caching, automatic blob storage for large data, and efficient indexing strategies.
- Interactive REPL Shell: SQL-inspired query syntax with multiple output formats (standard, JSON, table).

## Integration & Modernization

- Seamless Kinesis API Integration: Replace legacy .txt file mapping with unified database backend across all components.
- Eliminate Technical Debt: Remove ad-hoc file handling logic and consolidate data management into a single, reliable system.
- BlobStore for Large Data: Efficient storage and retrieval of large binary/string objects with reference counting and garbage collection.
- Configurable Startup: Full environment variable control for storage engine selection, isolation levels, buffer pool sizing, and recovery policies.

## Enhanced Query & Search

- Improved REPL Search Capabilities: Add –case-insensitive flag for flexible pattern matching across records.
- Advanced Query Support: Equality, range, pattern matching, and full-text search capabilities.
- Multiple Output Formats: Support JSON, table, and standard output for different use cases.

# Productization & Scale : Kinesis DB

## Infrastructure Improvements

- Overflow Pages for Large Data: Efficient handling of records spanning multiple pages with linked page chains.
- Memory-Mapped I/O (memmap2): Leverage OS-level memory mapping for faster, low-latency disk operations.
- Page Format Optimization: Structured 16KB page layout with checksums, overflow chain support, and atomic writes.
- Automatic Compaction: Reclaim fragmented space and optimize disk layout over time.
- Configurable Tuning: Buffer pool size, WAL rotation thresholds, and blob storage thresholds all adjustable.

## Performance & Reliability

- 4X+ Performance Boost: Load testing confirms significant throughput improvements over legacy systems.
- Zero Data Loss Guarantees: WAL ensures committed data survives crashes; automatic recovery on restart.
- Concurrent Transaction Support: Record-level locking with deadlock detection enables parallel operations.
- Scalable Architecture: Modular design supports future enhancements (distributed transactions, advanced indexing, encryption).
- Comprehensive Testing: Unit, integration, and performance test coverage for all major features.

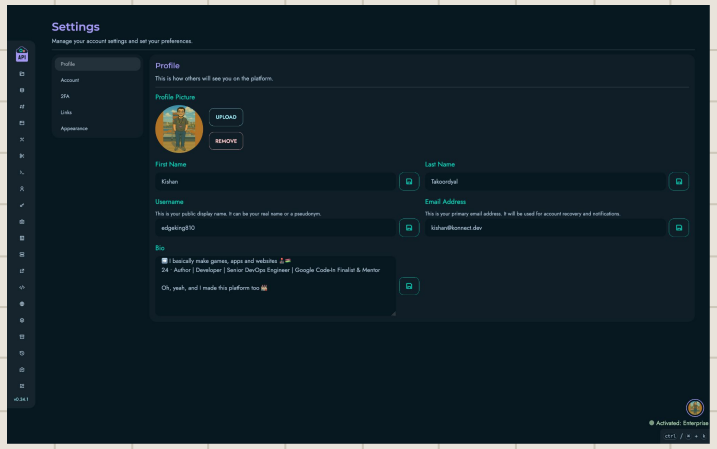
## Developer Experience

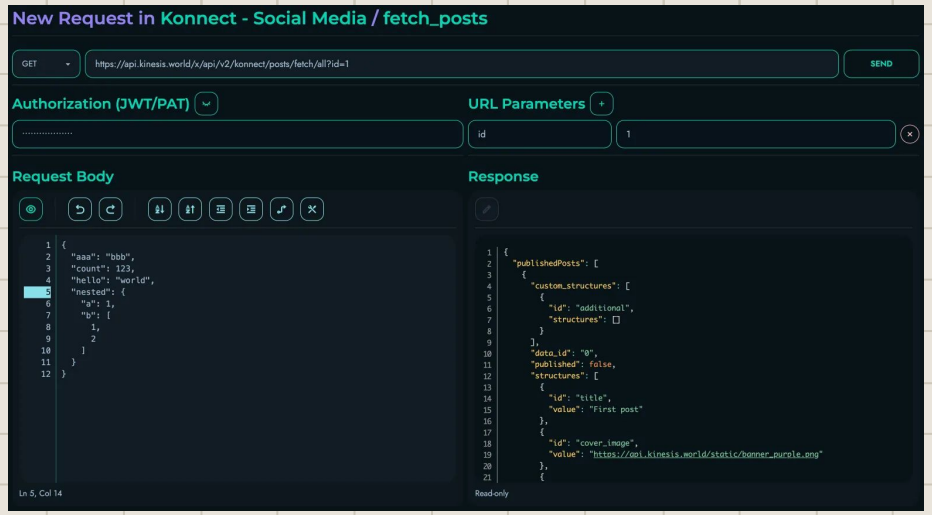
- Simple Configuration: Single environment variable interface for all tuning needs.
- Rich Diagnostics: Logging, checksums, and consistency checks for debugging.
- Extensibility: Support for custom field types, storage backends, and REPL commands.

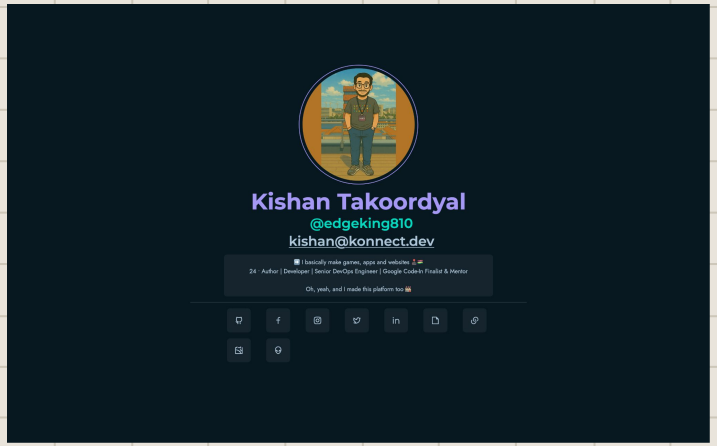
*"True reliability isn't built on hope—it's built on transactions, locks, and write-ahead logging."*

# Productization & Scale: Web-focused changes

- The playground page for testing API routes.
- Addition of the /setup page for first boot of K. API.
- New /about and /changelog pages to document releases.
- Implementation of the /settings and /user pages to allow user customization through bio, social links and profile picture.
- Launch an optional 2FA feature for users.
- Improvements to responsiveness of the Web UI.
- Introduction of the Global Navigate modal.

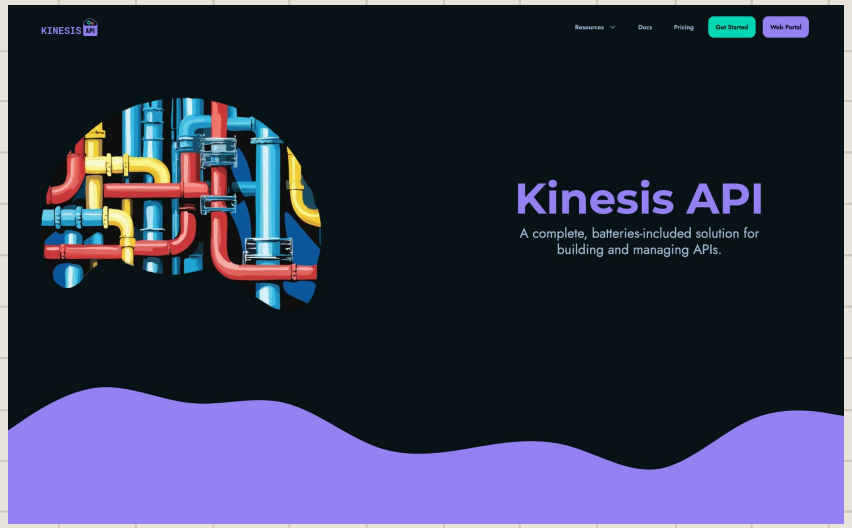


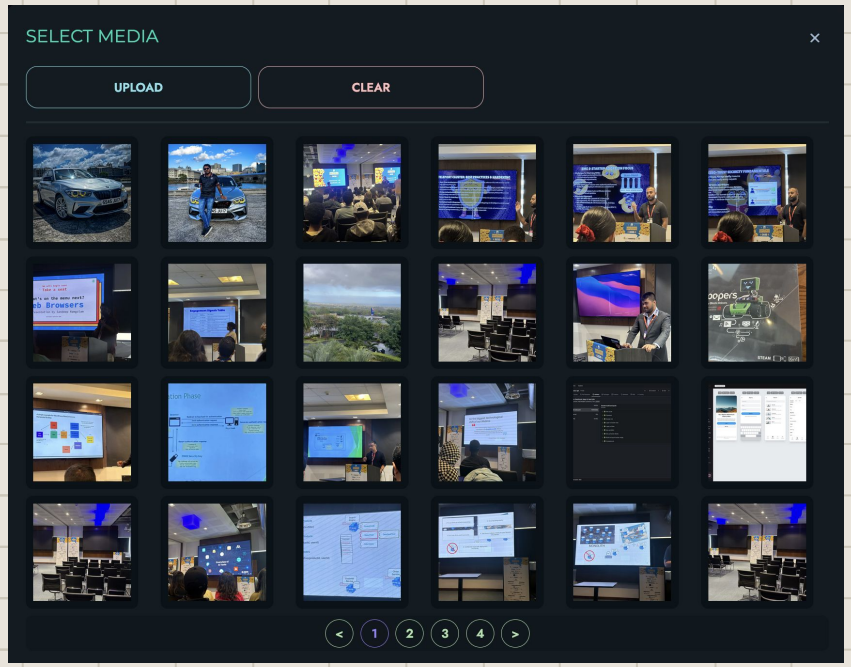



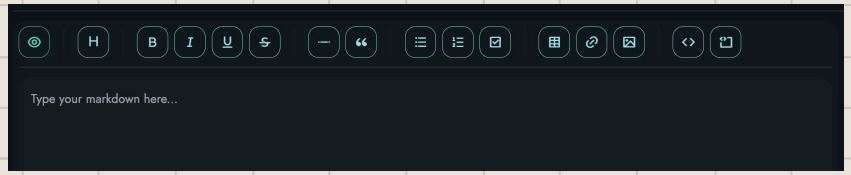



# Productization & Scale: Web-focused changes

- Remake the entire X Engine Routing System UI.
- Phase out the ToastUI Markdown Editor in favor of Markdown IT and other plugins.
- Replace jsoneditor with a custom-made JSON editor.
- SEO improvements & new compression algorithms.
- Addition of the Media Library Popup.
- Introduction of the landing pages including the features, contact, team, pricing, privacy policy, system status, and terms and conditions pages.

