Development Best Practices

Mendix

RapidData Technologies

Date of Issue: 20/12/2022

Version 1.0

Document Control				
Document Information				
Document Ref#	RD-MX-Best Practices	**Document Title**		Mendix Development Best Practices
Document Classification	○ Open data	■Shared & Confidential	■ Shared & Sensitive	○ Shared & Secret
Status	Current	**Type**	DOC	
Release Date	January			
Revision Date				
Version History				
Version Number	**Date**	**Author(s)**	**Remarks**	
1.0	09/10/2022	Raghav Balasubramanyam	Prepared	
1.1	24/07/2023	Krishna Vamshi	updated	
1.2	21/08/2023	raghav balasubramanyam	Updated Anti-Patterns	
Review History				

Version Number	Date	Reviewer(s)	Remarks
1.0	20/12/2022	Raghav Balasubramanyam	Updated
1.1	15/07/2023	Raghav Balasubramanyam	Updated

Approval History			
Version Number	**Date**	**Approver(s)**	**Remarks**

Mendix Development Best Practices

This document details the best practices for Mendix development. The specifics of the implementation are not provided here and can be found in the developer handbook or the Mendix official documentation.

Structure

Design Considerations – Architecture

Mendix projects are typically monoliths.

Monolithic – All the code resides inside a single application. Any change in code requires a redeployment of the entire project.

This further increases the overhead of regression testing.

Tightly coupled Monolith – In this architecture all the business functionality is crammed into the same module. Typically, there are one to three modules which have all the functionality built in amongst them.

This is the least recommended approach to be taken. Since it makes debugging and maintenance extremely difficult.

Consider this approach only for

- Extremely small applications where the app has between only one to three functionalities.
- When maintaining multiple modules becomes meaningless considering the small number of entities.

Loosely coupled Monolith – In this architecture, each business functionality has its dedicated module. This is the most common approach in Mendix projects.

For example, the module 'CreditCardManagement' can have the following functionalities.

Credit Card Management

1. Requesting a new credit card.
2. Approving a credit card application.
3. Rejecting a credit card application.
4. Managing credit card limits.
5. Blocking a credit card.

This design makes managing an application much simpler than the tightly coupled architecture. Although being a monolith, any change in code would still mean that the changes would take place only after an application deployment.

Microservices Architecture – In this architecture all functionalities are kept in separate projects, instead of the separate modules of the same project.

All the projects communicate with each other using webservices (REST/SOAP). Thus, each functionality acts as a microservice.

This is the most resilient architecture but comes are a high cost.

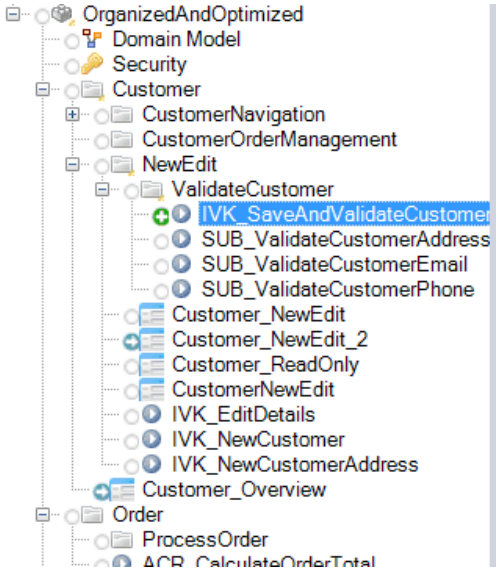
Project Structure

- Modules in the project must be separated to serve a unique business function.
- Marketplace modules should not be alerted.
    - When a marketplace mode is to be used, copy the marketplace module, and add the necessary logic in the copied module. The advantage of this method is that the custom changes are not overwritten when the original module gets upgraded.
- Keep the custom widgets and reusable functions separately, in a module or a folder based on the context.
- Make sure the project's layout is easy to understand. You can use a "layered" approach, where different parts of the application are organized based on their roles, to make the project more organized and manageable.
- Provide a simple overview or documentation that explains how the different modules are structured. This documentation helps developers understand the project layout and how various components interact.

Folder Structure

1. Create top-level folders based on modules and their primary functionalities. For example, if a module is named "Organized And Optimized," the top-level folder could be named " OrganizedAndOptimized."
2. Within each module folder, create second-level folders based on entities present in the module. For instance, if the module contains entities like "Customer," "Order," create respective folders like "Customer," "Order".

3. For third-level folders, group the functionalities that are related to the entities. For instance, if the "Order" entity requires overview pages, microflows, and other functionality, create folders named "Overview," "Microflows," etc., inside the "Product" folder.



4. Create a separate folder for reusable components, such as custom widgets or commonly used microflows. This promotes code reusability across modules.
5. Consider adding a folder to store shared resources like images, documents, or localization files.
6. If the application integrates with external systems or APIs, consider creating a dedicated folder for integration-related modules, microflows, and resources.
7. A folder for utility microflows or reusable logic that can be used across the application.
8. Consider having a folder for documentation files related to the module or application for better organization.

Naming Conventions

Item	Naming Convention	Example	Comments
Modules	PascalCase	MyFirstModule	e.g CompanyManagement
Entities	PascalCase	MyFirstEntity	Avoid underscore, special characters, or abbreviations
Entity Attribute	PascalCase	FirstName _ValidName	1. Attributes that are business-related data to use Pascal case. 2. Attributes needed only for technical purposes to start with an underscore (_).
Associations	Upper_Snake_Case	Person_Address	
Folders	PascalCase		
Microflows	Upper_Snake_Case	ACT_User_Update	Consistent naming conventions for microflows make it easier to identify their purpose and improve maintainability.
Enumerations	PascalCase	StatusType	Descriptive names for enumerations improve code readability and help understand attribute values better
Constants	UPPER SNAKE CASE	MAX_LENGTH	Constants should have meaningful names and be used for fixed values that do not change during runtime.

Java Actions	PascalCase	CalculateTotalPrice	Java actions represent custom logic and should have meaningful names to describe their purpose.

Microflow Naming Conventions

Microflow name should include,

1. Type of event that triggers it. (All event types are listed below)
    a. If a microflow is triggered by several events, consider using more than one prefix.
    b. If a microflow does not comply with any of the patterns listed below, it should not have a prefix.
2. Name of the main entity being processed.
    a. This does not apply if there is no entity or usage of a simpler entity name to improve understandability.
3. Operation it performs.

    Example - ACT_Vendor_StartWorkflow

    This example indicates that the microflow is triggered by an action event (ACT), processes the "Vendor" entity, and initiates a workflow to start the process.

Entity Event Microflows

These are microflows that are related to an entity attribute(s). They will have the below operations.

CRUD – Create Read Update Delete

Event Type	Prefix
Before commit	BCO_{Entity name}
After commit	ACO_{Entity name}
Before create	BCR_{Entity name}
After create	ACR_{Entity name}
Before delete	BDE_{Entity name}
After delete	ADE_{Entity name}
Before rollback	BRO_{Entity name}
After rollback	ARO_{Entity name}

Example:

- Before Commit Microflow (BCO_Order_CalculateTotalOrderAmount):-Triggered before saving an "Order" entity to perform calculations or validations.
- After Commit Microflow (ACO_Order_CalculateTotalOrderAmount): Triggered after saving an "Order" entity to perform additional actions or updates.

Calculated Attribute Microflows

For a microflow that does calculation use CAL_ as a prefix.

Event Type	Prefix
Calculation	CAL_{Entity name}_{Attribute name}

Page-Based Microflows

Event has a timestamp and it changes a record in a DB.

