Optimal Authentic Modernization of Heritage IBM i Software Assets

As IT has abused the fundamental, good meaning of the word "legacy," this document will use "heritage" instead. Contrary to popular opinion (especially IT opinion), legacy (aka heritage) is NOT bad. Many, if not most IBM i users can directly attribute their success as a company, to the use of their commercial applications and the fantastic, most successful commercial computing platform ever, it runs on.

As there is such a demand for a pragmatic solution and approach to address the concerns and potential disruption of true modernization, it is critical to recognize that a multi-faceted, multi-dimensional plan and process is required to satisfy this crucial requirement.

Ignoring ANY of these disciplines WILL result in less than desirable results. The highlighted restraints will inhibit an organization from remaining competitive and staying relevant as a business.

New entrants, with radically reduced barriers to entry, can easily disrupt most industries. Established companies must ensure their AGILE response to changes in the business environment or face their demise.

Established companies, with the albatross of technical debt stifling their AGILE response to changes in the business environment, are at severe risk. *("Technical debt is a useful metaphor for thinking about the shortcuts, workarounds, and postponed decisions that accumulate in a technology stack…"* – Ward Cunningham)

DATA and modern software engineering principles allow new entrants to act swiftly. Technology lowers barriers to entry in most industries significantly and allows for small, dynamic, AGILE players to completely redefine established business models. (Consider for instance the disruption of Amazon on many established industries globally since 1995).

Your business data, the leveraging of your heritage data, and the advances in data science (which underpins 4IR, cognitive computing, AI, etc.) are critical requirements going forward. Hence, the most vital, foundational consideration for your heritage commercial applications is the data, and how to unlock the treasure trove.

Business and transactional data is the LIFEBLOOD of ANY commercial entity today, and ALL impediments both within the data, as well as the underlying data platform and architecture, MUST be removed or mitigated, to provide a solid foundation to implement latest advances in data science.

Consider the accuracy of the preceding statements, assessing the advances in the three software engineering disciplines the past ten to fifteen years. (The three dimensions are: **M**odel = Database, **V**iew = “delivery channel’ and **C**ontroller = Logic or “orchestrator” of components

As a result of the preceding, we strongly advise all development installations to implement a strategic plan for each of the identified disciplines, prioritizing and allocating resources based on quick wins and tactical (primarily competitive pressures) commercial realities. But long term, clearly defined strategic objectives should be paramount throughout, to ensure continued focus on the end objectives.

Document Version: 2020-1
Publication Date: 2020/05/05

Short term tactical gains can usually be achieved by primarily focusing on the delivery channel ({**V**iew}, e.g., microservices or web-facing selected application functions) and modernizing the codebase. Investing too much effort too soon on the UI/UX (hereafter collectively referred to as the "delivery channel") may likely result in even more technical debt, caused by "double maintenance" and solidifying technical debt. It is, however, imperative to continuously assess your delivery channel and facilitating easy integration with your consumers.

These delivery channel efforts should, however, not detract from the focus on the underlying data and data "store" (DB2 for i), as most value and impact wait to be unleashed in this critical area. The benefit, impact, longevity, but especially AGILITY of leveraging the data/database and adopting data centricity, will be by orders of magnitude more valuable to the business. It also positions you best for the future.

We, therefore, propose a multi-pronged strategy on all three fronts, dictated by operational (tactical) demands, but continuously guided by strategic business objectives. A delicate balance is required between tactical and strategic plans to ensure investments deliver the necessary long-term ROI while providing short term tactical gains.

Fundamental to such a strategic project, is the ABSOLUTE support and "sponsorship" by the C-Suite, to ensure the required attention and protection against conflicting operational requirements. The "sponsorship" is paramount, as the temptation is simply too severe to redirect resources due to operating pressures. The perceived lack of immediate benefits working on strategic imperatives (data centricity, application architecture, and supporting software components), risks easy re-tasking. A work environment is necessary where developers are provided the freedom to assimilate new approaches and coding practices. A percentage of developers' time must be protected (usually in the 10% range) to become intimately familiar with the modern way of developing applications on IBM i. This percentage can (should) gradually be increased as RESULTS implementing the new approach, familiarity, and relevance to the specific application functionality and business are better assimilated.