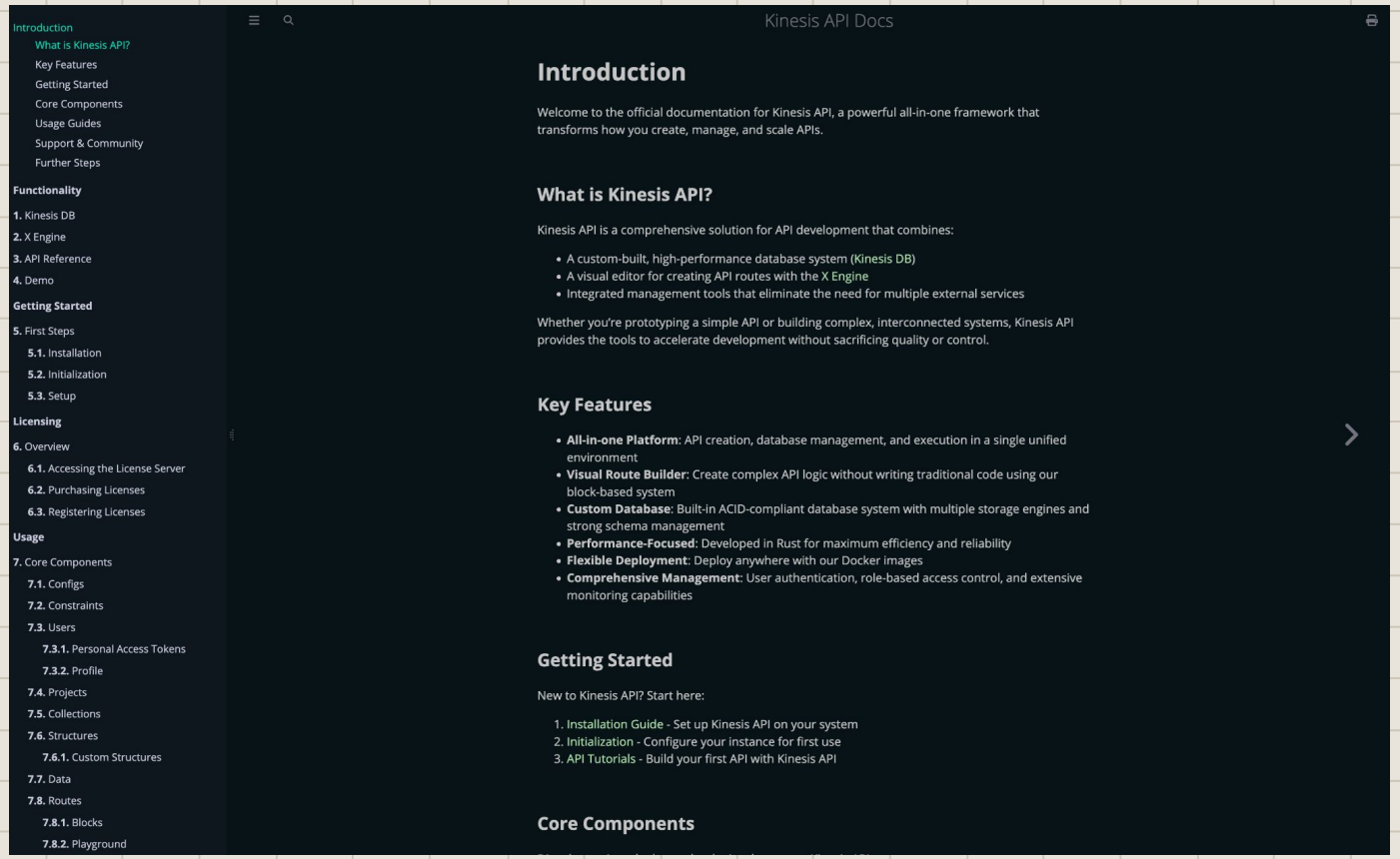


# Productization & Scale: Documentation

Complete documentation using mdbook for all components including tutorials.



# Productization & Scale: Ticketing System

A new ticketing component was introduced along with tests, routes, utils, docs and web pages.

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, ToSchema)]
pub struct Ticket {
    /// A unique identifier for the ticket.
    #[schema(example = 1)]
    pub id: usize,
    /// The title of the ticket.
    #[schema(example = "Example Ticket Title")]
    pub title: String,
    /// The description of the ticket.
    #[schema(example = "This is an example description for the ticket.")]
    pub description: String,
    /// The type of the ticket.
    #[schema(example = "BUG")]
    pub ticket_type: TicketType,
    /// The status of the ticket.
    #[schema(example = "OPEN")]
    pub status: TicketStatus,
    /// The priority of the ticket.
    #[schema(example = "LOW")]
    pub priority: TicketPriority,
    /// The ID of the user who created the ticket.
    #[schema(example = 1)]
    pub user_id: Option<usize>,
    /// An optional field for an anonymous user.
    #[schema(example = "")]
    pub anonymous_user: Option<TicketAnonymousUser>,
    /// Assignees for the ticket.
    #[schema(example = "[]")]
    pub assignees: Vec<usize>,
    /// The date and time when the ticket was created.
    #[schema(example = "2025-06-26T20:45:00+04:00")]
    pub created_at: String,
    /// The date and time when the ticket was last updated.
    #[schema(example = "2025-06-26T20:45:00+04:00")]
    pub updated_at: String,
    /// Whether the ticket is publicly visible.
    #[schema(example = true)]
    pub public: bool,
    /// Whether the ticket is archived.
    #[schema(example = false)]
    pub archived: bool,
    /// A list of email addresses associated with the ticket for notifications.
    #[schema(example = "[]")]
    pub subscribed_emails: Vec<String>,
    /// The tags and labels associated with the ticket.
    #[schema(example = "[]")]
    pub tags: Vec<String>,
    /// A list of comments associated with the ticket.
    #[schema(example = "[]")]
    pub comments: Vec<TicketComment>,
}
```

```rust
#[derive(Debug, Clone, Copy, Serialize, Deserialize, PartialEq, ToSchema)]
pub enum TicketStatus {
    OPEN,
    ACTIVE,
    RESOLVED,
    CLOSED,
    WONTDO,
}
```

```rust
#[derive(Debug, Clone, Serialize, Deserialize, PartialEq, ToSchema)]
pub struct TicketAnonymousUser {
    /// The name of the anonymous user
    #[schema(example = "John Doe")]
    pub name: String,
    /// The email of the anonymous user
    #[schema(example = "john_doe@local.host")]
    pub email: String,
}
```

```rust
#[derive(Debug, Clone, Copy, Serialize, Deserialize, PartialEq, ToSchema)]
pub enum TicketPriority {
    /// Represents a ticket with low priority.
    LOW,
    /// Represents a ticket with medium priority.
    MEDIUM,
    /// Represents a ticket with high priority.
    HIGH,
    /// Represents a ticket with critical priority.
    CRITICAL,
    /// Represents a ticket with urgent priority.
    URGENT,
}
```

```rust
#[derive(Debug, Clone, Copy, Serialize, Deserialize, PartialEq, ToSchema)]
pub enum TicketType {
    /// Represents a bug in the system.
    BUG,
    /// Represents a feature request.
    FEATURE,
    /// Represents an improvement to an existing feature.
    IMPROVEMENT,
    /// Represents a question or inquiry.
    QUESTION,
    /// Represents a task to be completed.
    TASK,
    /// Represents feedback on the system.
    FEEDBACK,
}
```

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, ToSchema)]
pub struct TicketComment {
    /// A unique identifier for the comment.
    #[schema(example = 1)]
    pub id: usize,
    /// The ID of the ticket to which this comment belongs.
    #[schema(example = 1)]
    pub ticket_id: usize,
    /// The content of the comment.
    #[schema(example = "This is a comment on the ticket.")]
    pub content: String,
    /// The ID of the user who created the comment.
    #[schema(example = 1)]
    pub user_id: Option<usize>,
    /// An optional field for an anonymous user.
    #[schema(example = "")]
    pub anonymous_user: Option<TicketAnonymousUser>,
    /// The date and time when the comment was created.
    #[schema(example = "2025-06-26T20:45:00+04:00")]
    pub created_at: String,
    /// The date and time when the comment was last updated.
    #[schema(example = "2025-06-26T20:45:00+04:00")]
    pub updated_at: String,
}
```

# Productization & Scale: Ticketing System

## #1 Background process for sending emails

HIGH | ACTIVE | TASK | PUBLIC

Author: Kishan Takoordyal (kishan@konnect.dev)
Created on: Thu Jul 03 2025 17:11:03
Last updated: Sun Feb 08 2026 12:31:29
Tags: email medium task

ASSIGNEES | SUBSCRIBERS | UNSUBSCRIBE
EDIT DESCRIPTION | ARCHIVE TICKET | DELETE TICKET

### Description

Behind the scenes, the API sends out mails for many events such as user registration, when a password reset is requested, etc. Now, with the implementation of the ticketing system, sending mails happens even more often, since assignees and users subscribed to a ticket need to receive notifications when the ticket is updated.

### The problem

Sending out a mail takes about 5 seconds, and the way this is implemented right now in the API, it all happens synchronously, which means responses from the API take a couple of seconds more to be returned. Further more, if sending out a mail fails, the API will return a 'negative response' even if every other operation in that specific route's logic successfully ran. Sending out mails also depend on having valid SMTP settings configured in the configs, which don't have anything to do with the logic of routes featuring sending mails.

### The solution

Maybe a middleware or an asynchronous function needs to be put in place for sending out mails in the background. Logs and errors while sending mails should be logged somewhere else or appear in the Rocket process' STDOUT/STDERR.

```rust
use rocket::tokio;
use rocket::tokio::task;
use rocket::response::content;

#[rocket::get("/")]
async fn index() -> content::RawHtml<&'static str> {
    // This is a simple example; in real applications, you might
    // be reading from a file or database.
    let message = task::spawn_blocking(|| {
        // Simulate a blocking operation
        std::thread::sleep(std::time::Duration::from_secs(1));
        "Hello from a blocking thread!"
    }).await.unwrap();

    content::RawHtml(format!("<p>{}</p>", message).as_str())
}

#[rocket::main]
async fn main() -> Result<(), rocket::Error> {
    let _rocket = rocket::build().mount("/", rocket::routes![index]).launch().await?;
    Ok(())
}
```

### Resources

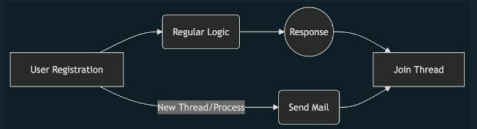
- Scheduled background task using database in Rocket?
- Rocket and scheduled tasks
- Background processes #1640

### Comments

POST A NEW COMMENT

Kishan Takoordyal (kishan@konnect.dev)
Posted on: Thu Jul 03 2025 17:12:25 (edited)

