Development Principles & Methodology with Xano

a Xano Development Methodology

Disclaimer: This is not the official development methodology for Xano. This an individual initiative, started by Guillaume Maison, to assist Xano Developers with a sustainable methodology to develop their backends and provide the smallest tech debt and the best practices for developing.

Contributors:
- Guillaume Maison (guillaume@guillaumemaison.fr) (Creator - Project Manager)

Introduction

This article is designed for all Xano developers seeking a structured approach to development, especially when collaborating in teams. It establishes a set of foundational principles for coding methodology, including best practices for naming conventions, algorithm design, design patterns, and effective teamwork strategies.

I created this guide after observing that many no-code developers often lack clarity on core development practices. The intention here is not to present

an ultimate or definitive methodology but to offer a foundational framework that can evolve with community feedback and serve as a reference for Xano developers.

While specifically tailored for Xano, the concepts outlined can also be adapted for other platforms and visual coding languages. The methodology draws inspiration from various established coding practices, aiming to provide a stable and effective foundation for professional development within Xano.

Methodological principles

1. **Functional analysis**
2. **Development principles**
3. **Implementation**
4. **Analysis <-> Implementation**
5. **Development and Tests**
6. **Production release**

Functional analysis

Functional analysis is a critical step for defining what to develop while avoiding both under-implementation due to insufficient feature analysis and over-design. Properly designed algorithms help streamline development and ensure clarity before coding begins.

Algorithms should be visually designed and prototyped before implementation. I recommend using MIRO for this purpose, as it provides a clear way to build and visualize the sequence of instructions. But any other diagram flow editor tool is good.

Principles:

- Draw up a list of mandatory elements (parameters) before starting.
- Check and validate that there are no gaps in the algorithm, i.e., untreated situations that could lead to data instability.
- Trace major changes in a dedicated box to maintain version clarity.
- Enter all useful references in a dedicated box for easy access.

These algorithms must be generated and documented before any implementation. Additionally, the tests used to validate data transformations should be clearly documented by describing the input data and the expected output results.

These visual algorithms should serve as a technical reference and must remain synchronized with the implemented code throughout the project lifecycle.

Development principles

We'll keep in mind mainly three development principles:

- The Framework way
- Loose coupling
- Strong coherence

The Framework Way of Thinking To ensure both loose coupling and strong coherence, consider building your application as a framework. This mindset encourages structuring your app into reusable, well-defined components where the core logic is abstracted, while the orchestration layer handles the flow of processes. Thinking in frameworks enhances scalability, simplifies maintenance, and ensures better separation of concerns. #### Fundamental principles

- Loose coupling:
  - Components (like functions) have minimal dependencies
  - They interact via well-defined interfaces
  - Changes to one component have minimal impact on the others
  - Improves maintenance and testing
- Strong coherence:
  - The elements of a same component are closely linked
  - Single, focused responsibility per element
  - Related methods and properties grouped together logically

When coupling can be relaxed Here are some legitimate situations for stricter coupling:

- Basic domain logic (intrinsically linked components)
- Performance-critical systems
- Small, single-use applications
- Prototypes/POCs
- Framework-specific components
- Process orchestration

Why does it work? The architecture achieves equilibrium by

- keeping individual components loosely coupled
- allowing the orchestration layer to define business processes
- limiting tight coupling to workflow definition
- maintaining component reuse
- making business processes explicit.

Best practices

- Using dependency injection
- Grouping related dependencies
- Make workflow steps clear and sequential
- Document process flows
- Ensure that components can be tested independently of each other
- Limit the coupling of orchestration to real business needs

This approach creates a clear separation between:

- reusable, loosely coupled components
- the orchestration of business-specific processes
- technical implementation details.

The key idea is that while loose coupling remains a fundamental principle for components, the orchestration layer can legitimately be more tightly coupled because it represents the actual workflows and processes of the business. This creates a pragmatic balance between architectural purity and practical business requirements.

Implementation

A clear and structured implementation phase is crucial for maintaining the integrity of the development process. The following principles aim to ensure that code implementation remains consistent with the design phase while being easy to manage and extend.

Principles

- Follow the algorithm designed and validated during the design stage to avoid unnecessary deviations.

- Break down complex tasks by creating sub-functions where appropriate to enhance readability and maintainability.

- Use Groups to organize instructions that belong to the same functional block, ensuring a logical structure.

