

DISCOVERING **THE BEST** CROSS-PLATFORM FRAMEWORK THROUGH BENCHMARKING

Delphi & Electron for Desktop Platforms

1 December 2021 Version 1.0



Authors

Adam Leone, Jim McKeeth and Eli M.



EMBARCADERO TECHNOLOGIES
10801 NORTH MOPAC EXPRESSWAY
BUILDING 1, SUITE 100
AUSTIN, TX, 78759
EMBARCADERO.COM

Executive Summary

When businesses choose a software framework they begin a long-term relationship for the duration of their application's lifecycle. Given the strategic consequences of this decision, businesses must carefully consider how frameworks enhance developer **productivity**, business **functionality**, application **flexibility**, product **performance**, the long-term viability of that framework, and the inherent security in each framework's design and technology. The ideal framework demonstrates strength in each category by minimizing product time-to-market, reducing maintenance costs, supporting product variety, and facilitating a superior customer experience.

This paper evaluates two frameworks supporting multi-platform desktop application development: **Delphi** and **Electron**.

In order to assess these frameworks, this paper defines four evaluation categories and 26 metrics, defines and analyzes a benchmark application, and scores each framework using a weighted evaluation. The benchmark, an RSS reader, assesses each frameworks' ability to create an application for multiple desktop operating systems and provides insight into framework performance differences between operating systems.

Evaluation conclusions include:

- 1) Delphi and its RAD Studio IDE profoundly enhance development flexibility and product time-to-market in agile environments
- 2) Delphi's runtime and network library performance exceed that of the competition and requires substantially less overhead from dependency file sizes and startup times
- 3) Electron bests Delphi on entry costs, and although like Delphi it does offer cross-platform capability, it lags behind in application performance and intellectual property security

Framework Comparison			
Categories	Weight (%)	Delphi	Electron
Productivity	25.2%	1.10	0.89
Functionality	23.5%	0.96	0.83
Flexibility	31.3%	1.57	1.14
Performance	20.0%	0.81	0.79
Weighted Score (5 is best)	100%	4.43	3.64

Table 1 - Weight Evaluation Summary

Table of Contents

Executive Summary	2
Table of Contents	3
1. Introduction	5
2. Related Work	5
3. Methodology	6
3.1. Evaluation Categories	6
3.2. Metrics	7
3.2.1. Productivity	7
3.2.2. Functionality	8
3.2.3. Flexibility	9
3.2.4. Performance	10
3.3. Frameworks	10
3.3.1. Delphi	10
3.3.2. Electron	11
3.4. Evaluation Strategy	11
3.4.1. Benchmark Application	11
3.4.2. Measurement Tools	12
3.4.3. Weight Profiles	13
4. Analysis	15
4.1. Productivity	16
4.2. Functionality	18
4.3. Flexibility	19
4.4. Performance	20
5. Conclusions	24
6. Future Work	25
References	26
Appendix 1. Benchmark Application Specification	27
Appendix 2. Detailed Framework Analysis	34
Appendix 3. Framework Decompilation Analysis	40
Appendix 4. Contributors & Sponsor	44

Disclaimer

Effective benchmarks are tricky at the best of times, and often very divisive. We've done our best to be objective and fair in this paper. The methods and sources of our research are listed. We encourage you to examine them, reproduce our tests, and do supplemental research. If you find different results, or disagree with our analysis we would love for you to share your results with us and the world.

All the sources are available on GitHub

github.com/Embarcadero/ComparisonResearch/tree/main/unicode-reader

Copyright © 2021

Embarcadero Technologies



This paper is the result of collaborative research sponsored by Embarcadero Technologies.

See [Appendix 5](#) for a full list of contributors.



This work is licensed under Attribution-ShareAlike 4.0 International

This is a human-readable summary of (and not a substitute for) the [license](#).