Characteristics of a modern IBM i commercial application

We believe a modern, native IBM i commercial application should adhere to the following characteristics:

1. Fully leverage the DB2 database engine, requiring the database engine to enforce ALL data validations and relationships between entities (tables) within the database.
2. All table (file) and column (field) names using descriptive commercial nomenclature, for easy reference and use by end-users, without technical interpretation.
3. Central metadata repository ("data dictionary") providing a single definition of all data elements, their attributes, and where it is used.
4. All data selection, filtering, and access managed by the database engine.
5. The absolute separation between the database (**M**), application logic (**C**), and the delivery channel (**V**).
6. Single instance, finite, reusable source, and object code components leveraging the ILE programming model to the fullest extent possible.
7. Total exploitation of inherent, unique IBM i operating system features, such as integrated security, work management, integration with DB2 for i, journaling, commitment control, and workload virtualization.
8. Discrete business functions with likely partner and consumer interactions decomposed into the smallest sensible components (read microservice). Care must be exercised here.

Document Version: 2020-1
Publication Date: 2020/05/05

9. AGILE, dynamic developer response to changing requirements.
10. The application must support multiple "clients."
11. API for accessing data and data services over HTTPS.
12. Data is delivered in a generic, consumable format (JSON).
13. Stateless processing between the delivery channel (client) and backend (IBM i based) application services.
14. Web interfaces leveraging CSS3, HTML5, JavaScript and JSON, implementing responsive frameworks.
15. Full exploitation of the latest developments in software engineering deployment strategies to facilitate the highest levels of agility in maintaining and bringing new functionality rapidly to users.

Following this approach will result in rapid improvement in modernization momentum and especially the value gained from that.

The following software dimensions of delivering modern commercial OLTP applications for IBM i need detailed strategic plans:

1) Database and Data Centricity (M)

Fundamentally, ALL lasting modernization efforts must start at the database. Any other modernization efforts are tactical at best, however important they may seem. The transactional data and business rules encapsulated within your applications are the LIFEBLOOD of your company or employer.

With the immense importance of DATA in most, if not all, advances in computer science and software engineering, especially the past two decades, the value of DATA and DATA CENTRICITY in commercial application development cannot be stressed too severely. Companies that cannot manage their data (data governance) and especially the QUALITY of their data will simply drown in the deluge (think IoT, big data, etc.) of data generated in the current data-driven world.

Foundation to this crucial strategy is the approach recommended by http://datacentricmanifesto.org/principles/, which provides a pragmatic framework of considerations behind our motives and strategy.

We advise on the following iterative process to gradually adopt and implement true data centricity within your heritage application database without disrupting your standard production processing:

1. Redefine as many as possible of all DDS defined PF's (physical files) into DDL (Data Definition Language) tables, as the foundation for all database definitions going forward. During this initial, first step, it is of paramount importance that absolutely NO LVLID changes must occur, as few, if any, installations can afford to recompile all impacted programs. This change must be absolutely transparent to the applications and users.
2. Define and publish modernization standards, architecture, and naming conventions as soon as possible to guide all modernization (maintenance) efforts.
3. Implement a central data dictionary or metadata repository as a matter of priority, to become the SOURCE of all data elements and especially the significant sanitization process ahead. All data elements, inconsistencies in definition, and duplication must be

identified and corrected as an integral part of modernization going forward. Data quality and data integrity is a paramount consideration in the data-driven world today.