Event Type	Prefix	Used In
On enter event	OEN_{Purpose}	Input elements such as text boxes, buttons etc.
On change event	OCH_{Purpsose}	Input elements
On leave event	OLE_{Purpose}	Input elements
Data source	DS_{Purpose}	Data view, list view, data grid, template grid.
Action button	ACT_{Purpose}	Menu item, navigation item, microflow and action button, drop-down button (“IVK_” is used historically)

Workflow microflows

Event Type	Prefix	Used In
User assignment	WFA_	Returns a list of users who can perform the workflow task.

System action	WFS_	Returns a list of users who can perform the workflow task.
On Created Event	WFC_	Starts when a user task is created, accepts a workflow object.

Page Naming Conventions

Pages use a **suffix** to indicate their use.

Pages that show an overview of a single entity should have a suffix of **_Overview**.

Pages that are to create, edit, or view entity data, and that are not part of a process, should have the suffix **_New**, **_Edit**, **_NewEdit**, or **_View**.

Pages that are used to make a selection of one object have a suffix of **_Select** where the multi-object selection pages should have the suffix **_MultiSelect**.

Pages that are used as a tooltip page should have the suffix **_Tooltip**.

Pages that are called when a user task in a workflow is executed, have suffix **_Workflow**. There is one task page per user task. These pages always have a WorkflowUserTask data view and are specific to performing workflow tasks.

Page Purpose	Suffix
List objects of a single entity type	_Overview
Create an object	_New
Update an object	_Edit
Create *or* Update an object	_NewEdit
View an object (read-only)	_View
Select a single object	_Select
Select multiple objects	_MultiSelect

Page Purpose	Suffix
Tooltip	_Tooltip
Interact with a user task	_Workflow

API Naming Conventions

1. Use nouns (Names, Places, Animals, Things) to represent the resource in the API. For example, /users can represent a list of users and /users/{id} can represent an object of the list.
2. Use verbs to indicate the action performed by the resource. For example
    a. 'Get-User/Users'
    b. 'Get-Invoice-Number/12345'
    c. Delete/John
    d. Update/123

    Note some commonly used verbs in APIs are Get, Delete, Update, Create, Retrieve, Sync, Check etc.
3. Always include the version of the API in the URL. For example, v1/users or V2/users etc.
4. Avoid using verbs in endpoint names. For example, /Users is preferable over /getUsers or /RetreiveUsers
5. For multiple word resources use a hyphen. For example, 'user profiles' can be 'user-profiles'
6. Use plural nouns for collection. For example, '/users' instead of '/user'
7. Use consistent casing throughout the API naming. Prefer either lowercase or camelCase for endpoint names and stick to the chosen convention.
8. Choose descriptive and intuitive names for endpoints. Make sure they convey the purpose and functionality clearly.
9. Keep endpoint names concise and avoid overly long names. A good practice is to strike a balance between descriptiveness and brevity.

10. Organize endpoints hierarchically to reflect the structure of the resources and maintain logical groupings.
11. Ensure the appropriate use of HTTP methods (GET, POST, PUT, DELETE, etc.) according to RESTful principles for CRUD operations.
12. Use standard HTTP status codes (e.g., 200, 201, 400, 404, etc.) to indicate the success or failure of API requests.

Domain Model Best Practices

- The domain model should be readable.
- All new entries should have duplicate validation checks.
- All entities should be logged in the audit trail by default unless specified otherwise in the requirements. Note: Audit trail has a huge performance downside. So, consider this carefully.
- Use auto numbers instead of custom logic to increment IDs.
- Choose descriptive and meaningful names for entities that reflect the real-world objects they represent. This improves understanding and clarity in the domain model.
- Limit the number of entity relationships to maintain simplicity in the domain model. Complex relationships can lead to confusion and difficulty in managing the model.

Domain Model Anti-Patterns

- Do not use underscore in entity names. Use CamelCase instead.
- Cramming a lot of entities in a single module makes the domain look like a spider web. Consider splitting the entities into a separate module based on the functionality.
- Do not inherit twice from an entity.

- Avoid the usage of calculated attributes as much as possible. These come with a high performance overhead.
- Minimize multiple levels of entity inheritance to prevent a tangled and hard-to-maintain domain model. Instead, favor a flatter inheritance structure for better readability.
- Use the audit trail judiciously and only for entities where tracking changes is crucial. Extensive use of the audit trail may impact application performance.

Microflow Best Practices

1. All list view items should be sorted in descending order to show the last updated item on top. unless specified otherwise in the requirements.
2. Avoid committing objects inside a loop.
3. Use exception handling for all important transactions.
4. Avoid committing objects early on in a microflow and keep the commit activity towards the end of the flow.
5. Consider if the microflow can be replaced with a nanoflow.
6. Always use nanoflows for page close actions.
7. For complex business rules or decision-making, consider using rule.
8. Try to avoid using heavy custom Java actions in microflows unless it's absolutely necessary.
9. Make sure microflows are short and focused on specific tasks.
10. Follow a consistent naming convention for microflows and other activities.
11. Pay attention to error handling in microflows. Catch and handle potential errors appropriately to avoid application crashes and unexpected behaviour.

XPATH Best Practices

Page Development Best Practices

1. Always use modal pop-ups and avoid normal pop-up pages. This is to avoid background clicks.
2. All page titles (as displayed in the browser tab's caption) should be meaningful and should not be left unnamed as 'untitled'.
3. Stick to using data grid and data grid 2 as much as possible over list view to avoid writing complex custom search logic.
4. When there are multiple data items/MDs slowing down the page load, use the page load widget to enhance the UX.
5. If multiple nested data views exist, consider breaking down the page into multiple pages. This will enhance the performance but might need a redesign.
6. Implement client-side validation to check user input before submitting data to the server.

Page Security

1. Ensure the page is not accessible for the correct roles and unnecessary role access is not provided. For example, a Mendix administrator may not need access to a customer page.
2. Prevent exposing sensitive or critical pages to users who do not require access to them.
3. Validate input parameters on the page to prevent unauthorized access or manipulation of data.
4. Use secure and non-predictable URLs for important pages to prevent unauthorized access via URL manipulation.
5. Implement proper logout functionality and session management to ensure users are securely logged out and inactive sessions are terminated automatically.
6. Ensure secure transmission of data through HTTPS, especially when handling sensitive information or login credentials.

Code Commit Best Practices

1. Avoid frequent code commits.
2. Do not commit code that is not unit tested locally in the developer environment.
    - As a best practice, the developers should be encouraged to mention in every commit that the code is tested locally and found bug free.
3. Avoid generic statements in the commit such as 'Bug Fixes', 'Committing Changes' etc. The commit comment should be meaningful and descriptive. This is very important when a certain bug needs to be traced back to a commit.
4. Ensure code commits at a healthy interval. By avoiding committing the code for a very long time will cause unnecessary conflicts and rework for the developers.
5. To avoid conflicts, the story assignment for the developers should not overlap.
6. Ensure to take regular updates at an agreed interval and highlight conflicts early on before more complications arise.
7. Ensure to commit the code at least once every day. This should be followed strictly.
8. Audit the API calls and responses to track the activity and analyse the data. This can be done via the API gateway if one is available.
9. Provide a heartbeat API to ping the app when necessary.
10. Take regular updates from the main repository at agreed intervals to stay updated on the latest changes and avoid conflicts.
11. Before committing code, ensure that all changes have been reviewed by another developer if you're working on same user story.
12. Utilize version control branching to work on new features or bug fixes without affecting the main codebase. Merge changes back after thorough testing and review.
13. Break down changes into small, logical units for better code management and easy tracking of modifications.

Web Services Best Practices

REST API Best Practices

1. All responses of a published REST service should be in JSON format.
    - Use the standard response codes for published REST services.