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

<http://creativecommons.org/licenses/by-sa/4.0/>

1. Introduction

Annual surveys of personal computer users over the last decade detail a substantial shift in operating system preferences for personal and business devices. While Microsoft's Windows once stood monolithic, capturing 92% of operating system installations as recently as 2010, it now controls just three-quarters of the desktop and laptop market. In the span of ten years, Apple's macOS has grown 173% from its 2010 market share to secure 17% of users, and Linux has doubled its footprint to nearly 2% of non-server systems in daily use.¹ This accelerating diffusion of users into incompatible ecosystems poses a challenge for developers accustomed to reaching most of the market with one product. It can, however, be surmounted using a software development framework designed to target all major desktop operating systems. Given the many options available, developers pivoting into multi-platform applications should select a framework that measurably maximizes product reach and provides outstanding productivity, functionality, flexibility, and performance. While this choice is ideally informed by solid research, popularity trends within the industry and interest in the next "shiny" tool undeniably influence framework selection. This paper strives to counter such subjective factors by refining an established benchmark methodology, applying it to Delphi and Electron using a multi-platform Really Simple Syndication (RSS) application, and drawing conclusions about the development utility, business advantages, and performance of each framework.

This paper is structured as follows: Section 2 introduces the history of developer tools, discusses related academic tool comparisons, and summarizes the first white paper in this series. Section 3 describes the broad methodology of this comparison in four categories, lays out specific metrics per category, introduces the frameworks under comparison, and describes the evaluation benchmark application and weighted comparison. Section 4 analyzes the metrics by category, and Section 5 draws conclusions. Additional information and detailed analyses are available in Appendix 1-3.

2. Related Work

It should come as no surprise that software developers and academics have conducted tests and comparisons since the world's second framework was released. A time-honored method of comparison is the *benchmark*. Formalized in the 80's, Robert Camp's widely accepted definition of a benchmark is "the search for the best industry practices which will lead to exceptional performance through the implementation of these best practices".² Software industry benchmarks most often take the form of quantitative performance tests and evaluations, such as task completion speeds, but can also be qualitative via scoring systems and weighted assessments. The key to any benchmark is to apply the test equally to *like systems* from different companies, avoiding an "apples-to-oranges" comparison, and to incorporate the objective results into business decisions.

¹ "Desktop Operating System Market Share Worldwide". Statcounter Global Stats. Accessed 20 January 2021. <https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-201012-202012>.

² Camp, Robert C. *Benchmarking: the search for industry best practices that lead to superior performance*. Asq Press, 1989.

Since 2015, academia has produced over a dozen comparison papers³ focusing with increasing intensity on cross-platform frameworks for mobile application development, seeking to fill that gap in academic literature.⁴ While each paper's focus varied, many used performance benchmarking techniques and qualitative assessments to compare applications written to the same level of functionality in different frameworks.⁵ Seeing that recent literature lacked evaluation cohesion and consistent metrics, Rieger and Majchrzak proposed a detailed evaluation schema for future comparisons.⁶ Their four-category, 31 metric schema and corresponding weighted evaluation table provide an excellent starting point for both qualitative and quantitative comparisons.⁷

Building upon Rieger and Majchrzak's work, Embarcadero Technologies published *Discovering the BEST Developer Framework through Benchmarking*⁸, a white paper examining Windows application development using Delphi, Windows Presentation Foundation with the .NET Framework, and Electron. That paper derived a weighted evaluation from the Rieger and Majchrzak template, defined metrics in four categories spanning the software development and business lifecycle, and tested each framework's implementation of the benchmark application - a Windows 10 calculator clone. The analysis focused on graphical-user-interface (GUI) creation as a means to measure developer productivity and framework runtime performance. Its conclusion, derived from quantitative measuring and qualitative research, ranked Delphi as the best framework for products with long lifecycles and wide market reach followed by Electron with WPF a distant third.⁹ This paper seeks to extend the framework comparison effort by assessing Delphi and Electron in a cross-platform desktop environment with emphasis on native database libraries and network interactions.

3. Methodology

3.1. Evaluation Categories

Each framework supporting Windows, macOS, and Linux development will be evaluated according to the four aspects of effectiveness defined in *Discovering the BEST Developer Framework through Benchmarking*: developer productivity, business functionality, framework application flexibility, and end-product performance.¹⁰ Combined, these aspects examine a framework's impact on the business

³ Majchrzak, Tim, and Tor-Morten Grønli. "Comprehensive analysis of innovative cross-platform app development frameworks." In *Proceedings of the 50th Hawaii International Conference on System Sciences*. 2017, Table 1.

⁴ Bjørn-Hansen, Andreas, Tim A. Majchrzak, and Tor-Morten Grønli. "Progressive web apps: The possible web-native unifier for mobile development." In *International Conference on Web Information Systems and Technologies*, vol. 2, pp. 344-351. SCITEPRESS, 2017, p. 349.