4. Rapidly introduce SQL Long Name/Alias both at Table (PF) and Column (field) level, as this will remove negative perceptions from your users who have difficulty relating to short, abbreviated file and field names that most developers take for granted. Your naming conventions and standards for your SQL long names must be succinctly (clearly, briefly) defined, as the perceived freedom with long names can very quickly devolve into something less than desirable.
5. Adopt the standard philosophy ***"you touch (maintain) it, you modernize it"*** for all developers, initially allowing more time to developers to become completely "au fait" with the recommended approach and process.
6. Gradually introduce standard error recovery routines throughout the entire application portfolio, consistent with how the operating system and database treats error handling. This (standard error recovery) is a crucial step, as the next steps may cause software failures due to incomplete and/or inaccurate and/or orphan records in the database. These "failures" need to be trapped, logged, and managed entirely transparently to the users and the applications. Please see https://www.ile-rpg.org/open-source.html for more detail - especially the ***"Pulling the (DB2) Trigger (Without shooting yourself in the foot)"*** article. Doing this will ensure uninterrupted transactions processing, while your team remove the accrued technical debt within the applications and sanitize the data in your database (orphan records and invalid data).
7. Publish detailed standards and guidelines on database KEY definitions for the modernized database (see "***Natural versus Surrogate KEYS – The "Key" to Success in Database Modernization*** "for some background).
8. Where a Primary key exists on the PF's, AO will introduce a *PRIKEY constraint definition on those DDL table definitions, subject to the following considerations:
    a. Unique keys are defined on a DDS defined Physical File by a combination of K-Specs and the "UNIQUE" keyword. Without the unique keyword, the key becomes non-unique, and without the K-Specs, this results in an arrival sequence access path.
    b. DDL (SQL) defined tables do not support non-unique access paths and do not recognize the K-Spec/UNIQUE combination as a *PRIKEY for the file. Therefore, when AO re-creates any PF or Table, it attempts to create it as an arrival sequence DDL defined table and will add a *PRIKEY constraint if a unique key was previously described.
    c. There are several conditions in which a DDS defined physical file cannot be converted to DDL, the non-unique key being one of them. In these cases, the DDS definition is retained. If the file was defined with a unique key, then the K-Specs and UNIQUE keyword is dropped, and a *PRIKEY constraint is added in their place. Only non-unique keys are retained in the K-Specs within the DDS under management by AO.
    d. Primary keys are, by definition, unique and are required on ALL Tables/Physical files making up a database to allow for the successful implementation of a relational database philosophy. This (relational database) uses *REFCST constraints, to establish appropriate parent-child relationships between tables. Built-in relationships provide a significant degree of integrity and control, internal to the database structure.
    e. All "Parent" files in a relational database are required to have at least one unique (Primary) key; otherwise, no dependant relationship can be established. If a dependant/child file (foreign key file) requires to be related to a parent where the *PRIKEY does not provide a suitable unique identity, then additional

unique definitions can be added to the parent by use of the *UNQCST definition.

f. If uniqueness on a parent file cannot be achieved by either a *PRIKEY or *UNQCST, then it is highly likely that further normalization of either the parent or the dependant file is required. Of course, it applies only to potential parent files. Files lower down the hierarchy, (usually transaction files), should have a primary/unique key but will never become a parent file.

g. Where the primary/unique keys are implemented via logical file definitions (LF's {logical files}, Indexes, or Views), MOVE the primary/unique keys into the base table definitions as *PRIKEY and *UNIQUE constraints, and phase out the use of that LF gradually.

9. Refine and review all KEYS, against the *PRIKEY, *REFCST, and *UNQCST requirement going forward (from a consistency perspective, we recommend all validations to be performed by using *BEFORE triggers and therefore to not use *CHKCST).
10. Start with the process of incrementally normalizing the data model to have NO duplicate data elements (as little) as practically possible eventually.
11. Gradually start with the process of implementing I/O Servers, to separate the application from the database. The ultimate objective is to remove all F-specs from application programs and have an ABSOLUTE separation between the database and logic ("controller") layer.
12. Identify and consolidate duplicate metadata entries and implement the sanitized entries back into the base table definitions. In our heritage applications, we used to implement significant duplicated fields (columns), using different prefixes or suffixes to identify the fields in the logic of our application uniquely. With the latest advances in the ILE programming model and RPG, there is no need for this any longer. Duplicate and inconsistent entity attributes cause SERIOUS data quality and data integrity issues. The objective is to eventually remove ALL duplicate fields and inconsistencies from the production database.
13. Adopt and implement standard data types, especially concerning international date and time data types, and gradually implement this back into your table definitions.
14. Introduce referential constraints between related entities.
15. Improve your data access, by implementing suitable permanent logical constructs (LF's, Joins, Views, Indexes) that will leverage the database engine to select and deliver result sets to your HLL programs for processing.
16. Move ALL data validation routines gradually out of all your application code into the database engine by way of *BEFORE triggers.
17. Implement all subsequent updates and database synchronization by way of *AFTER triggers.
18. Repeat the preceding process until your entire database conforms to 3NF principles; all tables/PF's have *PRIKEY constraints, and the database engine enforces all dependencies between entities (tables) by way of *REFCST. Ensure that no (as few as possible) duplicate fields (columns) exist in the database, with a clear indication of which field represents the absolute truth if duplicates are necessitated for valid reasons.
19. Ensure that eventually, all database record and column selection is performed by the database engine, with an optimal "access plan."
20. Ensure that ultimately, all database IO requests, regardless of its origin, is processed by I/O Servers.