    - Responses should be included in the response body for response codes other than 2XX responses (200,201 etc.)
2. All POST requests should be idempotent. (POST)
    - Ensure adding an idempotent key attribute to enforce idempotency.
    - Include an idempotent key.
3. All API requests should be well-formed. The system should perform a model validation for required values, under-posting, and over-posting.
4. Always version the APIs and store the published APIs as part of a versioned Postman collection.
    - The postman collection should have sample API requests and responses.
    - Document the usage of the APIs in the collection (recommended), or separately in a document if requested by the customer.
5. Consider processing API requests as part of a task queue or message queue.
    - Avoid sending non-essential notifications in real time to save the load in the systems. Such notifications can be sent in batches at the end of the day (recommended) or at some other interval of time.
6. CORS should be disabled for all APIs. If CORS is required, the allowed domains/origins must be specified explicitly. All domains should not be enabled in CORS.
    - The max age for the CORS settings is recommended to be 24 hours (86400 seconds) or 1 hour (3600 seconds) based on the use case.
7. Create comprehensive and user-friendly API documentation that explains each endpoint, request format, response structure, and possible error codes.
8. Implement rate limiting to prevent abuse and ensure fair usage of the API. Limit the number of requests allowed within a certain time period to prevent excessive usage and maintain system stability.
9. Implement security measures such as API key authentication, OAuth, or JWT tokens to ensure secure access to the API endpoints and protect sensitive data.
10. Validate all incoming API requests to prevent malicious input and protect the application from potential security vulnerabilities.
11. For endpoints returning large datasets, implement pagination to avoid overwhelming clients with excessive data in a single response.

SOAP API Best Practices

1. Prefer simple data types for parameters and return values in SOAP APIs to ensure compatibility and ease of use with various systems.
2. Minimize the size of SOAP responses by returning only essential data to improve response times and reduce bandwidth usage.
3. Implement secure communication using HTTPS to encrypt SOAP requests and responses and protect sensitive data during transmission.
4. Consider versioning SOAP APIs to maintain backward compatibility with existing clients while introducing new features or changes.
5. Validate all incoming SOAP requests to prevent malicious input and protect the application from potential security vulnerabilities.
6. Implement proper error handling in SOAP APIs to provide meaningful error messages and status codes. This helps clients understand and handle errors effectively.
7. Use consistent and descriptive naming conventions for SOAP operations, messages, and data types to ensure clarity and maintainability.
8. Regularly update the WSDL (Web Services Description Language) file to reflect any changes in the SOAP API's structure or functionality.
9. Optimize SOAP API performance by minimizing unnecessary data retrieval and optimizing data processing for faster response times.
10. Conduct thorough testing of SOAP APIs to ensure they function as expected and handle various scenarios gracefully.

Webhook API Best Practices

1. Ensure that the webhook endpoint is secure and accessible only to authorized parties. Use authentication mechanisms like API keys or OAuth tokens to protect against unauthorized access.
2. Validate incoming webhook requests to prevent potential security vulnerabilities and handle valid data correctly.
3. Implement logging and monitoring for webhook requests to track their activity and troubleshoot issues effectively.

4. Consider implementing rate limiting for webhook requests to prevent abuse and ensure system stability.
5. Be mindful of the payload size in webhook requests to avoid overwhelming the server with excessively large data.
6. Provide comprehensive documentation for webhook usage, payloads, and response formats to assist developers in integrating with the webhook API.

Standard API Response Codes

Response Code	**Message**	**Usage**
200	OK	Request received
201	Record Created	When a new record is created from a POST.
400	Bad Request	When the request is malformed.
401	Unauthorized	When the authentication fails.
403	Forbidden	When authentication passes but the access is not granted for the resource.
404	Not Found	When a record is not found in a get request.
405	Method Not Allowed	The requested HTTP method is not allowed for the resource
429	Too Many Requests	Rate Limiting
502	Bad Gateway	Server received an invalid response from an upstream server while

		acting as a gateway or proxy.
503	Service Unavailable	
504	Gateway Timeout	The server, acting as a gateway or proxy, did not receive a timely response from the upstream server.

API Session Management

API Security

1. Always use HTTPS for API communication.
    - Note: This may not be required in those cases where the Mendix app exposed the API to an existing API gateway or an integration layer.
2. Use digital signatures to verify the authenticity of an API response using a public and private key pair.
    - This is essential when publishing an API from Mendix.
    - When consuming an API, check with the API provider on how to authenticate the API response for tampering.
    - Recompute the hash from a response without the signature to verify tampering.
3. For publicly exposed API, it is mandatory to implement rate limiting. This can be done by maintaining a counter for allowing, for example, 100 requests per hour.
4. API authorization should be token based with an expiring token.
    - Avoid using basic auth for API authentication.
    - All APIs authentications should be based on expiring JWT tokens (preferred) or OAuth tokens.

    - The JWT token expiry is recommended to be 1 hour / 8 hours / 24 hours depending on the use case.
5. Strongly consider encrypting the response whenever exposed to the internet or to an untrusted network.
6. Monitor and audit API activity to detect and respond to any suspicious or malicious behavior. Regular monitoring helps identify potential security breaches and ensures the security of your Mendix application.

API Gateway Best Practices

The following is the list of recommended API gateways for various instances.

Mendix Cloud Deployment

1. Mendix Datahub
2. Mendix Event Broker

Mendix Private Cloud /On-Prem Deployment

1. Any pre-existing integration layer bus.
2. Mendix Event Broker
3. Apache Kafka
4. Rabbit MQ

Database Best Practices

1. Organize the data in your database into separate tables to eliminate redundancy and maintain data consistency.
2. Retrieve only the data you need from the database to avoid unnecessary data transfer, reducing load times and improving application performance.
3. Minimize the number of database joins, especially for complex queries, as they can impact performance.

4. Create regular backups of your database to safeguard against data loss due to system failures, accidents, or other unexpected events.
5. Validate data before inserting or updating it in the database to ensure data integrity and prevent erroneous or inconsistent data.
6. Use appropriate data types for each attribute to minimize storage requirements and improve query performance.
7. Regularly monitor your database performance to identify bottlenecks and optimize database operations for better application performance.

Consider using indexes with the following guidelines:

- 1. Only on larger tables, the effect on smaller tables is not useful.
- 2. Only on attributes that are searched on a lot, for example, for the entity User, the Name attribute will be the most likely attribute to search on.
- 3. Only apply indexes on attributes with a lot of unique values, like a String attribute. The values of a Boolean attribute will most likely be 'true' fifty per cent of the time.
- 4. Never on a String – unlimited attribute. This will result in major index pages.
- 5. When there are more read actions than write actions.
- 6. When the main index attribute is the sorting attribute of a data grid.
- 7. When searching for an indexed attribute, use Starts with function. Applying indexes in other situations are useless.

Which action does the end user perform most often?

1. Read - Index can be added and might improve app performance.
2. Write - Don't add an index, it will most likely reduce app performance.

Associations Best Practices

Always use an underscore in your association names.

1. 1. Make association names unique when you have multiple associations between the same entities. For example, Price_Product and Price_Product_2 doesn't tell you the difference between the associations. Price_Product and Price_Product_Current tells you that there is a current pricing object and other pricing objects associated to a product.

2. Select the most suitable association type (e.g., one-to-one, one-to-many, or many-to-many) based on the relationship between entities in your domain model.
3. Name associations with clear and descriptive terms that reflect the nature of the relationship between entities. Avoid ambiguous or generic names to improve the understanding of the domain model.
4. Limit the number of associations between entities to prevent an overly complex domain model. Excessive associations can make the model difficult to manage and maintain.
5. Utilize inverse associations to maintain consistency and optimize bidirectional navigation between entities. This allows for easy access to related data from both sides of the association.
6. Periodically review the associations in your domain model and refactor as needed. This helps keep the model organized and relevant as the application evolves.

Performance Best Practices

Batching Best Practices

- Consider alternatives like DataHub instead of batch sync.
- When doing a batch update always ensure that the retrieved list is sorted with a unique attribute in either ascending or descending order along with an offset and limit.
- Implement proper error handling and rollback mechanisms during batch updates to ensure data consistency. In case of any failure during the batch process, rollback changes to maintain data integrity.