⁵ Willocx, Michiel, Jan Vossaert, and Vincent Naessens. "A quantitative assessment of performance in mobile app development tools." In *2015 IEEE International Conference on Mobile Services*, pp. 454-461. IEEE, 2015, pp. 455-456.

⁶ Rieger, Christoph, and Tim A. Majchrzak. "Weighted evaluation framework for cross-platform app development approaches." In *EuroSymposium on Systems Analysis and Design*, pp. 18-39. Springer, Cham, 2016, p. 8-14.

⁷ Rieger and Majchrzak, 2016, p. 16.

⁸ McKeeth, Jim, Adam Leone, and Eli M. "Discovering the best development framework through benchmarking." Austin: Embarcadero Technologies, 2020.

⁹ McKeeth et. al., pp. 20-21.

¹⁰ McKeeth et. al., pp. 5-6.

throughout the product life cycle. An excellent framework accelerates product development but should also nurture a maintainable codebase while easily pairing with 3rd party tools. Furthermore, the frameworks considered in this paper must offer cross-platform options that broaden business opportunities without requiring new skills or development work.

Developer productivity is the measure of effort and code required for developers to complete typical development tasks. Productivity directly impacts product time-to-market and long-term labor costs so tools that increase developer productivity have substantial impacts on business timelines and bottom lines. Productivity can be realized in two distinct ways - reduced coding requirements due to native libraries, and IDE tools like code-completion and visual design. IDEs with greater library breadth generally result in fewer lines of code per application and produce a clean, lean codebase that minimizes opportunities for bugs or maintenance problems later in the product life cycle.

Business functionality refers to a framework's business suitability and impact on long-term plans. Excellent functionality allows companies to easily build custom tools or extensions, develop on a platform of their choosing, protect their source code from exploitation, and have confidence that their applications will be maintainable for decades.

Application flexibility assesses the breadth of tasks addressable with the framework. While IDEs and frameworks are Turing-complete, and thus technically infinitely flexible, some are better suited to a task than others. Flexible frameworks allow businesses to target a broad audience, build software for every application tier, and access operating system functions and consumer hardware.

Consumers judge applications in part by runtime performance. Businesses choosing the framework with superior performance avoid customer dissatisfaction by minimizing wait times and resource use on older computers or containers/virtual machines.

3.2. Metrics

3.2.1. Productivity

Framework *productivity* will be evaluated according to the following metrics:

[P1] Development Time: Total hours spent writing the fully functional application from scratch. This measurement assesses the value a framework's productivity tools add to an average developer with no prior task knowledge. Comprehensive documentation, plentiful native libraries, code completion, and other IDE tools will allow the developer to design and build the benchmark application more efficiently than would be the case in a "standard" text editor.

[P2] UI Design Approach: Does the framework's IDE allow for graphical/visual application creation and provide a "What You See Is What You Get" (WYSIWYG) view model?¹¹ IDEs that support development through "drag and drop" components or other visual methods allow users to engage different methods of thought and creativity as they work. Visual creation through WYSIWYG editors

¹¹ Rieger and Majchrzak, 2016, p. 11.

preclude businesses from needing every version of physical hardware to view platform-native styling.

[P3] Developer Environment Tools: Does the framework IDE standard installation include auto-completion, debugging, and emulation tools? Are multiple IDEs available for the framework?¹² Frameworks with multiple development tools and a choice of IDE better support individual development preferences, techniques, and requirements.

[P4] Final Build Time: Total hours required to “speedrun” the application using a known solution. This measures the number of actions and volume of code required to complete the full application by an expert developer with perfect knowledge of a working solution. Productive frameworks reduce development time on repetitive but slightly altered tasks.

[P5] Code Size: Total lines of code the developer must write, adhering to accepted formatting and styles, to create a fully functional application. This objective measure of code volume sheds light on the difficulty of future code maintenance - more code typically requires more time to learn and troubleshoot.

[P6] Application Store Deployment: Does the framework’s IDE facilitate direct deployment to native platform application stores (i.e. iOS App Store, Android’s Google Play, Microsoft Store)? Frameworks with built-in deployment features reduce product deployment complexity, limiting errors that could occur or compound, and time-to-market for initial products and updates/bug-fixes.