Resolved and released in the `v0.25.0` update




API
v0.34.1

Activated: Enterprise
ctrl / ⌘ + k

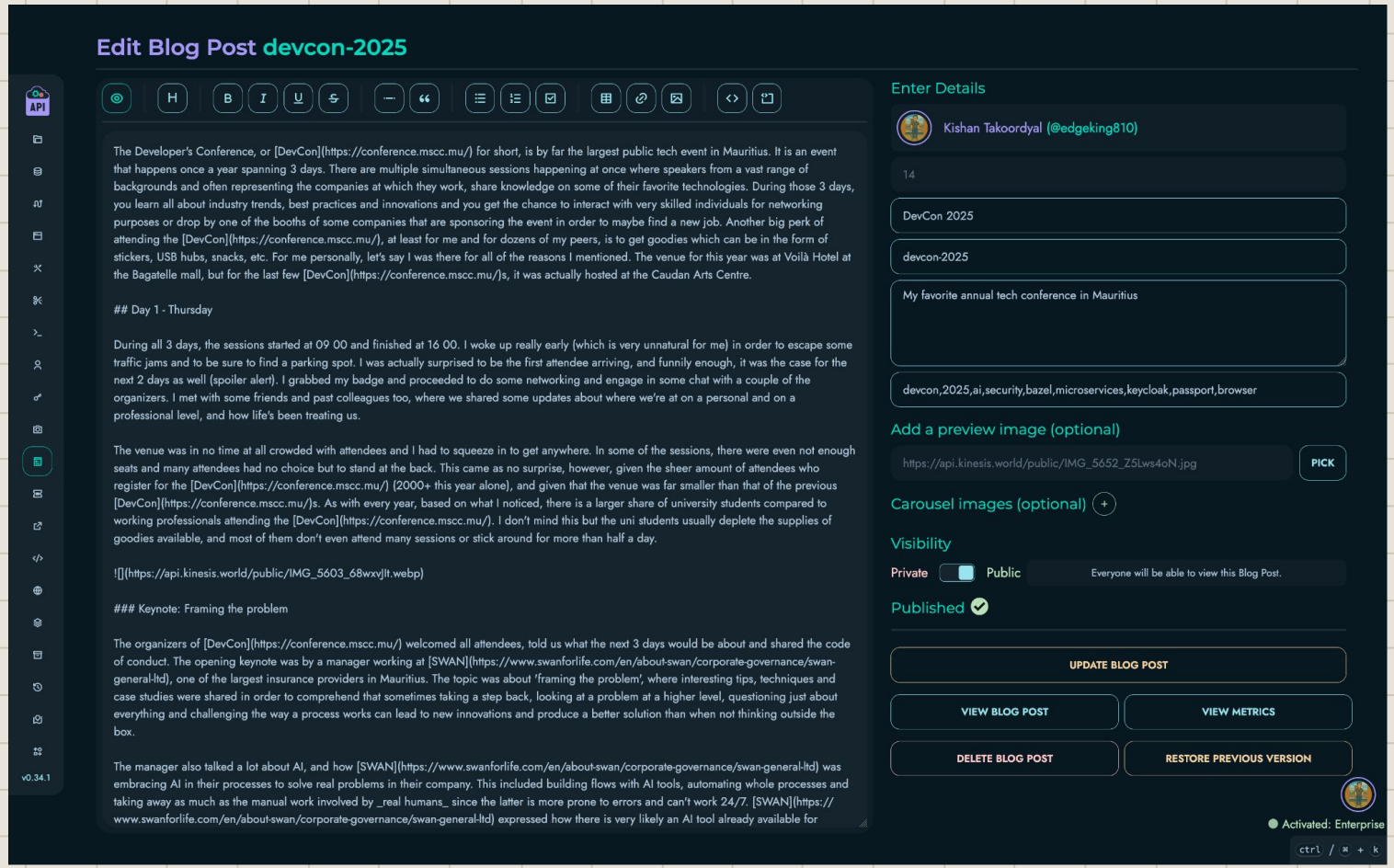
# Productization & Scale: Blog Engine

New blogging components were introduced along with tests, routes, utils, docs and web pages.

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, ToSchema)]
pub struct BlogPost {
    /// A unique identifier for the blog post.
    #[schema(example = 1)]
    pub id: usize,
    /// A slug for the blog post, used in URLs.
    #[schema(example = "example-blog-post")]
    pub slug: String,
    /// The title of the blog post.
    #[schema(example = "Example Blog Post Title")]
    pub title: String,
    /// The subtitle of the blog post.
    #[schema(example = "This is an example subtitle for the blog post.")]
    pub subtitle: String,
    /// A preview image URL for the blog post.
    #[schema(example = "https://example.com/image.jpg")]
    pub preview_image: String,
    /// Images forming part of a carousel for the blog post.
    #[schema(example = "[]")]
    pub carousel_images: Vec<String>,
    /// The content of the blog post.
    #[schema(example = "This is the content of the blog post.")]
    pub content: String,
    /// The ID of the user who created the blog post.
    #[schema(example = 1)]
    pub user_id: usize,
    /// The date and time when the blog post was created.
    #[schema(example = "2025-07-18T17:22:00+04:00")]
    pub created_at: String,
    /// The date and time when the blog post was last updated.
    #[schema(example = "2025-07-18T17:22:00+04:00")]
    pub updated_at: String,
    /// Whether the blog post is publicly visible.
    #[schema(example = true)]
    pub public: bool,
    /// Whether the blog post is published.
    #[schema(example = true)]
    pub published: bool,
    /// The tags and labels associated with the blog post.
    #[schema(example = "[]")]
    pub tags: Vec<String>,
    /// A list of comments associated with the blog post.
    #[schema(example = "[]")]
    pub comments: Vec<BlogComment>,
    /// The amount of views the blog post has received.
    #[schema(example = 0)]
    pub views: usize,
    /// A list of user IDs who have liked the blog post.
    #[schema(example = "[]")]
    pub likes: Vec<usize>,
}
```

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, ToSchema)]
pub struct BlogComment {
    /// A unique identifier for the comment.
    #[schema(example = 1)]
    pub id: usize,
    /// The ID of the blog post the comment belongs to.
    #[schema(example = 1)]
    pub blog_post_id: usize,
    /// The content of the comment.
    #[schema(example = "This is a comment.")]
    pub content: String,
    /// The ID of the user who created the comment.
    #[schema(example = 1)]
    pub user_id: Option<usize>,
    /// An optional field for an anonymous user.
    #[schema(example = "")]
    pub anonymous_user: Option<TicketAnonymousUser>,
    /// The date and time when the comment was created.
    #[schema(example = "2025-07-18T17:22:00+04:00")]
    pub created_at: String,
    /// The date and time when the comment was last updated.
    #[schema(example = "2025-07-18T17:22:00+04:00")]
    pub updated_at: String,
    /// A list of user IDs who have liked the comment.
    #[schema(example = "[]")]
    pub likes: Vec<usize>,
}
```

# Productization & Scale: Blog Engine



# Productization & Scale: Backup Mechanism

New backup, backup schedule and content history components were introduced along with tests, routes, utils, docs and web pages.

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, ToSchema)]
pub struct BackupSchedule {
    /// A unique identifier for the backup schedule.
    #[schema(example = 1)]
    pub id: usize,
    /// The name of the backup schedule.
    #[schema(example = "Daily Backup")]
    pub name: String,
    /// Whether the backup schedule is enabled.
    #[schema(example = true)]
    pub enabled: bool,
    /// The minute of the hour when the backup should be created.
    #[schema(example = "0")]
    pub minute: String,
    /// The hour of the day when the backup should be created.
    #[schema(example = "2")]
    pub hour: String,
    /// The day of the month when the backup should be created.
    #[schema(example = "1")]
    pub day_of_month: String,
    /// The month of the year when the backup should be created.
    #[schema(example = "1")]
    pub month: String,
    /// The day of the week when the backup should be created.
    #[schema(example = "1")]
    pub day_of_week: String,
    /// The amount of hours to keep the backup.
    #[schema(example = 24)]
    pub expiry_hours: usize,
}
```

```rust
#[derive(Debug, Clone, Serialize, Deserialize, ToSchema)]
pub struct ContentHistory {
    /// A unique identifier for the content history.
    #[schema(example = 1)]
    pub id: usize,
    /// The type of content that this history entry is associated with.
    #[schema(example = "DATA")]
    pub content_type: ContentHistoryType,
    /// The id of the content that this history entry is associated with.
    #[schema(example = 1)]
    pub content_id: usize,
    /// The actual content of the history entry.
    #[schema(example = "This is an example content history entry.")]
    pub content: String,
    /// The date and time when the content history was created.
    #[schema(example = "2025-08-15T21:44:00+04:00")]
    pub created_at: String,
}
```

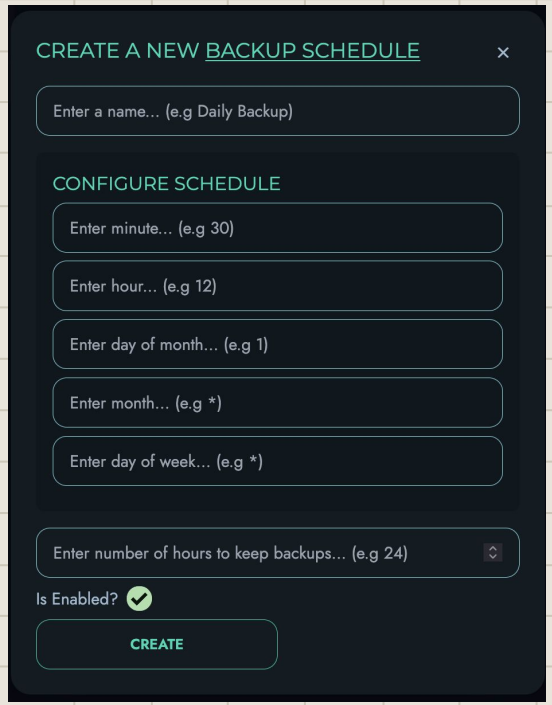
```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, ToSchema)]
pub struct Backup {
    /// A unique identifier for the backup.
    #[schema(example = 1)]
    pub id: usize,
    /// The name of the backup.
    #[schema(example = "example-backup.tar.gz")]
    pub name: String,
    /// The description of the backup.
    #[schema(example = "This is an example backup description.")]
    pub description: String,
    /// The date and time when the backup was created.
    #[schema(example = "2025-08-06T11:54:00+04:00")]
    pub created_at: String,
    /// An optional date and time when the backup should be deleted.
    #[schema(example = "2025-08-07T11:55:00+04:00")]
    pub expiry: Option<String>,
}
```