Caching Best Practices

- Do not cache sensitive data on the client's browser.

Accessibility Best Practices

- Consider the level of accessibility required. For example, required for protanopia, deuteranopia etc.
- Ensure the contrast level is at least 3 for all pages and 4.5 for pages that must be accessible.
- Alternative text should be added for all images in the image's general properties.
- Alternative text can be added for any element that does not have a meaningful context.
- Do a Grayscale analysis for all pages.
- Order the page headers to be hierarchical to aid blind users.

Security Best Practices

- Ensure proper session management.
    - Inactive users should be automatically logged off after a defined interval of inactivity.
        - Recommended session time out for non-sensitive applications is between 1 hour to 8 hours, for sensitive applications between 15 to 30 minutes and for highly sensitive applications (banking etc) between 5-15 minutes.
        - If the session data is stored in a client-side cookie, it should be encrypted.
    - Ensure to have CSRF token validation for any form field input.
        - Implement a CSRF token validation failure exception.
    - Track anonymous user sessions effectively.

        - Implement session cookies (that expire when the browser is closed) for anonymous users. If session cookies are not solving the purpose, only then consider persistent cookies. However persistent cookies for an anonymous session are NOT recommended as may raise privacy concerns.
- Use Role-Based Access Control to restrict user access to certain functionalities and data based on their assigned roles. This helps prevent unauthorized access to sensitive information.
- Always use HTTPS for secure communication between the client and the server, ensuring that data transmitted over the network is encrypted and protected from interception.
- Implement input data validation at the server-side to prevent potential security vulnerabilities such as SQL injection and cross-site scripting (XSS) attacks.
- Regular Security Audits: Conduct regular security audits and penetration testing to identify and address potential security weaknesses in the application.
- Keep all Widgets and Modules components, including libraries, frameworks, and plugins, up to date to prevent exploitation of known vulnerabilities.
- Implement data encryption at rest to protect sensitive information stored in databases and file systems from unauthorized access.

Storage of Sensitive Information (Data at Rest)

Sensitive information should always be encrypted before being persisted in the database. Some examples of sensitive information are,

- Bank account numbers
- Social Security numbers
    - Aadhar Number
    - PAN Number
    - Passport Number
    - Driver's License Number
    - Emirates ID
- Financial Information

    - Bank account number
    - Credit card number
- Health Information
    - Medical Records
    - Patient diagnosis
- Connection information of consumed services like
    - Keys/Tokens
    - Credentials
    - Service Locations
- Biometric data
    - Fingerprints
    - Iris scans
- Personal information
    - Personal chat messages
    - Personal email

Protection of Sensitive Information During Transmission (Data in Transit)

To be updated

Mobile Development Best Practices

Naming Conventions

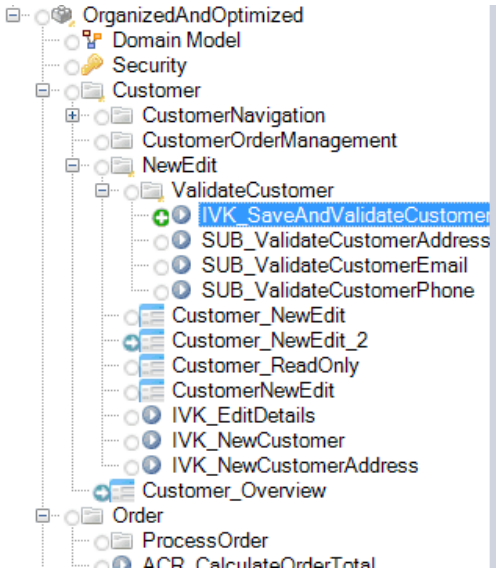
Item	Naming Convention	Example	Comments
Modules	PascalCase	MyFirstModule	
Entities	PascalCase	MyFirstEntity	Avoid underscore, special characters or abbreviations
Entity Attribute	PascalCase	FirstName	3. Attributes that are business related data to use pascal case

			4. Attributes needed only for technical purpose to start with an underscore (_).
Associations	Upper_Snake_Case	Person_Address	
Folders	PascalCase		
Microflows	Uppser_Snake_Case	ACT_User_Update	Consistent naming conventions for microflows make it easier to identify their purpose and improve maintainability.
Enumerations	PascalCase	StatusType	Descriptive names for enumerations improve code readability and help understand attribute values better
Constants	UPPERCASE	MAX_LENGTH	Constants should have meaningful names and be used for fixed values that do not change during runtime.
Java Actions	PascalCase	CalculateTotalPrice	Java actions represent custom logic and should have meaningful names to describe their purpose.

- All entities' names adhere to Pascal case conventions.
- All module names adhere to Pascal case conventions.
- All module names adhere to Pascal case conventions.

Folder Structure

1. Create top-level folders based on modules and their primary functionalities. For example, if a module is named "Organized And Optimized," the top-level folder could be named " OrganizedAndOptimized."
2. Within each module folder, create second-level folders based on entities present in the module. For instance, if the module contains entities like "Customer," "Order," create respective folders like "Customer," "Order".
3. For third-level folders, group functionality related to the entities. For instance, if the "Order" entity requires overview pages, microflows, and other functionality, create folders named "Overview," "Microflows," etc., inside the "Product" folder.



4. Create a separate folder for reusable components, such as custom widgets or commonly used microflows. This promotes code reusability across modules.
5. Consider adding a folder to store shared resources like images, documents, or localization files.
6. If the application integrates with external systems or APIs, consider creating a dedicated folder for integration-related modules, microflows, and resources.
7. A folder for utility microflows or reusable logic that can be used across the application.
8. Consider having a folder for documentation files related to the module or application for better organization.

Microflow Naming Conventions

Microflow name should include.

1. Type of event which triggers it. (All event types are listed below)
    a. If a microflow is triggered by several events, consider using more than one prefix.
    b. If a microflow does not comply to any of the patterns listed below, it should not have a prefix.
2. Name of the main entity being processed.
    c. This does not apply if there is no entity or usage of a simpler entity name to improve understandability.
3. Operation it performs.

    Example - ACT_Vendor_StartWorkflow

    This example indicates that the microflow is triggered by an action event (ACT), processes the "Vendor" entity, and initiates a workflow to start the process.

Entity Event Microflows

These are microflows that are related to an entity attribute(s). They will have the below operations.

Event Type	Prefix
Before commit	BCO_{Entity name}
After commit	ACO_{Entity name}
Before create	BCR_{Entity name}
After create	ACR_{Entity name}
Before delete	BDE_{Entity name}
After delete	ADE_{Entity name}

Before rollback	BRO_{Entity name}
After rollback	ARO_{Entity name}

Example:

- Before Commit Microflow (BCO_Order_CalculateTotalOrderAmount):-Triggered before saving an "Order" entity to perform calculations or validations.
- After Commit Microflow (ACO_Order_CalculateTotalOrderAmount): Triggered after saving an "Order" entity to perform additional actions or updates.

Calculated Attribute Microflows

For a microflow that does calculation use CAL_ as a prefix.

Event Type	Prefix
Calculation	CAL_{Entity name}_{Attribute name}

Page-Based Microflows

Event Type	Prefix	Used In
On enter event	OEN_{Purpose}	Input elements such as text boxes, buttons etc.
On change event	OCH_{Purpsose}	Input elements
On leave event	OLE_{Purpose}	Input elements
Data source	DS_{Purpose}	Data view, list view, data grid, template grid.

Action button	ACT_{Purpose}	Menu item, navigation item, microflow and action button, drop-down button (“IVK_” is used historically)

Workflow microflows

Event Type	Prefix	Used In
User assignment	WFA_	Returns a list of users who can perform the workflow task.
System action	WFS_	Returns a list of users who can perform the workflow task.
On Created Event	WFC_	Starts when a user task is created, accepts a workflow object.