- Utilize database element identifiers as parameters instead of passing entire objects. Perform data retrieval using Get Record or Query All Records within the function to ensure only necessary data is processed.

Handling Add-ons

- If the additional information is a single data point closely related to the main record, use JOINS + EVAL to combine the data effectively.

- If the additional information consists of multiple records related to the main record (e.g., child records), use Add-ons to manage the data relationship clearly.

- These principles ensure that the code remains modular, maintainable, and consistent with the designed architecture, making it easier to debug, extend, and collaborate on across development teams.

Analysis <-> Implementation

A solid approach to analysis and implementation ensures continuous improvement and effective collaboration between technical and business teams. The following principles highlight the importance of validation and iteration in the development process.

Principles

- Missing elements in the first version of a solution is acceptable; the process thrives on iterative improvements.

- Ensure that any functional analysis of modifications is reviewed and validated by the head of the relevant business team. If the team lead is unavailable, obtain validation from the founders or key decision-makers.

- Once the functional analysis is validated, it is essential to update the MIRO diagrams before proceeding with implementation. This ensures the design documentation stays synchronized with the development work.

- By maintaining a structured approach to analysis and validation, development remains focused and aligned with business objectives, while reducing the risk of overlooked requirements.

Development and Tests

Private Development branch Principles:

- they must be carried out as described during the analysis.
- they must be validated via Run & Debug and/or Postman
- they are carried out exclusively in the development datasource (live)

Once the developments and tests are passed, you can merge in "Main" branch, and the private development branch can be deleted.

"Main" Branch Unless you're in a big hurry, you don't develop in the "Main" branch. Once a feature is finished and tested, it must be merged into the "Staging" branch.

"Staging" branch Principles:

- The tests are to be carried out with the business team(s) concerned
- These tests are carried out in user mode aka 'real keyboard+mouse conditions'.
- If the tests are not conclusive, return to "Main" branch
- If the tests are conclusive, production release is requested.

"Production" (live) branch Principles:

- No developments are made on the "production" branch (consider it as read-only)
- This is the branch used by the users of the app
- When a bug is discovered, the process is the following:
  - "Production" branch is cloned
  - bug is reproduced on "Dev" datasource
  - fix is made & tested in cloned branch on the "Dev" datasource
  - if fix is ok, then:

    - the functions & endpoints modified are merged back to "Production" branch
    - same in the "Main" & "Staging" branches. Although precautions must be taken not to overwrite some potential modifications.

Production release

Principles:

- Deployment in production is planned the week after the end of the staging tests
- Each release is numbered

Nomenclature principles

Consistent and clear naming conventions help maintain code clarity, reduce confusion, and promote collaboration across development teams. The following principles should be adhered to for variables, functions, and triggers. 1. **Variables** 1. **Functions** 1. **Triggers** 1. **API Endpoints** 1. **Database schema**

Variables

Prefixing Conventions

- `g` : Global / Environment variables
- `p` : Function input parameters
- `l` : Local variables

Case Style

- Use **camelPascalCase** for variable naming:
  - Examples: `pIndustryId`, `lCompanyId`

Functions

Case Style

- Use **snake_case** and **lowercase** for function names.

Naming Pattern

- Function names should follow the structure: `[entity/feature/tool/service]_[functional_terms]()`
- **Examples**

  - `apicall_stripe_payment_intent_create()`
  - `helper_build_pagination()`
  - `stripe_orch_payment_create()`
  - `company_db_create()` / `company_db_update()`

Standardized Prefixes

- Refer to the section on **Function Types** for more detailed prefix usage.

Conventions

- Always end function names with parentheses `()`.

Triggers

Case Style

- Use **snake_case** and **lowercase**.

Naming Pattern

- Trigger names should follow the structure: `tg_[table]_[functional_terms]()`
- **Examples:**
  - `tg_property_configuration_set_last_updates()`
  - `tg_property_create_internal_nickname()`

Comments

- Triggers themselves should not perform any operations directly.
- Their sole purpose is to call a function with the same name, e.g., `tg_property_create_internal_nickname()` should only call the identically named function.
- Always end trigger names with parentheses `()`.
- In most cases, Triggers should not process business rules (processed in orchestrators) but just technical operations.