3.2.2. Functionality

Framework *functionality* will be evaluated according to the following metrics:

[F1] License: Does the license allow the development of commercial applications, and at what cost? Is the licensing difficult to understand? Proprietary frameworks and tools may require one-time or recurring payments and have different levels of licenses according to the commercial applications desired.¹³ Open-source frameworks may have license-specific restrictions.

[F2] Long-term Feasibility: Does the framework have a history of stability, backward compatibility between major releases, bug fixes, and security updates?¹⁴ This metric highlights the confidence businesses can enjoy or the strategic risk they may take when choosing a framework.

[F3] Supported Development Platforms: Can application development occur on any major operating system or does the framework IDE impose limitations? This metric indicates how a framework may hinder a team without homogenous equipment.

[F4] Testing Support: Does the framework ship with a testing suite, test coverage analysis, and runtime monitoring capability?¹⁵

[F5] Tool Extension: Can the framework be extended in its own language? Frameworks that require plug-ins, extensions, or modifications to be written in a different language impose costs on businesses that require altered functionality. Rather than creating the required tool from resident

¹² Rieger and Majchrzak, 2016, p. 10.

¹³ Rieger and Majchrzak, 2016, p. 8.

¹⁴ Rieger and Majchrzak, 2016, p. 10.

¹⁵ Rieger and Majchrzak, 2016, p. 11.

knowledge, businesses may have to invest time and resources to hire an external contractor or build in-house skills in that alternate language.

[F6] Accessibility: Do programs built with the framework support the major OS accessibility features like screen readers and font size/color changes?

[F7] Intellectual Property Security: How secure is the intellectual property of the source code in a deployable project? After businesses invest resources into their projects, they face the challenge of putting their product into the hands of the public while protecting the code and techniques that produce revenue. This qualitative metric evaluates the ability of a user to access source code via decompilation.

3.2.3. Flexibility

Framework *flexibility* will be evaluated according to the following metrics:

[X1] Supported Target Platforms: How many user platforms can the framework deploy an application to? Great frameworks will support most platforms on the market, whether mobile, desktop, 32-bit, or 64-bit. Businesses benefit from multi-platform support because they can develop and maintain one codebase to reach many customers. One codebase rather than separate code for each target application reduces development time, bug potential, maintenance requirements, and time-to-market for new features.

[X2] Project Variety: Does the framework support development of different types of applications from stand-alone desktop apps to Windows services? Flexible frameworks allow developers to create mobile applications, desktop services, and everything in between.

[X3] Scalability: Can the code be partitioned into subcomponents based on architectural design? Is the framework suitable for client, middle-tier, and backend systems? Frameworks that support modularity and multiple design tiers are better suited for large enterprise applications and specialization among multiple teams working on the same project.¹⁶

[X4] Unicode Support: Does the framework support the Unicode standard in its libraries, components, and datatypes? Unicode supports nearly every language on earth, broadening the reach of business applications to an international market.

[X5] Database Access: Does the framework contain native libraries supporting database access? How many databases are supported and at what abstraction? Data persistence is critical for many applications and must be user-friendly and integrated with any good development framework.

[X6] Access to Device-specific Hardware: Does the framework facilitate access to data from device sensors (GPS, microphone, accelerometers, camera, etc.) and physical action through similar devices?¹⁷ Frameworks that “throw open the doors” to the plethora of sensors and actuators available on smart devices today create business opportunities and novel solutions to consumer pain points.

[X7] Access to Platform-specific Functionality: Does the framework allow applications to interact with the host platform's operating system and access services like file system access,

¹⁶ Rieger and Majchrzak, 2016, p. 10.

¹⁷ Rieger and Majchrzak, 2016, p. 12.

contact list, battery state, and push notifications?¹⁸ Access to core OS functions prevents code duplication, avoids presenting potentially inconsistent data to users, and provides more ways to collect and analyze information.

3.2.4. Performance

Framework *performance* will be evaluated according to the following metrics:

[R1] Deployment Requirements: What is the number and size of files for the compiled project? Larger application sizes require more storage on user devices and longer download times while numerically more files can increase deployment complexity.

[R2] Startup Time: Over 100 executions, what is the average time from command to a visible application ready for user input when started on a local machine and over a network? Frameworks producing applications with shorter start-up times facilitate good user experiences and minimize system resources required before the application is useful.