2) Code (C)

Fundamentally, the primary objective of the code modernization dimension revolves around:

1. The adoption and rapid implementation of the ILE programming model and moving all fixed form RPG to "fully free" form code. Both these initiatives have a significant impact on securing and retaining young talent. Hand in hand with this requirement is the adoption of a modern IDE (Integrated Development Environment), as the older tooling does not support the latest enhancements to the language. Still, more importantly, companies will not attract young talent using antiquated software development tools.
2. Development teams should implement "separation of concerns" (the de-coupling of the database from the application's logic by removing F-specs and separating/wrapping all the "delivery channel" code) as a matter of priority. This (separation of concerns) will improve responding in a more agile fashion to changes in the business environment.
3. Standardized error handling must be implemented in all applications, regardless of where the functionality will be delivered (read "delivery channel"). This (standard error handling) is essential, due to behavioral differences between SQL and RPG error handling and message formats and especially when data centricity is fundamental to the business.
4. Identification of validation routines, IO, referential integrity, and "common" routines.
    a. All validations should gradually be moved out of the code into the associated trigger programs for that file and/or data element.
    b. Standard I/O Servers and Enterprise Services must be introduced, to gradually de-couple the IO from the code.
    c. Old code enforcing file relationships should be gradually phased out, once those relationships are applied by the database engine.
    d. Everyday routines (for instance, tax calculations, freight handling, volume discounting, route planning, etc.) should be gradually consolidated and moved into single instance modules in service programs.
5. A primary objective of modernization is to be more agile in response to rapidly changing business environments. As much duplication (read double maintenance) of functions, both within source code and in resulting objects, must be removed or consolidated. The objective is to achieve "single instance objects AND source." Code re-use is critical to achieving the required quality and rapid response to changes in the business environment.

3) User Interface aka "Delivery Channel" (V)

It is critical to acknowledge that regardless of the merit and benefits of the remarkable character-based interface of IBM i; its continued use is partly responsible for the perception that our applications and the platform are outdated.

Foundationally, using the standard 5250 character-based interfaces, application programs dictate the response to the user.

In the modern world, the objective is to allow (guide) the user to define their business objective and then respond with associated functionality. The contemporary requirement is much more of an "event-driven" paradigm than what 5250 usually delivers.

It is indeed possible to use one of the myriads of "screen scraper" solutions to deliver functionality to the web or any of the other required "delivery channels.

However, "screen scraping" must be acknowledged as a short term, tactical solution at best. The same requirement that drives database modernization demands that the "delivery channel" be de-coupled from the code. Application functionality should be delivered to any potential device or service, regardless of location.

If screen scraping technology becomes more than a short term, tactical solution, it causes technical debt to be solidified. It will result in the demise of the application much, much faster. The technical debt will be glaringly exposed due to increased transaction volumes and will cause C-Suite disillusionment.

An open-ended architecture is necessary, to deliver a solution, responding with equal agility to changes in business requirements, as the two preceding (modernized database and code) dimensions provide. The following technical conditions exist to deliver on this need:

1. Separation of the "delivery channel" from both the Database (M) and Code (C).
2. As few platforms as practically possible based on business and security requirements, to deliver the complete stack. This is especially true of SMB installations, where lean (yet highly specialized) skillsets are a reality.
3. No double maintenance.
4. Native, RESPONSIVE Web interfaces.
5. Use of JSON as a delivery mechanism for result sets.
6. Use of HTML5 and CSS3 on the application web pages.
7. Use of JavaScript for richer UX in selected areas.
8. Microservices and API components, for those business functions exposed to the outside world, to deliver a better solution.

Based on the immense importance of quality data in the data science-driven world today, it is clear that any authentic modernization of our heritage applications MUST start at the database. Not using the incredibly powerful database to its maximum, ensuring quality, real-time data is a recipe for disaster and the demise of your commercial application, which will ultimately have a severe negative impact on the company.