API Endpoints

Clear and consistent API design and endpoint naming are essential for a well-structured and maintainable codebase. Each CRUD endpoint, even those for

sub-entities, must be able to handle both a single item and an array of items, except on specific and relevant endpoints or functions. Consequently, the functions used within these endpoints must also be capable of processing both single items and arrays of items.

The following conventions ensure clarity across all API interactions.

Endpoint Naming Conventions

Standard CRUD Operations

- `POST [entity]`: Create a single or array of new records for the specified entity.
- `POST [entity]/search`: Search for a list of items based on filters.
- `POST [entity]/search/<name>`: Perform a specific search, such as dropdown suggestions.
- `GET [entity]/{id}`: Retrieve a unique item using its identifier.
- `PATCH [entity]`: Update a single or array of items
- `DELETE [entity]?id=[]`: Delete a single or array of items

Sub-Entity Management

- `POST [entity]/{id}/sub_entities`: Create a single or array of sub-entities linked to the main entity.
- `PATCH [entity]/{id}/sub_entities`: Update a single or array of specific sub-entities linked to the main entity.
- `DELETE [entity]/{id}/sub_entity?id=[]`: Delete a single or array of specific sub-entities linked to the main entity.
- `GET [entity]/{id}/sub_entity/{id}`: Retrieve a unique sub-entity related to the main entity.

Custom Functional Calls

- `POST [entity]/rpc/[function]`: Run a specific function related to the entity.

General Naming Conventions

- Use **snake_case** for multi-word entities and functions.
- Maintain a consistent naming pattern for endpoints and sub-entities.

Examples

- `POST /companies`: adds one or many companies
- `PATCH /companies`: patches one or many companies.
- `GET /companies/{company_id}`: retrieves one company
- `POST /companies/search`: retrieves a list of companies based on filter criterias

- `DELETE /companies?company_id=[id,id,id]`: deletes one or more companies. In that case, the url parameter `company_id` is a single or array of company_ids
- `POST /companies/{company_id}/addresses`: adds one or many addresses to the company
- `POST /companies/{company_id}/addresses`: adds one or many addresses to the company
- `POST /companies/rpc/check_completion`: checks that data is complete for companies
- `POST /companies/{company_id}/rpc/check_completion`: checks that data is complete for the company

Parameters

Naming and Case

- Use **snake_case** and lowercase.
- Ensure parameter names clearly describe the data they represent.
- Examples:
  - `company_id`, `user_id`, `booking_start_date`, `invoice_amount_novat`

General Comments

- By default, there is no predefined input for an endpoint. Use the `Get All Raw Inputs` function to access all data.
- Depending on the frontend tool being used (e.g., Weweb), it may be necessary to declare a JSON input parameter to manage all transmitted data. The JSON content should be generated on the frontend and passed to the backend as a single dynamic JSON object. This approach helps maintain consistency across versions, preventing the need to redefine the endpoint structure on either Xano or the frontend tool, as the payload format remains flexible and adaptable.
- Some parameters may be explicitly declared within the endpoint path.

Database schema

A consistent and clear database schema ensures data integrity and simplifies development and collaboration across teams. The following conventions should be applied when designing tables, fields, and enumerations in the database.

Tables

Prefix convention

- Use prefixes only for **non-functional** entities:

  - **Option Sets:** `os_industry`
  - **Parameter Tables:** `prm_airtable_schema`, `prm_lang`, `prm_errors`

- **Functional entities** should be represented by their actual name in **plural form**:
  - Examples: `invoices`, `invoice_lines`, `companies`, `users`

Case Style

- Use **snake_case**.
- Use **lowercase** exclusively.

Fields

Prefixing Convention

- Field names should use a **3, 4 or 5-letter prefix** to indicate the table context:
  - **Companies:** `cny_name`, `cny_city`
  - **Invoices:** `inv_number`, `inv_date`
  - **Products:** `prod_reference`, `prod_name`

Comments

- Using prefixes for field naming is a convenient way to prevent confusion between identical field names from different tables.
- Table Ids should always be integer. If you want to secure your entities by providing a uuid as an identifier, create a second column with uuid type, create a unique index on this column and in the _db_create() function for this table, create and set this column value. It should not be handled in triggers, as triggers execution can be delayed **after** the the **add record** or **bulk add record** functions are returned (with an empty uuid field).

Case Style

- Use **snake_case**.
- Use **lowercase** exclusively.