[R3] Standing Memory Usage: How much memory is required for the application to run while idle as measured by a task manager tool? Better frameworks produce applications requiring lower overhead when the user isn't actively using them, thus conserving total system resources.

[R4] Peak Memory Usage: What is the maximum memory required for the program from startup through heavy use as measured by the Windows Task Manager?

[R5] Network Utilization: What percent of the time required to retrieve RSS feeds is attributable to network use (i.e. waiting for HTTP GET responses with XML data)? A higher percentage of time spent on waiting on network responses for the same RSS feeds indicates more efficient framework implementation of network libraries.

[R6] Database Utilization: How long does the application take to retrieve RSS feed data? This paper does not seek to analyze PostgreSQL database efficiency but the total time required to pull entries may indicate differences in the efficiency of framework database libraries.

3.3. Frameworks

This paper compares Embarcadero Technologies' Delphi and the open-source Electron framework.

3.3.1. Delphi

Delphi, encapsulated in the Rapid Application Development (RAD) Studio IDE, is Embarcadero Technologies' flagship product. A proprietary version of the Object Pascal language, Delphi features graphical application development with "drag and drop" components, a WYSIWYG viewer for most mobile platforms, and robust style options including platform-standard and unique palettes that provide a fully customized look and feel. Among other features, included libraries provide GUI controls, database access managers, and direct access target platform hardware and platform operating systems. The Delphi FireMonkey (FMX) framework will compile projects to native code for

¹⁸ Rieger and Majchrzak, 2016, p. 12.

32-bit and 64-bit Windows, macOS, Android, iOS, and Linux, allowing users to develop and maintain one codebase reaching most of the market. Delphi has been available for over 25 years.

3.3.2. Electron

Electron is an open-source (MIT License), Chromium-based framework that utilizes web technologies to build desktop applications on Windows, macOS, and Linux. It is developed and maintained by GitHub, a subsidiary of Microsoft. Electron combines the Chromium-based rendering engine with a Node.js server environment. As such, the user interface for an Electron application is available via HTML5 and CSS. Generally, Electron works with most Javascript frameworks such as Angular, Vue.js, and React. The HTML5, CSS, and Javascript-based technologies found in Chromium provide a rich ecosystem of user customization familiar to any web developer. Despite its relatively young age of five years, its community boasts open source packages for database access, operating system interactions, and other common tasks.

3.4. Evaluation Strategy

3.4.1. Benchmark Application

The benchmark application for this comparison is an RSS reader supporting Unicode feeds with a PostgreSQL relational database for channels and articles storage. This application specification incorporates common tasks for which good frameworks offer productivity-improving abstractions: GUI development, database access, basic HTTP operations, XML processing for display, and Unicode support. Additionally, it evaluates the ability of each framework to create an application for multiple desktop operating systems and provides insight into framework performance differences between platforms. **Appendix 1** contains the full RSS reader specification provided to the expert contractors for each framework.



Figure 1 - RSS Reader Benchmark Application Mock-up

3.4.2. Measurement Tools

The benchmark application will be analyzed according to the qualitative metrics laid out in section 3.2. To assess productivity, initial development time [P1] is measured by the contractor and coordination website ([UpWork](https://www.upwork.com/)) and the speed implementation time [P4] is reflected in a speedrun video. Each application's source code will be examined using the framework's native IDE to count total lines of code [P5], the number of developer-typed lines according to their application instructions, and the number of lines focused on the user interface.

Several internal and third-party tools will gather data for the performance metrics evaluated in this project. Once the RSS readers developed by Delphi and Electron contractors function according to the specification, Embarcadero Technologies will direct the addition of two internal tests. The internal storage test will measure time required to download a large list of RSS feeds and store them in the database [R6]. Network time [R5] will be measured using HTTP Toolkit Pro¹⁹, a third-party network analysis tool, and removed to leave only framework processing and database access time. Because each framework's application will connect to the same instance of PostgreSQL, database time is assumed to be consistent and differences in test times directly attributable to framework processing requirements. This storage test will capture differences in each framework's implementation of networking and database access tasks. The internal retrieval test measures time

¹⁹ <https://httptoolkit.tech/>

required to access each article in the database [R5] and concatenate them into a simple HTML “newspaper”. It targets just the database libraries used in each application.