```rust
#[derive(Debug, Clone, Serialize, Deserialize, ToSchema, PartialEq)]
pub enum ContentHistoryType {
    DATA,
    ROUTE,
    BLOG_POST,
}
```

BACKUP-20260208_0000_1.TAR.GZ SUN FEB 08 2026

Description: Scheduled backup for Daily at Midnight (1)

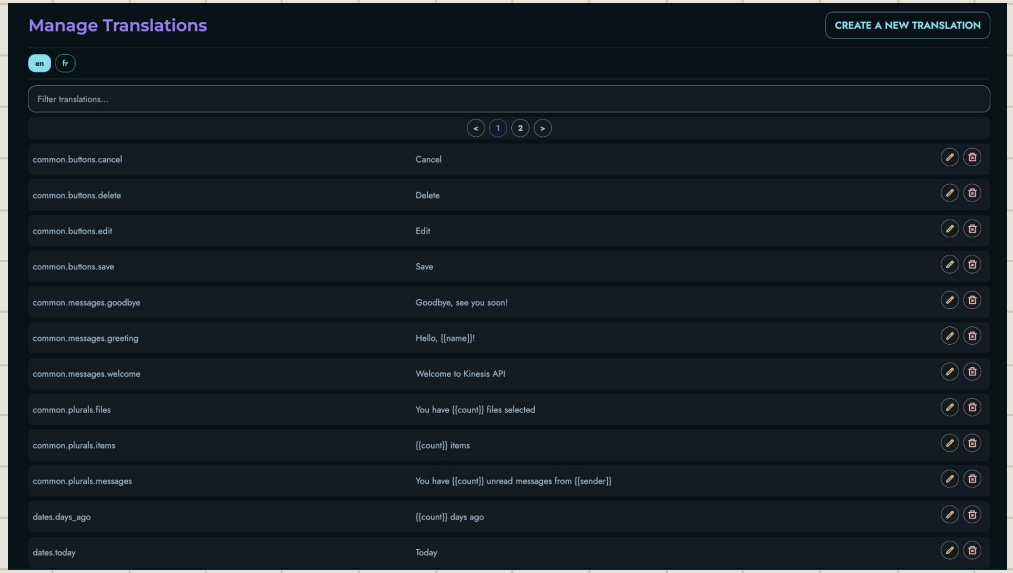
Expiry: 2/10/2026, 4:00:00 AM



# Productization & Scale: Localization Engine

New localization engine component for managing translations for different languages.

```rust
/// Localization engine that manages translations for multiple locales
#[derive(Debug)]
pub struct LocalizationEngine {
    /// Translation data for all loaded locales
    translations: RwLock<HashMap<Locale, HashMap<Key, TranslationValue>>>,
    /// Metadata about loaded locale files for hot reloading
    locale_metadata: RwLock<HashMap<Locale, LocaleMetadata>>,
    /// Default locale to fall back to when translations are missing
    fallback_locale: Locale,
    /// Track missing translation keys for diagnostics
    missing_keys: RwLock<HashSet<String>>,
    /// Pluralization rules for different languages
    plural_rules: HashMap<String, PluralRuleFn>,
    /// Enable auto-reloading of translation files when they change
    auto_reload: bool,
    /// Cache for commonly used translations
    translation_cache: RwLock<HashMap<(String, String, Option<u64>), String>>,
}
```



# Productization & Scale: Search Engine

New search engine component for finding relevant data quicker.

```rust
pub fn fetch_records(engine: &DBEngine, table_name: &str, encryption_key: &str) -> Vec<Record> {
    let mut all_records = Vec::<Record>::new();

    let tables_with_encrypted_data: HashMap<&str, Vec<&str>> = HashMap::from([
        ("config", vec!["value"]),
        ("data_pair", vec!["value"]),
        ("encryption_key", vec!["key"]),
        ("snippet", vec!["content"]),
    ]);

    if let Some(table) = engine.get_table(table_name) {
        for (_, record_lock) in &table.data {
            if let Ok(record) = record_lock.read() {
                all_records.push(record.clone());
            }
        }
    }

    for record in all_records.iter_mut() {
        for (k, v) in record.values.iter_mut() {
            if let ValueType::Str(s) = v {
                let mut value = s.into_string();

                if value.starts_with("[BLOB REF:") && value.contains("(size:") {
                    value = engine.resolve_blob_reference(&value);
                }

                if let Some(encrypted_value) = tables_with_encrypted_data.get(table_name) {
                    if encrypted_value.contains(&k.as_str()) {
                        value = match EncryptionKey::decrypt(value, encryption_key) {
                            Ok(v) => v,
                            Err(_) => String::new(),
                        };
                    }
                }

                *v = ValueType::Str(StringValue::new(value));
            }
        }
    }

    all_records
}
```

```rust
fn fuzzy_match(target: &str, query: &str, max_dist: u8, substring: bool) -> (bool, usize) {
    let mut best_score = 0;
    let mut matched = false;

    if substring {
        let qlen = query.chars().count();
        let tlen = target.chars().count();

        if qlen > tlen {
            let dist = levenshtein(&target, query);
            if dist <= max_dist as usize {
                matched = true;
                best_score = 8 - dist; // higher score for closer match
            }
        } else {
            for i in 0..=(tlen - qlen) {
                let window = target.chars().skip(i).take(qlen).collect::<String>();
                let dist = levenshtein(&window, query);
                if dist <= max_dist as usize {
                    matched = true;
                    let score = 10 - dist;
                    best_score = best_score.max(score);
                }
            }
        }
    } else {
        let dist = levenshtein(&target, query);
        if dist <= max_dist as usize {
            matched = true;
            best_score = 6 - dist; // lower weight for full-field fuzzy
        }
    }

    (matched, best_score)
}
```

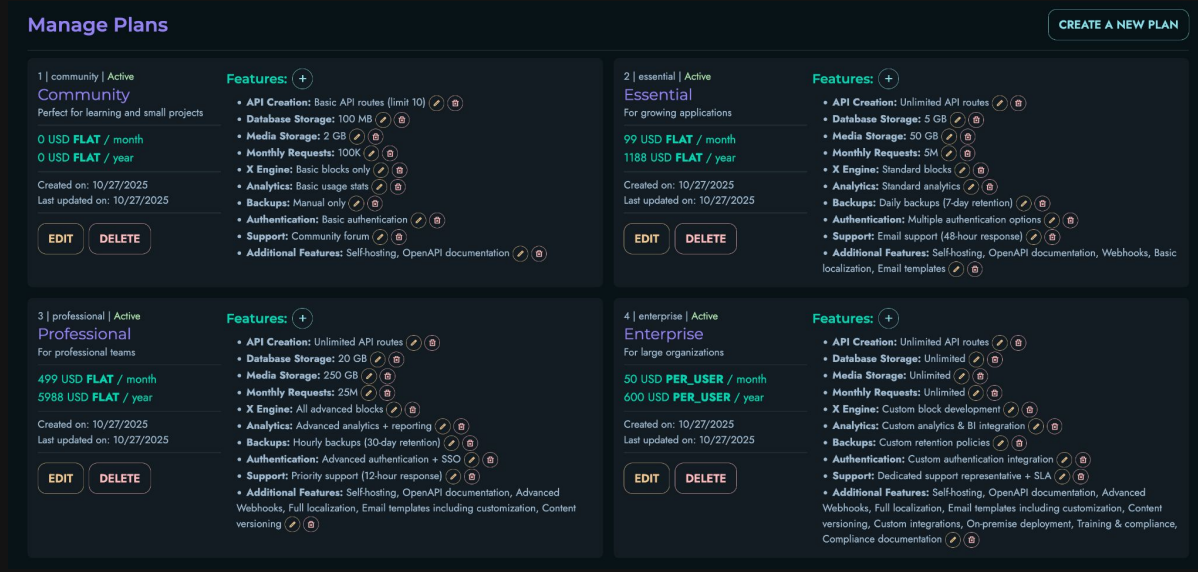
```rust
#[derive(Debug, Clone, PartialEq)]
enum Token {
    // Field identifiers
    Identifier(String),
    Dot,

    // Operators
    OpContains,
    OpEquals,
    OpStartsWith,
    OpEndsWith,
    OpFuzzy,
    OpGreaterThan,
    OpLessThan,
    OpBetween,

    // Logical operators
    And,
    Or,
    Not,

    // Values
    StringLiteral(String),
    NumberLiteral(f64),
    BooleanLiteral(bool),

    // Structure
    LParen,
    RParen,
    Comma,
}
```

```rust
#[derive(Debug, Clone, Serialize, Deserialize, ToSchema)]
pub enum StringMatch {
    Contains(String, bool),          // (value, case_sensitive)
    Equal(String, bool),             // (value, case_sensitive)
    StartsWith(String, bool),        // (value, case_sensitive)
    EndsWith(String, bool),          // (value, case_sensitive)
    Fuzzy(String, Option<u8>, bool), // (value, max edit distance, substring match)
}
```

```rust
pub enum NumericMatch {
    GreaterThan(f64),
    LessThan(f64),
    Equal(f64),
    Between(f64, f64),
}
```

```rust
pub enum BooleanMatch {
    Equal(bool),
    NotEqual(bool),
}
```

# Productization & Scale : Licensing




```rust
pub struct LSPlan {
    pub id: i32,
    pub name: String,
    pub display_name: String,
    pub description: Option<String>,
    pub created_at: String,
    pub updated_at: String,
    pub price_monthly: f64,
    pub price_yearly: Option<f64>,
    pub billing_type: String,
    pub currency: String,
    pub active: bool,
}
```

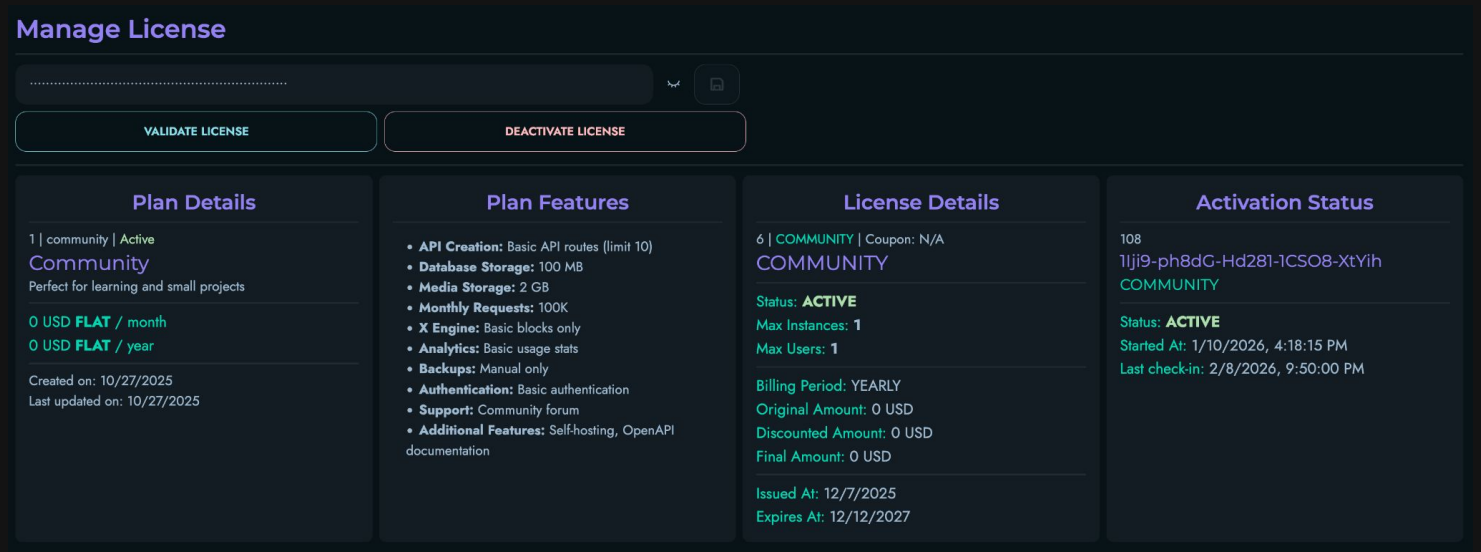
```rust
pub struct LSLicense {
    pub id: i32,
    pub plan_id: i32,
    pub owner_id: i32,
    pub coupon_id: Option<i32>,
    pub name: String,
    pub license_key: String,
    pub max_instances: Option<i32>,
    pub max_users: Option<i32>,
    pub issued_at: String,
    pub expires_at: String,
    pub status: String,
    pub billing_period: String,
}
```