API Naming Conventions

1. Use nouns (Names, Places, Animals, Things) to represent the resource in the API. For example, /users can represent a list of users and /users/{id} can represent an object of the list.
2. Use verbs to indicate the action performed by the resource. For example
    a. ‘Get-User/Users’
    b. ‘Get-Invoice-Number/12345’
    c. Delete/John
    d. Update/123

Note some commonly used verbs in APIs are Get, Delete, Update, Create, Retrieve, Sync, Check etc.

3. Always include the version of the API in the URL. For example, v1/users or V2/users etc.
4. Avoid using verbs in endpoint names. For example, /Users is preferable over /getUsers or /RetreiveUsers
5. For multiple word resources use a hyphen. For example, 'user profiles' can be 'user-profiles'
6. Use plural nouns for collection. For example, '/users' instead of '/user'
7. Use consistent casing throughout the API naming. Prefer either lowercase or camelCase for endpoint names and stick to the chosen convention.
8. Choose descriptive and intuitive names for endpoints. Make sure they convey the purpose and functionality clearly.
9. Keep endpoint names concise and avoid overly long names. A good practice is to strike a balance between descriptiveness and brevity.
10. Organize endpoints hierarchically to reflect the structure of the resources and maintain logical groupings.
11. Ensure the appropriate use of HTTP methods (GET, POST, PUT, DELETE, etc.) according to RESTful principles for CRUD operations.
12. Use standard HTTP status codes (e.g., 200, 201, 400, 404, etc.) to indicate the success or failure of API requests.

Best Practices

This section details the best practices for development. The specifics of the implementation are not provided here and can be found in the developer handbook or the Mendix official documentation.

Project Structure

- Modules in the project must be separated to serve a unique business function.
- Marketplace modules should not be alerted.

- Like custom widgets and reusable functions, into a separate module. This central location allows easier access and encourages consistency throughout the application.
- Make sure the project's layout is easy to understand. You can use a "layered" approach, where different parts of the application are organized based on their roles, to make the project more organized and manageable.
- Provide a simple overview or documentation that explains how the different modules are structured. This documentation helps developers understand the project layout and how various components interact.

Domain Model Best Practices

- The domain model should be readable.
- All new entries should have duplicate validation checks.
- All entities should be logged in the audit trail by default unless specified otherwise in the requirements. Note: Audit trail has a huge performance downside. So, consider this carefully.
- Use auto numbers instead of custom logic to increment IDs.
- Choose descriptive and meaningful names for entities that reflect the real-world objects they represent. This improves understanding and clarity in the domain model.
- Limit the number of entity relationships to maintain simplicity in the domain model. Complex relationships can lead to confusion and difficulty in managing the model.

Domain Model Anti-Patterns

- Do not use underscore in entity names. Use CamelCase instead.
- Cramming a lot of entities in a single module makes the domain look like a spider web. Consider splitting the entities into a separate module based on the functionality.
- Do not inherit twice from an entity.
- Avoid the usage of calculated attributes as much as possible. These come with a high performance overhead.
- Minimize multiple levels of entity inheritance to prevent a tangled and hard-to-maintain domain model. Instead, favor a flatter inheritance structure for better readability.
- Use the audit trail judiciously and only for entities where tracking changes is crucial. Extensive use of the audit trail may impact application performance.

Microflow Best Practices

1. All list view items should be sorted in descending order to show the last updated item on top.
   unless specified otherwise in the requirements.

2. Avoid committing objects inside a loop.
3. Use exception handling for all important transactions.
4. Avoid committing objects early on in a microflow and keep the commit activity towards
   the end of the flow.
5. Consider if the microflow can be replaced with a nanoflow.
6. Always use nanoflows for page close actions.

7. For complex business rules or decision-making, consider using decision tables.
8. Try to avoid using heavy custom Java actions in microflows unless it's absolutely necessary.
9. Make sure microflows are short and focused on specific tasks.
10. Follow a consistent naming convention for microflows and other activities.
11. Pay attention to error handling in microflows. Catch and handle potential errors appropriately to
    avoid application crashes and unexpected behaviour.

Page Development Best Practices

1. Always use modal pop-ups and avoid normal pop-up pages. This is to avoid background clicks.
2. All page titles (as displayed in the browser tab's caption) should be meaningful and should not be left unnamed as 'untitled'.
3. Stick to using data grid and data grid 2 as much as possible over list view to avoid writing complex custom search logic.
4. When there are multiple data items/MDs slowing down the page load, use the page load widget to enhance the UX.
5. If multiple nested data views exist, consider breaking down the page into multiple pages. This will enhance the performance but might need a redesign.
6. Implement client-side validation to check user input before submitting data to the server.

Page Security

1. Ensure the page is not accessible for the correct roles and unnecessary role access is not provided. For example, a Mendix administrator may not need access to a customer page.
2. Prevent exposing sensitive or critical pages to users who do not require access to them
3. Validate input parameters on the page to prevent unauthorized access or manipulation of data
4. Use secure and non-predictable URLs for important pages to prevent unauthorized access via URL manipulation.
5. Implement proper logout functionality and session management to ensure users are securely logged out and inactive sessions are terminated automatically.
6. Ensure secure transmission of data through HTTPS, especially when handling sensitive information or login credentials.

Code Commit Best Practices

1. Avoid frequent code commits.
2. Do not commit code that is not unit tested locally in the developer environment.
    - As a best practice, the developers should be encouraged to mention in every commit that the code is tested locally and found bug free.
3. Avoid generic statements in the commit such as 'Bug Fixes', 'Committing Changes' etc. The commit comment should be meaningful and descriptive. This is very important when a certain bug needs to be traced back to a commit.
4. Ensure code commits at a healthy interval. By avoiding committing the code for a very long time will cause unnecessary conflicts and rework for the developers.
5. To avoid conflicts, the story assignment for the developers should not overlap.
6. Ensure to take regular updates at an agreed interval and highlight conflicts early on before more complications arise.
7. Ensure to commit the code at least once every day. This should be followed strictly.
8. Audit the API calls and responses to track the activity and analyse the data. This can be done via the API gateway if one is available.
9. Provide a heartbeat API to ping the app when necessary.

10. Take regular updates from the main repository at agreed intervals to stay updated on the latest changes and avoid conflicts.
11. Before committing code, ensure that all changes have been reviewed by another developer if you're working on same user story.
12. Utilize version control branching to work on new features or bug fixes without affecting the main codebase. Merge changes back after thorough testing and review.
13. Break down changes into small, logical units for better code management and easy tracking of modifications.

Web Services Best Practices

REST API Best Practices

1. All responses of a published REST service should be in JSON format.
    - Use the standard response codes for published REST services.
    - Responses should be included in the response body for response codes other than 2XX responses (200,201 etc.)
2. All POST requests should be idempotent.
    - Ensure adding an idempotent key attribute to enforce idempotency.
3. All API requests should be well-formed. The system should perform a model validation for required values, under-posting, and over-posting.
4. Always version the APIs and store the published APIs as part of a versioned Postman collection.
    - The postman collection should have sample API requests and responses.
    - Document the usage of the APIs in the collection (recommended), or separately in a document if requested by the customer.
5. Consider processing API requests as part of a task queue or message queue.
    - Avoid sending non-essential notifications in real time to save the load in the systems. Such notifications can be sent in batches at the end of the day (recommended) or at some other interval of time.

6. CORS should be disabled for all APIs. If CORS is required, the allowed domains/origins must be specified explicitly. All domains should not be enabled in CORS.
    - The max age for the CORS settings is recommended to be 24 hours (86400 seconds) or 1 hour (3600 seconds) based on the use case.
7. Create comprehensive and user-friendly API documentation that explains each endpoint, request format, response structure, and possible error codes.
8. Implement rate limiting to prevent abuse and ensure fair usage of the API. Limit the number of requests allowed within a certain time period to prevent excessive usage and maintain system stability.
9. Implement security measures such as API key authentication, OAuth, or JWT tokens to ensure secure access to the API endpoints and protect sensitive data.
10. Validate all incoming API requests to prevent malicious input and protect the application from potential security vulnerabilities.
11. For endpoints returning large datasets, implement pagination to avoid overwhelming clients with excessive data in a single response.