Enums (Enumerations)

Case Style

- Use **snake_case** for the enum name.
- Use **UPPERCASE** for enum values as they are considered constants.

- Enums should be consistently formatted to ensure readability and avoid ambiguity

Example:

- ENUM: order_status
  - PENDING
  - COMPLETED
  - CANCELED

Function types

It exists different types of functions:

1. **Apicalls**
2. **Helpers**
3. **DBs**
4. **Validators**
5. **Orchestrators**
6. **Services**
7. **Errors**

Apicalls

Apicalls (apicall_) have a well-defined purpose: they act as a transport layer for a service not managed by the application.

For example, all third-party tools accessible via an API must have helpers functions. Their sole purpose is to transmit a correctly formatted payload to an API URL.

What they need to do:

- they need to manage their API URL themselves, as well as potential tokens
- they need to know which HTTP VERB to use and configure the API call
- they must execute the call and return the full result, without handling errors. This is because an API error is not always blocking.

What they must not do

- they must not build the payload. Each endpoint is perfectly identified and the various parameters of the call must be found in the function inputs.

- they must not handle errors in the API call response (HTTP 4xx, 5xx, etc. errors)

- they must not contain any functional logic other than that relating to this transport layer. For example, if the definition of the API call is stored in a database, it will be able to access the elements of this definition in the database.

Helpers

Helpers (helper_) are functions that perform specific tasks and are reusable. Their purpose is to simplify sometimes complex tasks and reduce code duplication. This makes the code easier to read and maintain. This is the principle of weak coupling.

What they have to do

- they must manipulate input information and return the result of these manipulations

- They must do one thing and one thing only.

- they must be generic and reusable

What they must not do:

- they must not contain business logic

- they must not make external API calls

- they must not access the database - except if they need database information to perform their task

- they must not handle complex error scenarios

DBs

dbs (_db_) are functions that only manage access to database data. They perform direct data access operations.

What they must do:

- They must check that the data used for this access is correct (not null, id exists in the database, …)

- They must only execute the data access.

- They must stop all data processing (precondition) when they encounter an error.

What they must not do:

- They must not transform the data
- They must not contain any business logic

Validators

Validators (__validator__) are functions that validate specific rules. They can validate rules concerning inputs (from endpoints, DAOs, orchestrators, etc.), authentication or business logic.

What they have to do:

- They must check the values passed to them as parameters according to defined validation rules
- They must be consistent with the specific validation they implement
- They must be reusable and predictable.
- They must return true (data is validated) or false with an error code otherwise

What they must not do

- They must not perform business operations
- They must not use third-party services (API, DB access unless the rule is driven by data DB)
- They must not transform or modify data

Note: For an input validator, it's better to handle an action like CREATE/UPDATE/DELETE, as a second input. So there is only one input validator, which is easier to maintain.

Orchestrators

Orchestrators (__orch__) are the functions at the heart of the software platform. They implement the business rules and logic. Each of these functions is specific to a business rule.

What they have to do:

- They coordinate multiple operations in a very precise order
- Implement business rules and logic

- They manage the flow of data between different functions
- They manage the sequence of operations.
- They must delegate specific tasks to the appropriate wrappers/helpers
- They must maintain high-level flows only

What they must not do

- they must not implement low-level functions (those contained in wrappers or helpers)
- They must not integrate data validation (contained in validators, but contain validators)

Services

Services (service_) are functions that act as orchestrators for services requiring the orchestration of several lower-level functions. These service functions are primarily technical functions.

What they have to do:

- They coordinate several lower-level functions (apicall, helpers, dbs, errors, etc.)
- Their service is exclusively a technical service
- they must remain consistent only for the service requested
- they must be reusable and predictable

What they must not do

- they must not implement business rules and logic

Errors

Errors (error_ or _error_) are functions that handle generic or specific errors.

What they must do

- they must format errors consistently and be predictable
- they must be reusable if the same error occurs again and again
- They must categorise errors

- They must present internal errors as public errors

What they must not do:

- They must not implement business logic
- They must not call on third-party tools unless otherwise specified
- They must not expose sensitive information

Design Patterns

In software development, design patterns provide a structured approach to solving common problems, ensuring code that is efficient, scalable, and maintainable. This chapter delves into essential design patterns and best practices for managing APIs, with a focus on handling data, error management, and response consistency.