```rust
pub struct LSPlanFeature {
    pub id: Option<i32>,
    pub plan_id: i32,
    pub feature_id: String,
    pub feature_name: String,
    pub feature_details: String,
}
```

```rust
pub struct LSPriceInfo {
    pub original_amount: f64,
    pub discount_amount: f64,
    pub final_amount: f64,
    pub currency: String,
}
```

```rust
pub struct LSActivation {
    pub id: i32,
    pub license_id: i32,
    pub instance_id: String,
    pub started_at: String,
    pub last_check_in: String,
    pub status: String,
}
```

# Productization & Scale : Licensing




```rust
#[derive(Default, Clone, Debug, ToSchema)]
pub struct License {
    pub license_key: String,
    pub activation_id: Option<i32>,
    pub instance_id: String,
    pub instance_endpoint: String,
    pub license_token: Option<String>,
}
```

```rust
#[derive(Clone, Debug, Serialize, Deserialize, ToSchema)]
pub struct LicenseBackupPolicy {
    pub backup_enabled: bool,
    pub backup_schedule_enabled: bool,
    pub backup_schedule_limit_daily: bool,
    pub backup_schedule_limit_hourly: bool,
    pub backup_schedule_max_retention: Option<usize>,
}
```

# Productization & Scale : Licensing

- Unified License Server (license.kinesis.world) handles all licensing operations across instances.
- License Server Components: User, Config, Plan, PlanFeature, Coupon, License, Activation.
- Single license key can activate multiple Kinesis API deployments with unique instance tracking.
- Plans determine available X Engine blocks, storage limits, backup policies, and custom features.
- License key validation, JWT token issuance, and encrypted storage of credentials.
- Background check-ins every 5 minutes ensure license remains active and compliant.
- License Server tracks active users per instance and enforces plan-based user limits.
- Automatic JWT refresh when tokens expire to maintain continuous validation.
- Failed validation reverts instance to Community tier features without data loss.
- License registration, validation, and deactivation restricted to ROOT users via /web/license.
- Feature gating applied at route execution level through X Engine block availability checks.
- License removal notifies License Server and safely reverts to base feature set.
- License validation happens automatically; users experience uninterrupted feature access when compliant.
- License Server URL is obfuscated using ChaCha20-Poly1305 encryption with scrambled key parts embedded in the binary to prevent endpoint discovery and ensure secure communication.

# Productization & Scale: Other changes

- New workflow for building multi-arch & multi-registry docker images through docker buildx.
- API routes for warming up Kinesis DB and compacting records.
- New SimpleUpdateBlock for the X Engine.
- Video tutorials on YouTube and Facebook/Instagram socials.
- Removal of the publish functionality for data objects.
- Making min/max fields optional for structures.
- Addition of the migration commands section in changelog entries.
- Allow custom locators for redirects.
- Asynchronously send emails in another thread.
- CI/CD pipeline improvements.
- Scalar API documentation improvements.
- Improvements to pagination logic and to SEO for blog posts.

```rust
#[derive(Default, Debug, Clone, Serialize, Deserialize, PartialEq, ToSchema)]
pub struct SimpleUpdateBlock {
    /// The global index of the block.
    #[schema(example = 0)]
    pub global_index: u32,
    /// The block or local index of the block.
    #[schema(example = 0)]
    pub block_index: u32,
    /// The x position of the block when rendering it on the Web UI.
    #[schema(example = 0)]
    pub x: f64,
    /// The y position of the block when rendering it on the Web UI.
    #[schema(example = 0)]
    pub y: f64,
    /// The collection being referenced.
    #[schema(example = "users")]
    pub ref_col: String,
    /// The property to update for all data records that match the 'targets' field of this block. If left empty, the whole data record will be replaced by a new one.
    #[schema(example = "name")]
    pub ref_property: String,
    /// Whether to save the results of the update.
    #[schema(example = true)]
    pub save: bool,
    /// A list of targets in order to decide which data records should be updated.
    pub targets: Vec<UpdateTarget>,
    /// The data to set for the property of data records not having been filtered out or to replace the data record by.
    pub set: RefData,
}
```

```bash
#!/bin/bash

set -e  # Exit immediately if a command exits with a non-zero status

# Extract the version from Cargo.toml
VERSION=$(grep "^version" ./Cargo.toml | head -1 | cut -d '"' -f2)

# Ensure the buildx builder exists and supports multi-arch builds
if ! docker buildx inspect multiarch &>/dev/null; then
    echo "Creating a multi-architecture builder..."
    docker buildx create --name multiarch --driver docker-container --use \
    --buildkitd-flags '--debug' \
    --driver-opt image=moby/buildkit:latest,network=host \
    --driver-opt env.BUILDKIT_MEM_SWAPPINESS=0 \
    --driver-opt env.BUILDKIT_MEMORY=8192m
fi

echo "Building and pushing version '${VERSION}'"
TAG=${VERSION} docker buildx bake --builder multiarch all

# Retag the image and push
echo "Building and pushing 'latest' tag"
docker buildx imagetools create \
  gitea.konnect.dev/rust/kinesis-api:${VERSION} \
  --tag gitea.konnect.dev/rust/kinesis-api:latest

docker buildx imagetools create \
  docker.io/edgeking8100/kinesis-api:${VERSION} \
  --tag docker.io/edgeking8100/kinesis-api:latest