SOAP API Best Practices

1. Prefer simple data types for parameters and return values in SOAP APIs to ensure compatibility and ease of use with various systems.
2. Minimize the size of SOAP responses by returning only essential data to improve response times and reduce bandwidth usage.
3. Implement secure communication using HTTPS to encrypt SOAP requests and responses and protect sensitive data during transmission.
4. Consider versioning SOAP APIs to maintain backward compatibility with existing clients while introducing new features or changes.

5. Validate all incoming SOAP requests to prevent malicious input and protect the application from potential security vulnerabilities.
6. Implement proper error handling in SOAP APIs to provide meaningful error messages and status codes. This helps clients understand and handle errors effectively.
7. Use consistent and descriptive naming conventions for SOAP operations, messages, and data types to ensure clarity and maintainability.
8. Regularly update the WSDL (Web Services Description Language) file to reflect any changes in the SOAP API's structure or functionality.
9. Optimize SOAP API performance by minimizing unnecessary data retrieval and optimizing data processing for faster response times.
10. Conduct thorough testing of SOAP APIs to ensure they function as expected and handle various scenarios gracefully.

Webhook API Best Practices

1. Ensure that the webhook endpoint is secure and accessible only to authorized parties. Use authentication mechanisms like API keys or OAuth tokens to protect against unauthorized access.
2. Validate incoming webhook requests to prevent potential security vulnerabilities and handle valid data correctly.
3. Implement logging and monitoring for webhook requests to track their activity and troubleshoot issues effectively.
4. Consider implementing rate limiting for webhook requests to prevent abuse and ensure system stability.
5. Be mindful of the payload size in webhook requests to avoid overwhelming the server with excessively large data.
6. Provide comprehensive documentation for webhook usage, payloads, and response formats to assist developers in integrating with the webhook API.

Standard API Response Codes

Response Code	Message	Usage
200	OK	Request received
201	Record Created	When a new record is created from a POST.
400	Bad Request	When the request is malformed.
401	Unauthorized	When the authentication fails.
403	Forbidden	When authentication passes but the access is not granted for the resource.
404	Not Found	When a record is not found in a get request.
405	Method Not Allowed	The requested HTTP method is not allowed for the resource
429	Too Many Requests	Rate Limiting
502	Bad Gateway	Server received an invalid response from an upstream server while acting as a gateway or proxy.
503	Service Unavailable	
504	Gateway Timeout	The server, acting as a gateway or proxy, did not receive a timely response from the upstream server.

API Security

1. Always use HTTPS for API communication.
    - Note: This may not be required in those cases where the Mendix app exposed the API to an existing API gateway or an integration layer.
2. Use digital signatures to verify the authenticity of an API response using a public and private key pair.
    - This is essential when publishing an API from Mendix.
    - When consuming an API, check with the API provider on how to authenticate the API response for tampering.
    - Recompute the hash from a response without the signature to verify tampering.
3. For publicly exposed API, it is mandatory to implement rate limiting. This can be done by maintaining a counter for allowing, for example, 100 requests per hour.
4. API authorization should be token based with an expiring token.
    - Avoid using basic auth for API authentication.
    - All APIs authentications should be based on expiring JWT tokens (preferred) or OAuth tokens.
    - The JWT token expiry is recommended to be 1 hour / 8 hours / 24 hours depending on the use case.
5. Strongly consider encrypting the response whenever exposed to the internet or to an untrusted network.
6. Monitor and audit API activity to detect and respond to any suspicious or malicious behaviour. Regular monitoring helps identify potential security breaches and ensures the security of your Mendix application. – Needs elaboration

API Gateway Best Practices

The following is the list of recommended API gateways for various instances.

Mendix Cloud Deployment

1. Mendix Datahub
2. Mendix Event Broker

Mendix Private Cloud /On-Prem Deployment

1. Any pre-existing integration layer bus.
2. Mendix Event Broker
3. Apache Kafka
4. Rabbit MQ

Database Best Practices

1. Organize the data in your database into separate tables to eliminate redundancy and maintain data consistency. – Needs elaboration.
2. Retrieve only the data you need from the database to avoid unnecessary data transfer, reducing load times and improving application performance.
3. Minimize the number of database joins, especially for complex queries, as they can impact performance.
4. Create regular backups of your database to safeguard against data loss due to system failures, accidents, or other unexpected events. *Note: Mendix cloud deployed applications have automatic nightly backup feature. Consider this for on-prem applications only.*
5. Validate data before inserting or updating it in the database to ensure data integrity and prevent erroneous or inconsistent data.
6. Use appropriate data types for each attribute to minimize storage requirements and improve query performance.
7. Regularly monitor your database performance to identify bottlenecks and optimize database operations for better application performance.

Consider using indexes with the following guidelines:

- 1. Only on larger tables, the effect on smaller tables is not useful.
- 2. Only on attributes that are searched on a lot, for example, for the entity User, the Name attribute will be the most likely attribute to search on.
- 3. Only apply indexes on attributes with a lot of unique values, like a String attribute. The values of a Boolean attribute will most likely be 'true' fifty per cent of the time.
- 4. Never on a String – unlimited attribute. This will result in major index pages.
- 5. When there are more read actions than write actions.
- 6. When the main index attribute is the sorting attribute of a data grid.
- 7. When searching for an indexed attribute, use Starts with function. Applying indexes in other situations are useless.

Which action does the end user perform most often?

1. Read - Index can be added and might improve app performance.
2. Write - Don't add an index, it will most likely reduce app performance.

Associations Best Practices

Always use an underscore in your association names.

1. Make association names unique when you have multiple associations between the same entities. For example, Price_Product and Price_Product_2 doesn't tell you the difference between the associations. Price_Product and Price_Product_Current tells you that there is a current pricing object and other pricing objects associated to a product.
2. Select the most suitable association type (e.g., one-to-one, one-to-many, or many-to-many) based on the relationship between entities in your domain model.

3. Name associations with clear and descriptive terms that reflect the nature of the relationship between entities. Avoid ambiguous or generic names to improve the understanding of the domain model.
4. Limit the number of associations between entities to prevent an overly complex domain model. Excessive associations can make the model difficult to manage and maintain.
5. Utilize inverse associations to maintain consistency and optimize bidirectional navigation between entities. This allows for easy access to related data from both sides of the association.
6. Periodically review the associations in your domain model and refactor as needed. This helps keep the model organized and relevant as the application evolves.

Performance Best Practices

Batching Best Practices

- Consider alternatives like DataHub instead of batch sync.
- When doing a batch update always ensure that the retrieved list is sorted with a unique attribute in either ascending or descending order along with an offset and limit.
- Implement proper error handling and rollback mechanisms during batch updates to ensure data consistency. In case of any failure during the batch process, rollback changes to maintain data integrity.

Caching Best Practices

- Do not cache sensitive data on the client's browser.

Accessibility Best Practices

- Consider the level of accessibility required. For example, required for protanopia, deuteranopia etc.
- Ensure the contrast level is at least 3 for all pages and 4.5 for pages that must be accessible.
- Alternative text should be added for all images in the image's general properties.
- Alternative text can be added for any element that does not have a meaningful context.
- Do a Grayscale analysis for all pages.
- Order the page headers to be hierarchical to aid blind users.