1. **Principles**
2. **Error Handling**
3. **Response Handling**
4. **Endpoint generic design pattern**

Principles

Handling array of items or lists For functions, by default, you need to be able to handle multiple items, so we transform everything into an array: who can do more (several items) can do less (a single item).

Most function results will therefore be lists.

Facade An endpoint is a facade. In other words, it presents a simplified interface to a more complex system, which is controlled from the functional dimension of the facade.

For example, the ***__db__select()** function can respond to the **POST /entity/search** facade as well as the **GET /entity/{id}** facade. In the latter case, it is possible to get a single item from the array returned by the ***__db__select()** function.

Error Handling

Proper error handling is essential for preserving data integrity and maintaining application stability. There are two ways—non-mutually exclusive—to manage errors:

- **General Error:** If the error can compromise the entire process or harm data integrity, treat it as a *General Error* that immediately stops the algorithm.

- **Business-Process Error:** If the error can be addressed without halting the entire process, handle it within the *calling function* rather than in the function itself.

Guidelines

- **Missing or Invalid Parameters:** If a required parameter is missing or invalid, terminate the process and return an error message immediately.
- **Database Transactions:** For endpoints prone to errors, wrap the entire process in a transaction. This way, if an error occurs, the transaction rollback prevents data corruption.
- **Error Logging:** Use an internal endpoint call—unaffected by transaction rollbacks—to log errors. ### Response Handling

Consistent response management improves the reliability of API interactions.

Guidelines

- All endpoints should return a payload, except for `DELETE` endpoints where a payload may not be necessary. For DELETE endpoints, you can either return `null`, deleted ids or true/false, depending on deletion result.
- For search results (lists/arrays), the results should always be paginated. The endpoint must accept and handle pagination parameters. In some cases, especially param tables with few items, pagination is not mandatory.
- For bulk additions or patches, pagination in resultsets is not required.
- If a search yields no results, an empty list is acceptable without triggering a `NOT_FOUND` error. An empty search is not an error. Except if a specific `GET` request targets an expected record but it's not found, a `NOT_FOUND` error should be returned.

Endpoint generic design pattern

- Endpoint
  - calls an _orch_() function
    - transform input parameter as an array
    - calls for one or many *_validator_*() functions
      - for each validator function, handle error codes
    - proceeds to business data operations & manipulations according to functional rules prior to db operations (through orchestrators)
    - calls one or more *_db_*() functions
      - does a bulk operation
    - proceeds to business data operations & manipulations according to functional rules after db operations (through orchestrators)
    - calls *_db_select() as a result

Database Optimization

Functional Table Splitting

- Split tables based on their functional meaning to improve clarity and maintainability.
- **Example:**
  Instead of one `contracts` table, consider splitting it into `provider_contracts` and `client_contracts` if their structures and purposes differ significantly. ### Indexing
- Create indexes for fields frequently used in filters, sorts, or joins (e.g., foreign keys, unique constraints, timestamps).
- Use composite indexes for queries involving multiple fields (e.g., `created_at` + `user_id`). ### Avoid Overindexing
- Indexes improve read performance but slow down writes. Avoid indexing fields that are rarely queried. ### Query Optimization
- Use Xano's built-in query filters efficiently and avoid fetching unnecessary fields in your queries. ### Audit and Logging Tables
- Maintain separate tables for audit logs or tracking changes (e.g., "activity_logs") to preserve a clean database structure. ### Soft Deletes
- Instead of permanently deleting records, add a `deleted_at` field. This allows you to restore data if needed while filtering "deleted" records in queries. ### Use Primary Keys
- Always use a primary key (preferably auto-incremented) for each table. If you want to have a non-deterministic primary key to be exposed in URLs, then add a second uuid field with a unique index built on it. ### Enums
- Use Enums when you don't need to display the field value.
- Use meaningful names (e.g., `status: ACTIVE, INACTIVE`)
- If it has to be displayed, then link them to a parameter table where you can add a humanly readable label ### Documentation
- Document your schema and relationships clearly, so other developers understand how the database is structured.
- For database diagrams, you can use MIRO, dbDiagrams, … ## Workspace configuration

This is the recommended setup for your workspace to efficiently and securely develop, test, and deploy your backend.

Branches

- v1: main branch for development. It centralizes all the development.
- staging: branch for tests & QA
- production (live): production branch

Datasources

- live: Development db
- staging: Test & QA db
- production: Production db