echo "Building completed successfully!"
```

# MULTI-DB & ANALYTICS

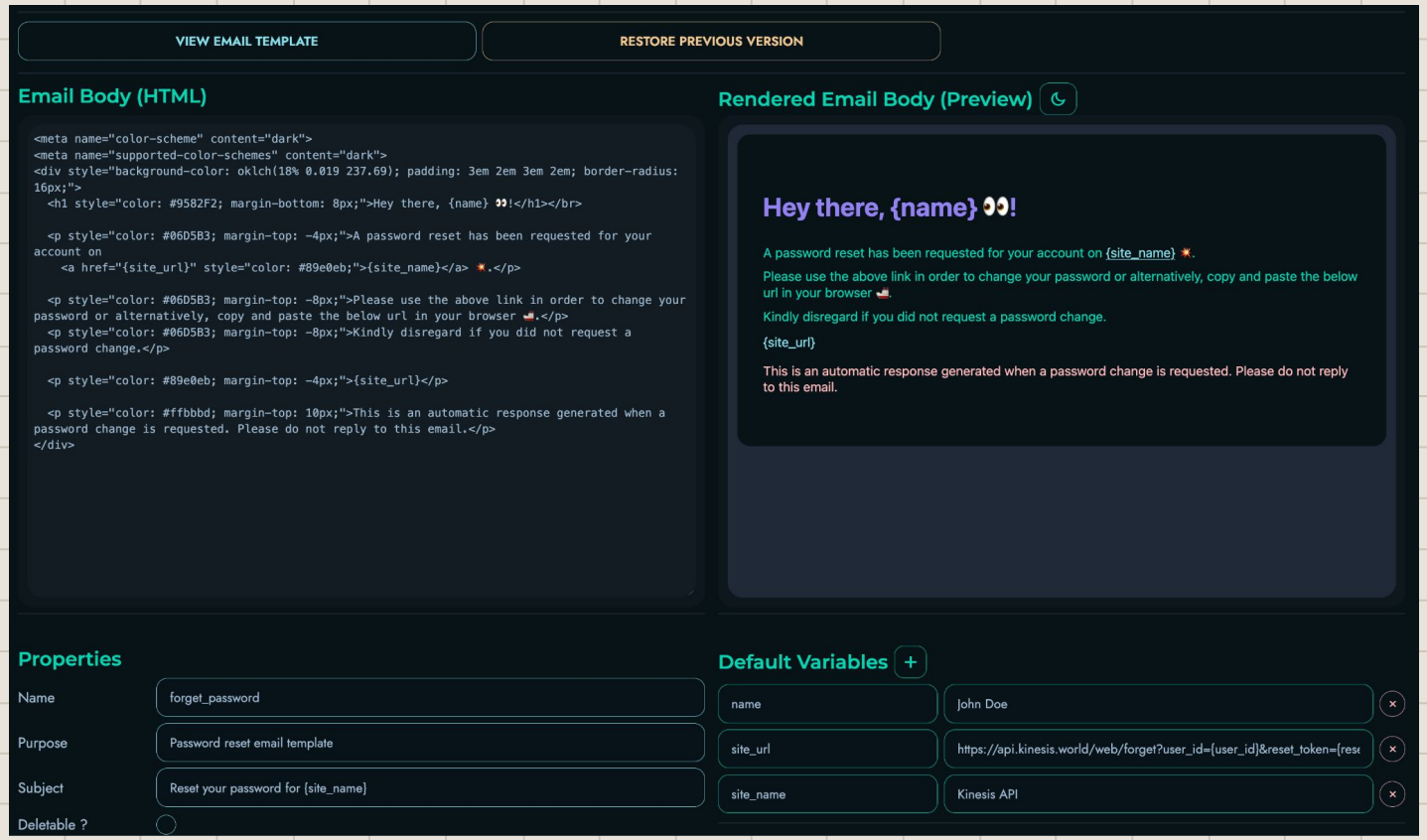
2026

# Multi-DB & Analytics : External DB Support

- Multiple Database Backends: Support for Kinesis DB (default), SQLite, MySQL, and PostgreSQL with seamless switching.
- Diesel ORM Integration: Full Diesel support for all external databases with automatic migration management on startup.
- Bidirectional Data Migration: Migrate data between Kinesis DB and external databases in both directions.
- Component Conversion Layer: Universal converters for all components that handle encryption, blob references, and transactional consistency.
- Safe Migration Process: Lock files prevent concurrent migrations; checksums verify data integrity; detailed logging tracks progress.
- Batch Processing: Records processed in chunks to minimize memory overhead; streaming inserts for large datasets..
- Transaction Atomicity: Each table migration wrapped in transaction; rollback on any error prevents partial data transfers.
- Error Recovery: Graceful degradation with connection retry logic, partial data cleanup, and diagnostic logging for troubleshooting.
- Environment Variables: Configure via DB_BACKEND, DATABASE_URL, DB_NAME, and connection pool settings.
- Schema Compatibility: Automatic backend detection with SQL migrations supporting all three external database syntaxes.
- Migration Tests: Comprehensive unit tests, round-trip testing, and crash recovery validation across all backends.