Security Best Practices

- Ensure proper session management.
    - Inactive users should be automatically logged off after a defined interval of inactivity.
        - Recommended session time out for non-sensitive applications is between 1 hour to 8 hours, for sensitive applications between 15 to 30 minutes and for highly sensitive applications (banking etc) between 5-15 minutes.
        - If the session data is stored in a client-side cookie, it should be encrypted.
    - Ensure to have CSRF token validation for any form field input.
        - Implement a CSRF token validation failure exception.
    - Track anonymous user sessions effectively.
        - Implement session cookies (that expire when the browser is closed) for anonymous users. If session cookies are not solving the purpose, only then consider persistent cookies. However persistent cookies for an anonymous session are NOT recommended as may raise privacy concerns.

- Use Role-Based Access Control to restrict user access to certain functionalities and data based on their assigned roles. This helps prevent unauthorized access to sensitive information.
- Always use HTTPS for secure communication between the client and the server, ensuring that data transmitted over the network is encrypted and protected from interception.
- Implement input data validation at the server-side to prevent potential security vulnerabilities such as SQL injection and cross-site scripting (XSS) attacks.
- Keep all Widgets and Modules components, including libraries, frameworks, and plugins, up to date to prevent exploitation of known vulnerabilities.
- Implement data encryption at rest to protect sensitive information stored in databases and file systems from unauthorized access.

Offline Best Practices

Generic Best Practices

- Limit the amount of data that will be synchronized by customizing the synchronization configuration or security access rules.
- Because network connections can be slow and unreliable and mobile devices often have limited storage, avoid synchronizing large files or images (for example, by limiting the size of photos)
- Try to synchronize through a nanoflow instead of a UI element so you can add error handling to the synchronization activity which can handle cases when synchronization fails (connection errors, model, and data related errors, and more)
- Synchronize large files or images using selective synchronization.
- Use an ‘isDeleted’ Boolean attribute for delete functionality so that conflicts can be handled correctly on the server.
- Use before- and after-commit microflows to pre- or post-process data.
- Use a microflow call in your nanoflows to perform additional server-side logic such as retrieving data from a REST service, or accessing and using complex logic such as Java actions

- Help your user remember to synchronize their data so it is processed as soon as possible: you can check for connectivity and automatically synchronize in the nanoflow that commits your object, or remind a user to synchronize while using a notification or before signing out to ensure no data is lost.

Preventing Synchronization Issues

- Avoid altering entities or their attributes, renaming them, or changing their type in offline apps after the initial release. Doing so may lead to objects or values becoming inaccessible to offline users. If necessary, consider performing an intermediate release that maintains backward compatibility before implementing changes in the subsequent release after synchronizing all apps.
- Refrain from deleting objects that can be synchronized with offline users, as this action would result in the loss of any changes made to those objects during synchronization attempts.
- Instead of employing domain-level validation for offline entities, opt for nanoflows or input validation. Additionally, it is a good practice to revalidate on the server using microflows.
- When committing objects that are referenced by other objects, ensure that the other objects are also committed to maintaining data consistency.
-

If synchronization is triggered using a synchronize action in a nanoflow and an error occurs, it is possible to handle the error gracefully using the nanoflow error handling.

Conflict Resolution

In certain scenarios, multiple users may synchronize the same state of an object on their respective devices, make modifications, and subsequently sync the updated object back to the server. This process can result in the last synchronization overwriting the entire content of the object on the server, commonly referred to as a "last wins" approach.

If an alternative approach is required to handle conflicts, a before-commit microflow can be utilized. For instance, conflicts can be detected by using a revision ID attribute on the entity.

With this information, custom conflict resolution can be implemented to manage conflicting changes effectively.

Offline Microflow Best Practices

To avoid backwards-compatibility errors in offline microflow calls after the initial release, we suggest these best practices:

- Do not rename microflows or move them to different modules.
- Do not rename modules that contain microflows called from offline apps.
- Do not add, remove, rename, or change types of microflow parameters.
- Do not change return types.
- Do not delete a microflow before making sure that all devices have received an update.

Offline Data Security Best Practices

Configuring Domain Model Access Rules

The Mendix Client only stores objects and attributes that the current user has read access to. Incomplete or misconfigured access rules on the domain model may cause too much data to be synchronized to the device databases.

Limiting Data Xpath Constraints

In apps where you want to grant end-users working with objects access to the responsive profile, but you do not wish to grant them access to an offline-first navigation profile, it is possible to limit the amount of data by an XPath constraint using the **Configure Synchronization** screen.

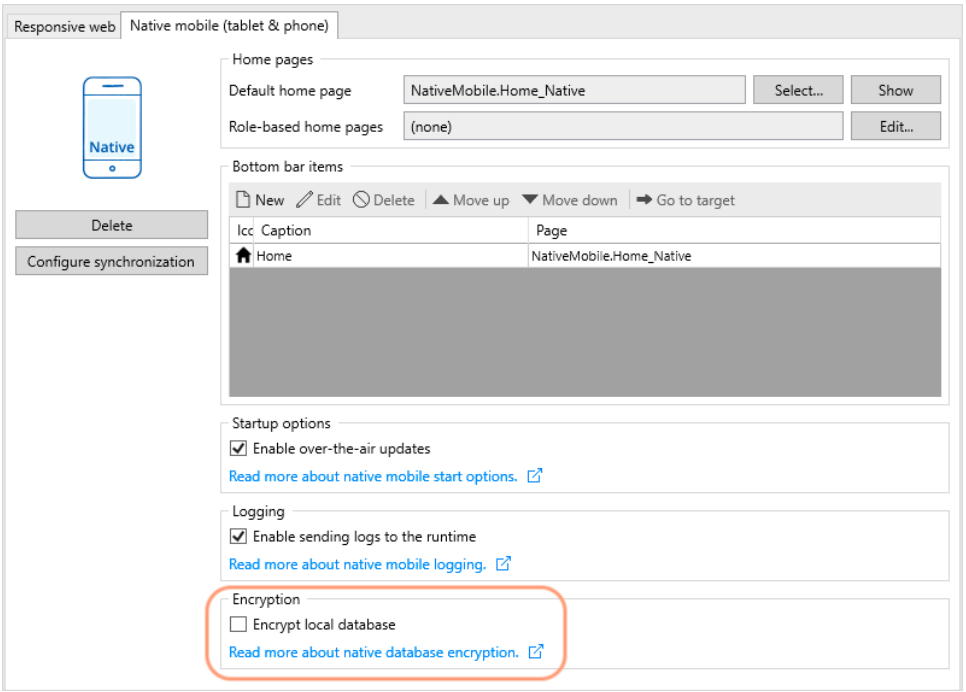
Using Non-Persistent Entities

For sensitive data that should never be stored locally, consider using non-persistent entities to store the data temporarily and use microflow calls to securely process the data on the server.

The app keeps the non-persistent objects only in the memory and removes them when they are no longer needed. However, this approach requires connectivity to the Mendix Runtime to call microflows, and thus limits the app's offline-first capabilities.