- Future Enhancements: Scheduled migrations, incremental sync, multi-destination export, and query optimization per backend.

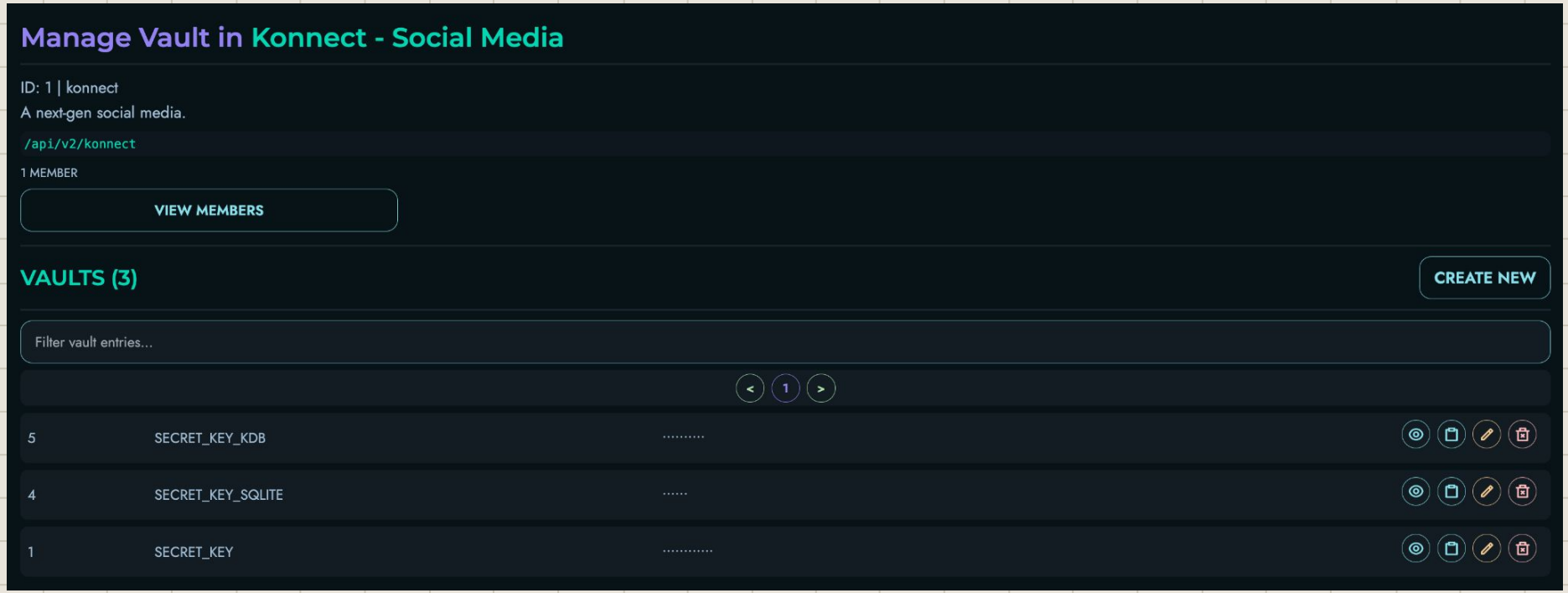
# Multi-DB & Analytics: Email Templates

A new component was introduced along with tests, routes, utils, docs and web pages for allowing end-users to create reusable email templates across various operations.

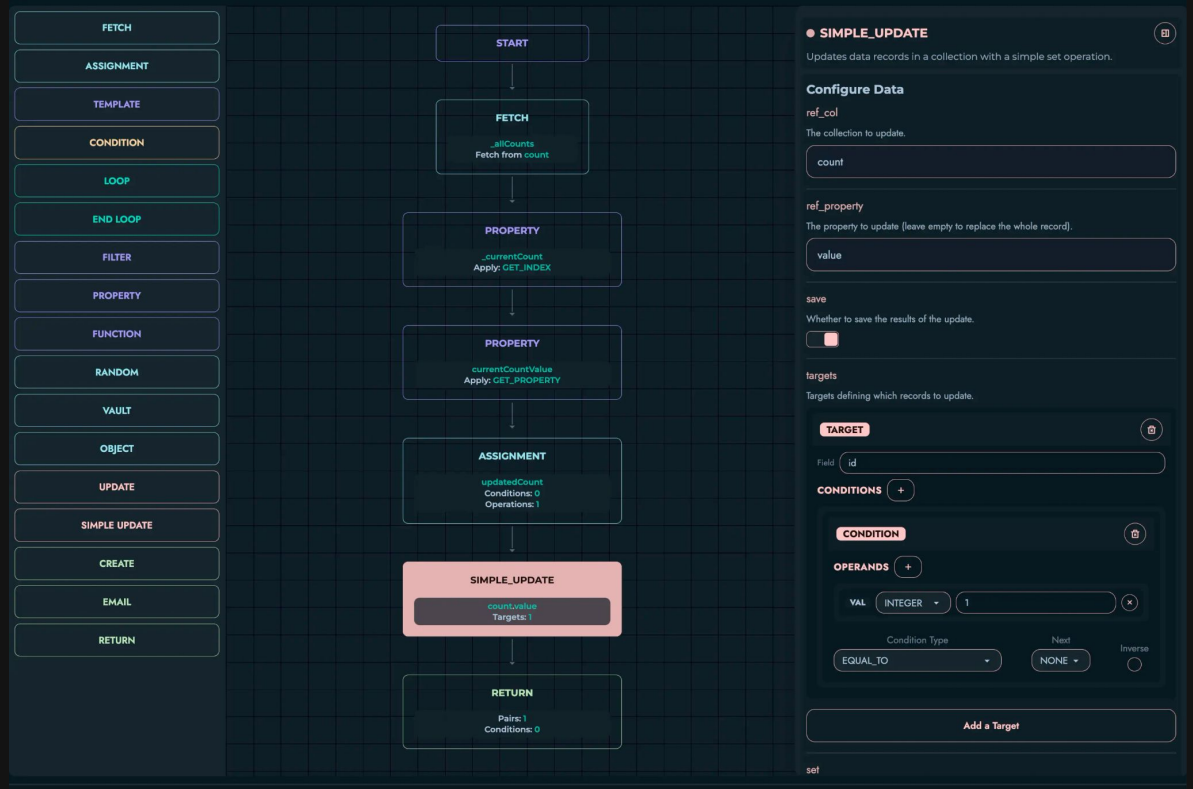


# Multi-DB & Analytics: Vault

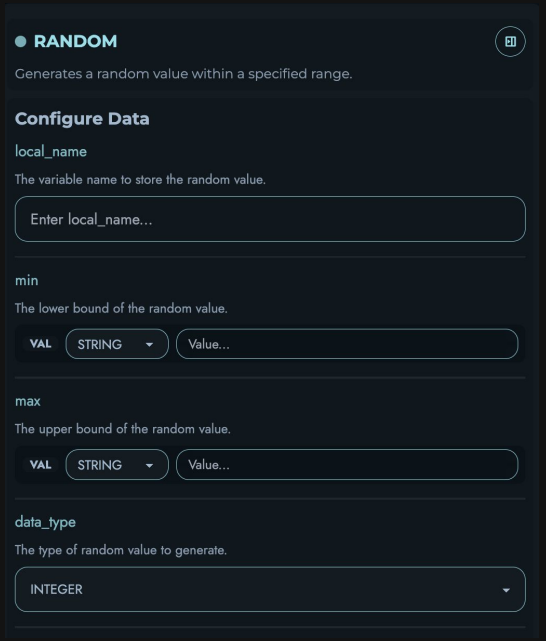
A new component was introduced along with tests, routes, utils, docs and web pages for allowing end-users to create and use secrets in their routes.

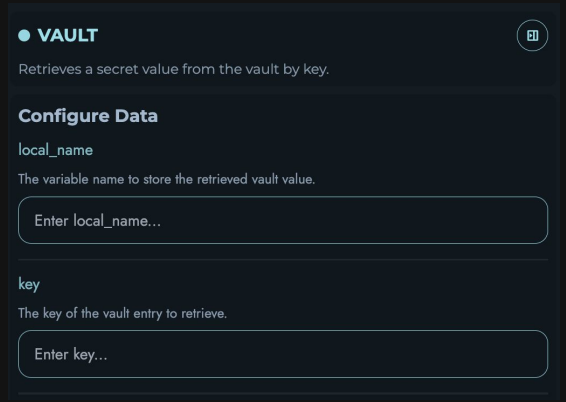


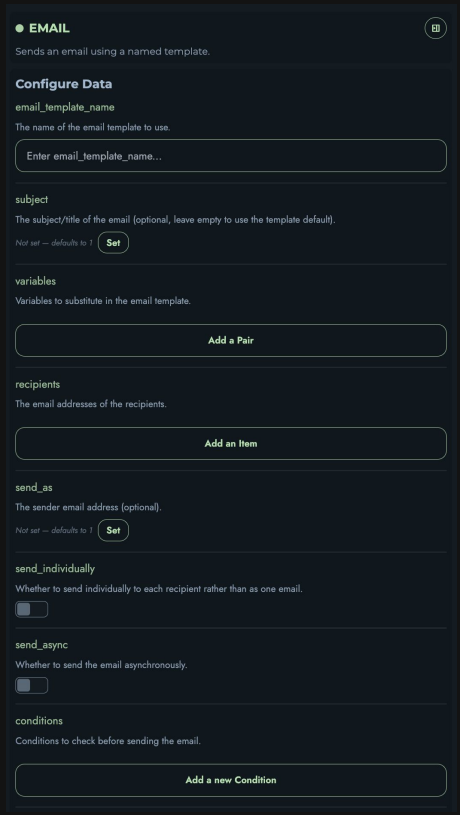
# Multi-DB & Analytics : X Engine UI Refactoring



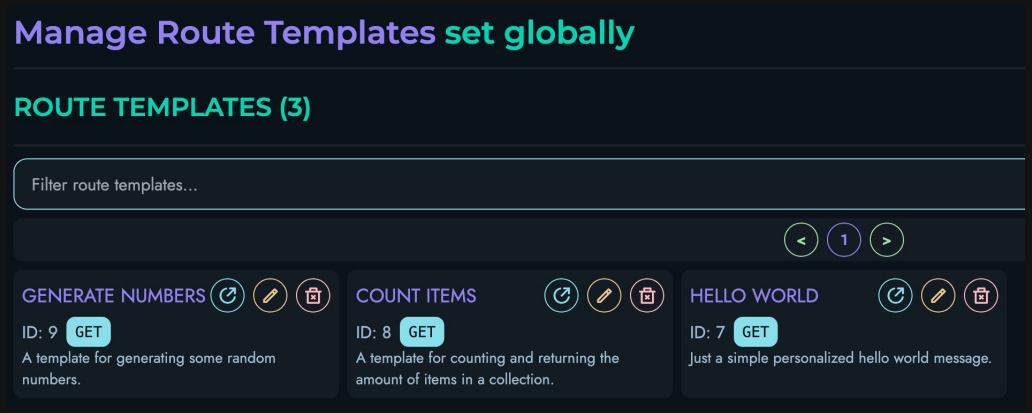
# Multi-DB & Analytics : New X Engine Blocks

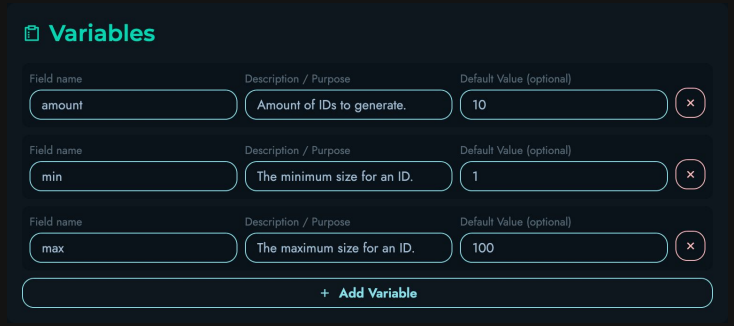


# Multi-DB & Analytics : Route Templates

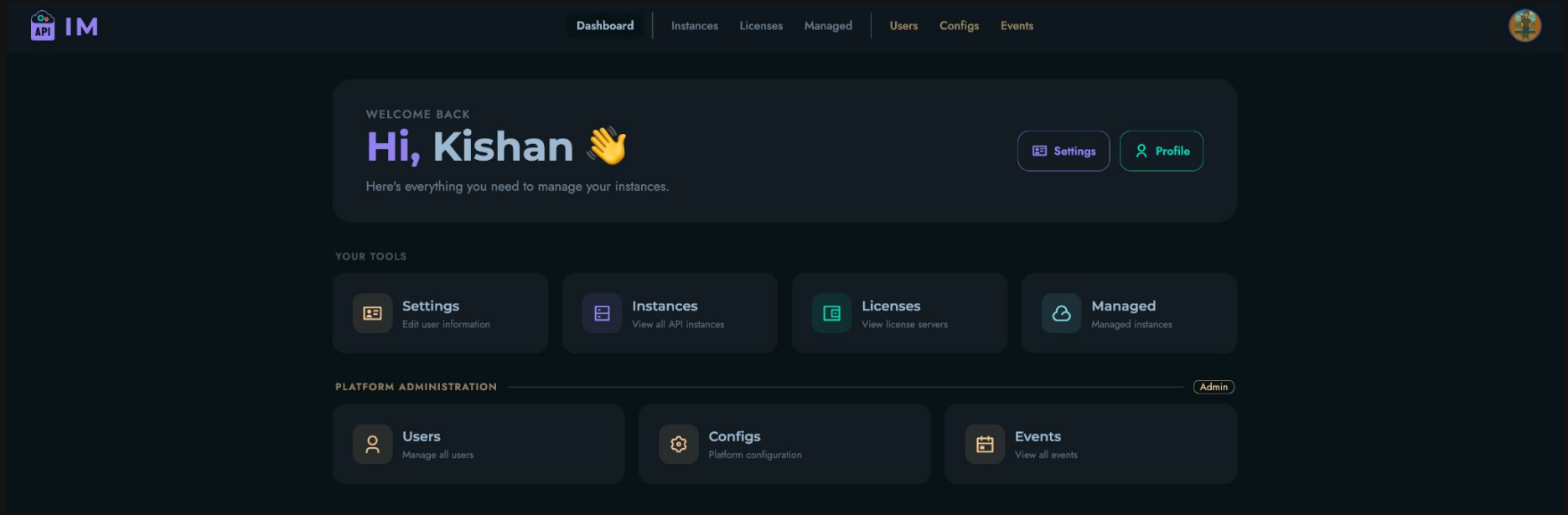


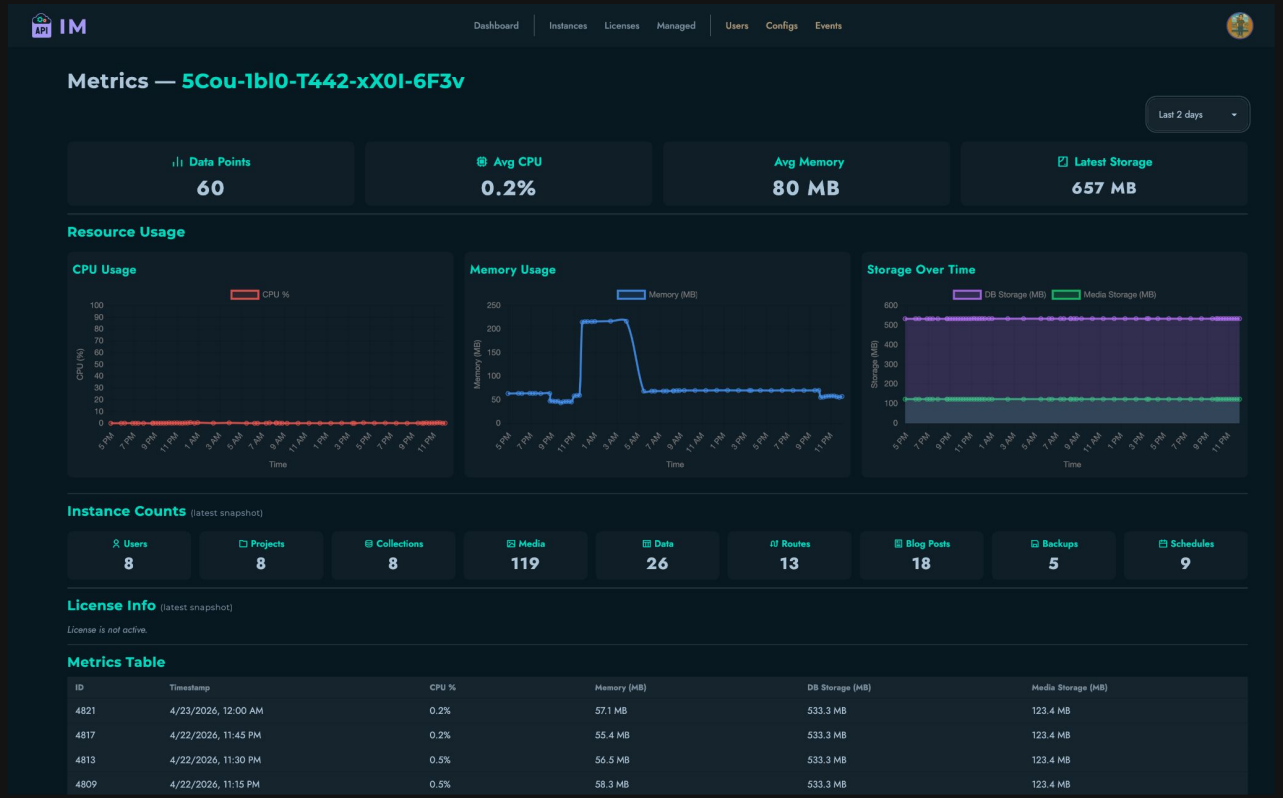

- Introduce the Route Template component for defining reusable, parameterized X Engine route blueprints.
- Add Route Template Variable support for declaring named placeholders with descriptions and optional default values.
- Implement template application workflow in the route creation page: browse templates, fill in variable values, and populate the form.
- Implement full variable substitution across all X Engine block types when applying a route template.