Encrypting the Local Database

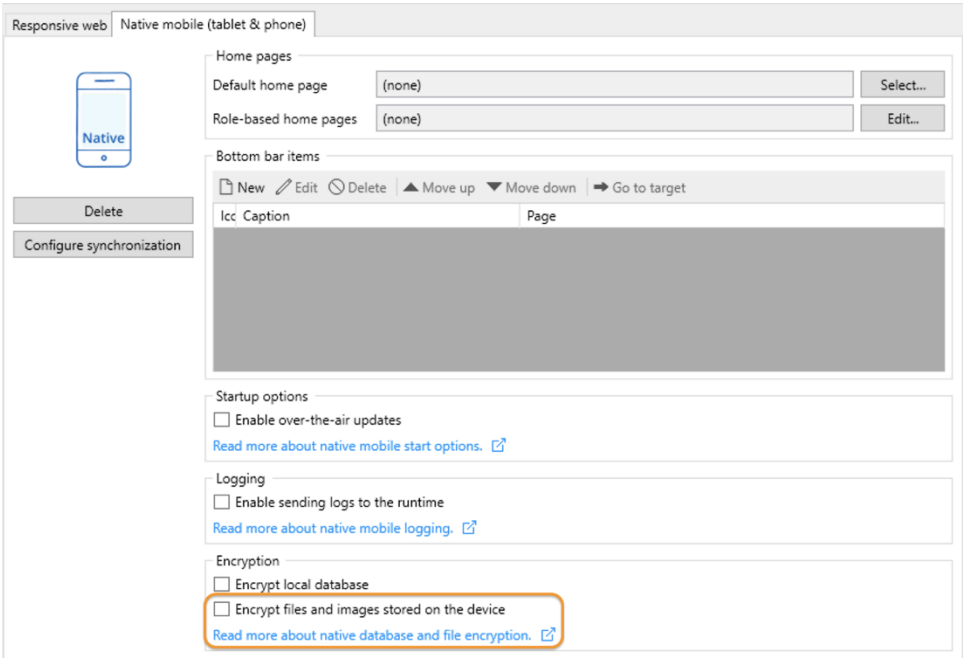
Suppose you have to store sensitive data on a device and cannot control the operating system your app is running on. In that case, you should enable database encryption for your app. This ensures that all data in the local database is encrypted before storing it on the device. Note that encryption will impact your app's performance. Full synchronization of all clients is required to enable it.



Encrypting User Files

Similar to encrypting the local database, you should enable local file encryption for extra protection if the files or images your app stores are sensitive. This option ensures that file contents for entities that specialize from either System.FileDocument or System.Image are encrypted. This option also includes files added to the app by the end-user, for example,

when a user takes a photo using the camera of the device and store it in an entity that specialized from System.Image.



Storage of Sensitive Information (Data at Rest)

Sensitive information should always be encrypted before being persisted in the database. Some examples of sensitive information are,

- Bank account numbers
- Social Security numbers
    - Aadhar Number
    - PAN Number
    - Passport Number
    - Driver's License Number
    - Emirates ID
- Financial Information
    - Bank account number
    - Credit card number
- Health Information
    - Medical Records
    - Patient diagnosis

- Connection information of consumed services like
    - Keys/Tokens
    - Credentials
    - Service Locations
- Biometric data
    - Fingerprints
    - Iris scans
- Personal information
    - Personal chat messages
    - Personal email

Antipatterns

Anti-patterns in Mendix development are common practices or design patterns that should be avoided because they can lead to poor code quality, decreased maintainability, or inefficient use of the Mendix platform. Identifying and addressing these anti-patterns is essential to ensure that Mendix applications remain scalable, maintainable, and performant. Here are some common anti-patterns in Mendix development:

Monolithic Modules

Monolithic Microflows:

Transaction (ACID) in Mendix

1. Mendix transaction – Any microflow is a transaction.
2. Database transaction

All or nothing (rollback)

Anti-Pattern: Creating large, monolithic microflows that handle multiple responsibilities and functionalities.

Issue: Monolithic microflows are difficult to maintain, test, and understand.

Solution: Break down microflows into smaller, focused units, and use sub-microflows for modularization.

Excessive Nested Microflows:

Try and limit the sub flows to 2 or 3 levels at max.

Anti-Pattern: Nesting multiple layers of microflows to handle complex logic.

Issue: Deeply nested microflows can make the application harder to follow and maintain.

Solution: Limit nesting depth and use sub-microflows judiciously. Keep most logic in the main microflow.

Inefficient Database Queries:

Anti-Pattern: Making excessive database queries within a microflow without optimizing data retrieval.

Issue: Poor database performance and increased resource consumption.

Solution: Optimize queries by retrieving only necessary data and leveraging the Mendix Object Cache.

Mistakes being done currently.

1. Managing lists poorly –
    a. Avoid retrieve all instead limit the list items using XPATH
    b. Consider batching
    c. Managing empty lists
    d. Objects – always check for empty
2. Ensure all commit activities are
    a. Part of the parent MF
    b. Towards the end of the MF
    c. Avoid commit in loops
    d. Avoid commits (ip possible) in sub MFs

Overreliance on Custom Java Actions:

Anti-Pattern: Using custom Java actions for tasks that can be accomplished with Mendix's built-in capabilities.

Issue: Increased complexity and reduced maintainability.

Solution: Explore Mendix's native functionality before resorting to custom Java actions.

Mark MF used as part of java action as used in the MF properties.

Global Variable Abuse:

Anti-Pattern: Overusing global variables as a means to pass data between microflows.

Issue: Reduced maintainability and code that is harder to understand.

Solution: Use input/output parameters in microflows and minimize global variable usage.

Inadequate Error Handling:

Anti-Pattern: Neglecting error handling, leading to unexpected application behaviour and poor user experience.

Issue: Unhandled exceptions and user frustration.

Solution: Implement comprehensive error handling, including feedback to users and logging.

1. No error handling (Default Behaviour)
2. Custom
    a. With Rollback
    b. Without rollback
    c. Continue

Where to use error handlings

Thumb Rule – Any external integrations should be handled.

1. Emails Notifications
2. All REST calls
3. Account creations or any important business logic.

Ignoring Mendix Best Practices:

Anti-Pattern: Disregarding Mendix best practices and guidelines.

Issue: Reduced application quality, performance issues, and difficulties in collaboration.

Solution: Familiarize yourself with Mendix best practices and follow them consistently.

Overuse of Microflow Listeners:

Anti-Pattern: Using microflow listeners extensively, leading to complex and hard-to-maintain logic.

Issue: Reduced readability and increased complexity.

Solution: Use microflow listeners sparingly and for specific use cases where they provide clear benefits.

Excessive Custom Widget Development:

Anti-Pattern: Developing custom widgets for functionality that can be achieved using standard Mendix widgets.

Issue: Increased development effort and potential compatibility issues.

Solution: Leverage built-in widgets and use custom widgets only when necessary.

Lack of Documentation:

Anti-Pattern: Failing to provide adequate documentation for microflows, modules, or the application as a whole.

Issue: Difficulty in onboarding new team members, understanding existing logic, and maintaining the application.

Solution: Maintain clear and up-to-date documentation for your Mendix project.

Avoiding these anti-patterns and adhering to Mendix best practices will contribute to the creation of maintainable, efficient, and high-quality Mendix applications. Regular code reviews and adherence to a set of coding standards can help identify and mitigate these issues during development.

Delete Of Attributes and associated objects with event handlers

Feedback

Please enter your feedback to include more topics here.

Logging & Monitoring

Logging of errors in all the services/MFs should be done in the same node in the same format.

All informational logs should be in the 'info' node and errors should be in 'error' node and not the info node.

The error format to be followed for is.

- **Schedulers** – The format of error logging in case of an error while running a scheduled event is.
    - Log node name should be: - An error while executing the schedule %ScheduleName%
    - Error Message:-
        - $latestError/ErrorType
        - $latestError/Message
        - $latestError/Stacktrace
        - [%currentDataTime%] – formatted as per the time zone of the customer (IST or Gulf time etc)

- **Microflows** – The format of microflows in case of an error while running a scheduled event is.
    - Log node name should be: - An error while executing the microflow %MicroflowName% on page 'PageName' by user [%currentUser%] with role [UserRole]

    - Error Message:-
        - $latestError/ErrorType
        - $latestError/Message
        - $latestError/Stacktrace
        - [%currentDataTime%] – formatted as per the time zone of the customer (IST or Gulf time etc)

- **REST call failure** – The format of microflows in case of an error while running a REST call fails is.
    - Log node name should be: - An error while calling the REST service from the microflow %MicroflowName% on page 'PageName' by user [%currentUser%] with role [UserRole]

    - Error Message:-
        - $latestHttpResponse/StatusCode
        - $latestHttpResponse/ReasonPhrase
        - $latestHttpResponse/Content (Optional)
        - [%currentDataTime%] – formatted as per the time zone of the customer (IST or Gulf time etc)

---------- End of Document ----------