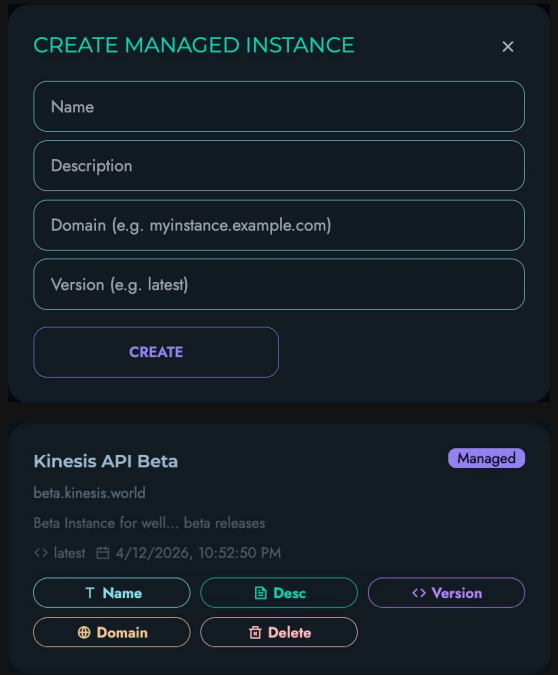
# Multi-DB & Analytics : Instance Manager

- Single Interface for managing instances and licenses.
- Ability to spin and deploy Kinesis API instances within seconds, including DNS configuration.
- Real-time metrics on all instances.

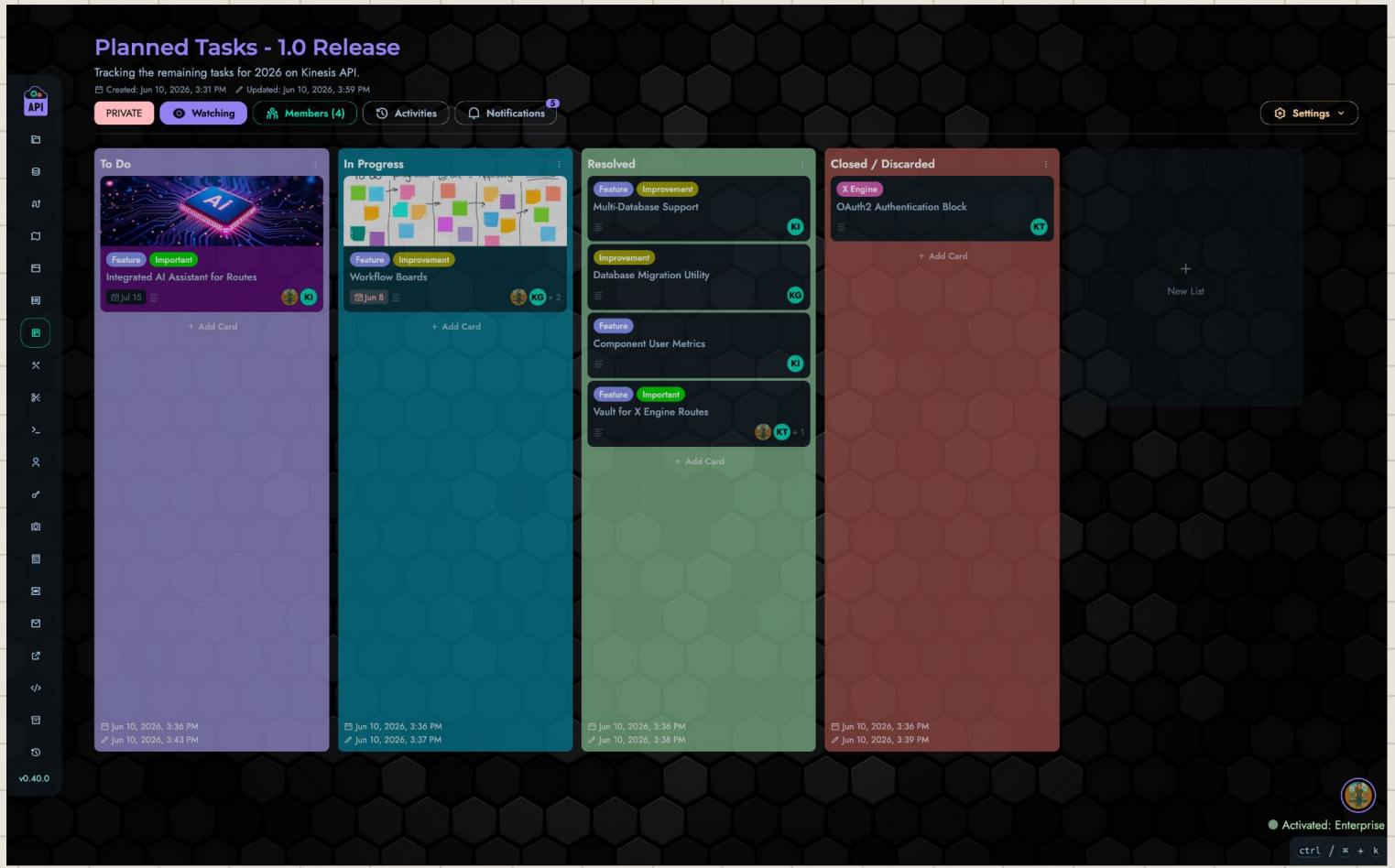


# Multi-DB & Analytics : Instance Manager

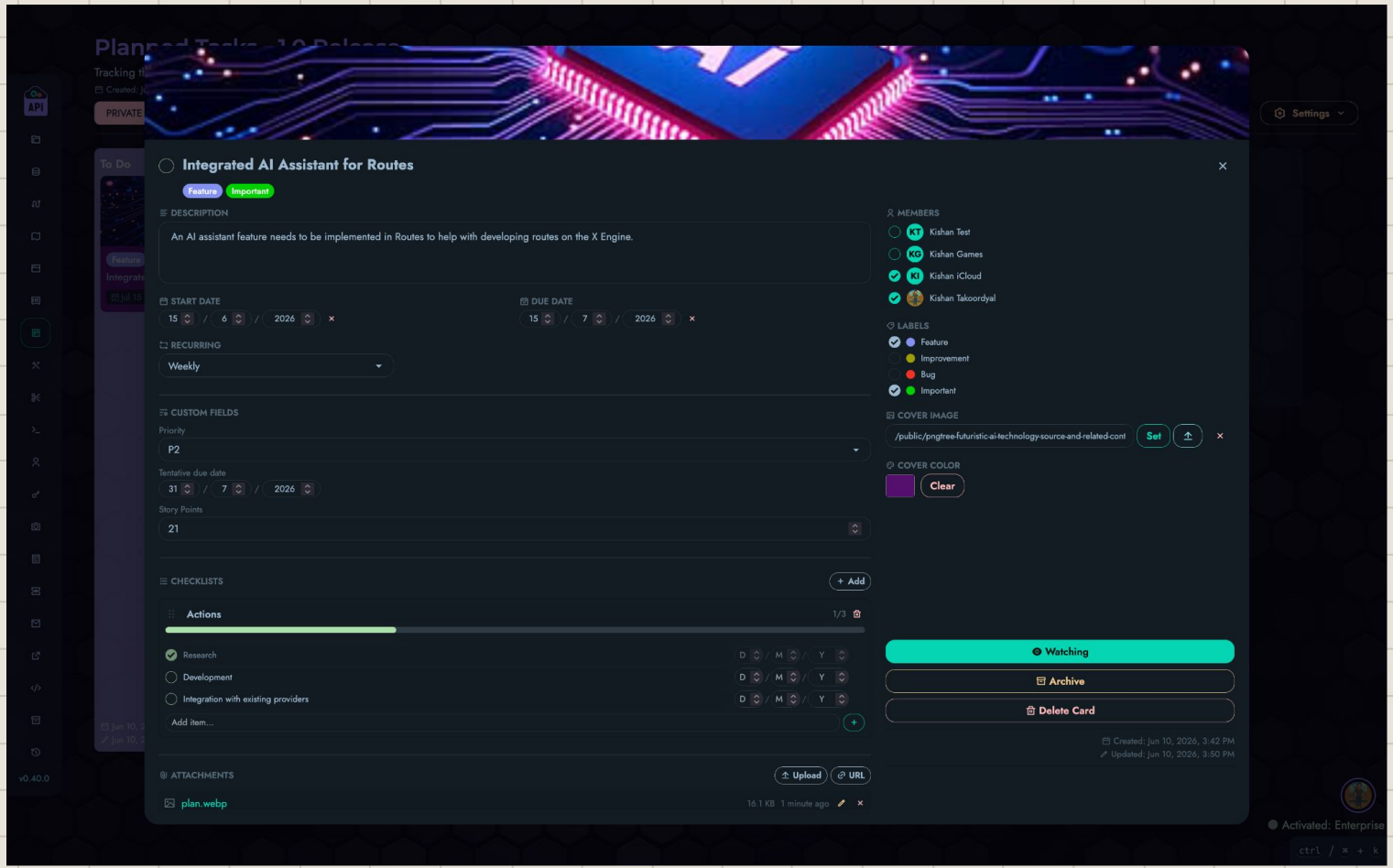


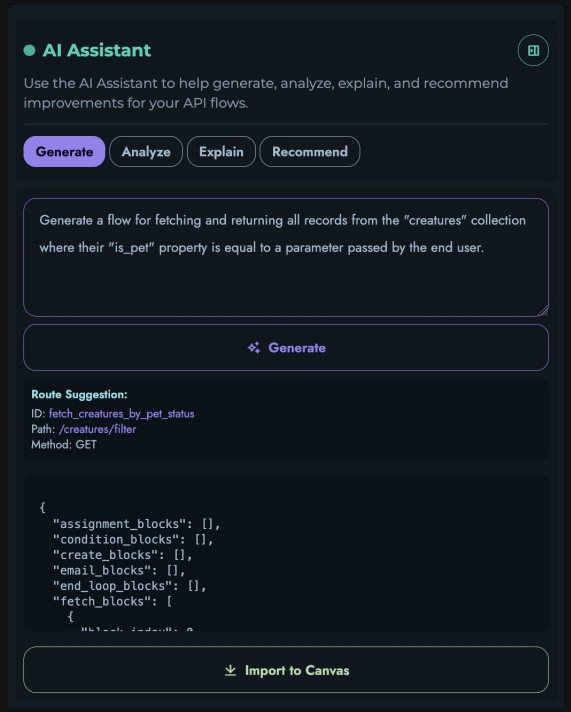
# Multi-DB & Analytics: Workflow Boards

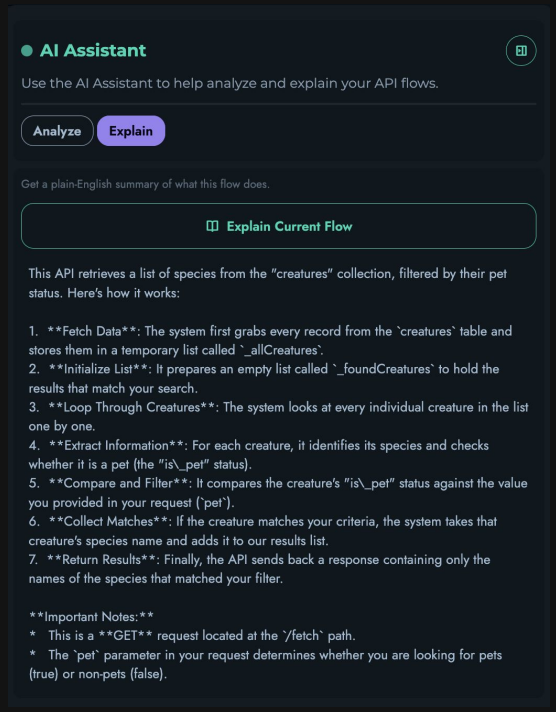


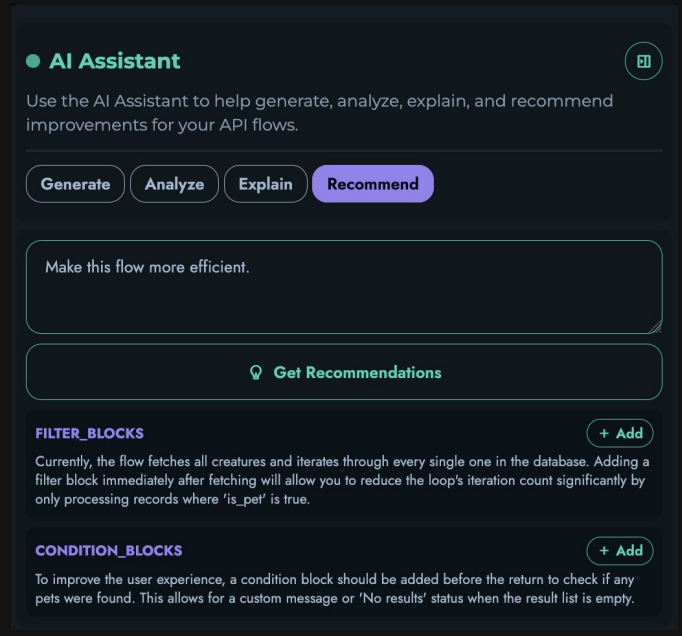
# Multi-DB & Analytics: Workflow Cards



# Multi-DB & Analytics : AI Assistant



# NEXT STEPS

## Productivity

Integrated task planning and organization system with project tracking, deadline management, and team collaboration tools. Enhanced analytics, performance metrics, and reporting dashboards to monitor productivity and identify bottlenecks across workflows.

## Extensibility

New X Engine blocks (Authentication) for advanced workflow automation. Webhook support and dynamic WebSocket connections for real-time event handling and bidirectional communication between systems.

## Experience

Associated SDK or module for JavaScript frameworks (React, Vue, Angular) enabling seamless integration and component-based development.

# THANK YOU!

*"Every complex system starts with a single decision and the willingness to learn from mistakes along the way."*