Application startup times [R2] will be evaluated with Passmark Software’s AppTimer²⁰ with a sample size of 100 iterations from both local and network locations. Finally, application memory use [R3, R4] will be measured using the MiTeC Task Manager Deluxe²¹ utility. Full instructions to download, install, and configure each of these 3rd party tools are available on the [Comparison Research/Unicode Reader](#) repository.

Windows testing will be conducted on a 2018-era desktop running Windows 10 Pro with an Intel Pentium G4560 CPU, 16GB of DDR3 RAM, NVMe solid state drive, and a wireless network connection. Linux testing will occur within an Virtual-Box Ubuntu 20.05 virtual machine hosted on the Windows desktop machine with 4GB of allocated memory and a bridged wireless network connection. macOS tests will utilize a 2012-era MacBook Pro running macOS Sierra 10.12.6 on an Intel Core i5 with 8 GB of DDR3 RAM, a solid-state hard drive, and wireless network connection. Each system will connect to an Asus RT-AC66U router with a network speed of 40+ Mb/s. Because the hardware specifications differ for each operating system, frameworks will be evaluated within each OS and relative framework performance differences assessed across the three platforms.

3.4.3. Weight Profiles

In order to facilitate the comparison of frameworks that serve similar but not identical purposes, this paper will use a weighted evaluation mechanism similar to that proposed by Rieger and Majchrzak.²² Each of the 26 metrics is given a weight between 1 and 7 points, summing to 115.²³ Frameworks will be evaluated in each category and assigned a score between 0 (unsatisfied) and 5 (optimally satisfied).²⁴ When the metric is a direct, quantitative comparison between frameworks (i.e. startup times), the objectively “winning” framework will score a 5 and the “losing” framework a 3. Ties will award each framework a 5 to negate that metric’s impact on the final scores. Once calculated, the weighted score will fall between 0 and 5 and indicate which framework best satisfies these 26 criteria. See *Table 2* for the metric weights and the emphasis placed on each evaluation category.

Framework Comparison			
Criterion	Weight (%)	Delphi	Electron
P1: Development Time	7		
P2: UI Design Approach	7		
P3: Developer Environment Tools	5		
P4: Final Build Time	2		

²⁰ <https://www.passmark.com/products/apptimer/>

²¹ <https://www.mitec.cz/tmx.html>

²² Rieger and Majchrzak, 2016, p. 15.

²³ Rieger and Majchrzak, 2016, p. 15.

²⁴ Rieger and Majchrzak, 2016, p. 15.

P5: Code Size	4		
P6: App Store Deployment	4		
Productivity Total	25.2%		
F1: License	2		
F2: Long-term Feasibility	6		
F3: Development Platforms	3		
F4: Testing Support	6		
F5: Tool Extension	3		
F6: Accessibility	3		
F7: IP Security	4		
Functionality Total	23.5%		
X1: Target Platforms	7		
X2: Project Variety	3		
X3: Scalability	5		
X4: Unicode Support	4		
X5: Database Access	5		
X6: Hardware Access	6		
X7: Platform Access	6		
Flexibility Total	31.3%		
R1: Deployment Requirements	4		
R2: Startup Time	4		
R3: Standing Memory Usage	4		
R4: Peak Memory Usage	3		
R5: Network Utilization	4		
R6: Database Utilization	4		
Performance Total	20.0%		
Weighted Score (5 is best)	100%		

Table 2 - Weighted Criteria Schema

The Flexibility category is given the most emphasis in this weighted evaluation at 31%. As with learning a new programming language, developers and companies must invest time and effort into understanding how to harness the power of a framework. They benefit most when a framework handles nearly every type of task and precludes investing in multiple limited-scope tools. Productivity follows with a total category weight of 25%. Because frameworks are tools to enhance

development, facilitate applications of increasing complexity, and bring a product to market faster, they should objectively aid rather than hinder developers. At 23%, Functionality is the third most important category because developers must be able to build, test, and extend their tools easily and enjoy confidence in their long-term security and viability. While not de-emphasized, Performance has the lowest category weight at 20% because modern computers run most desktop applications without complaint. Framework performance has greater impacts on slower hardware or computation-intensive problems, however, which opens opportunities for companies targeting those markets.

4. Analysis

Quantitative analysis of each benchmark application and expert-assisted qualitative research resulted in the following scored Framework Comparison (*Table 3*). The following sections expound upon and analyze each evaluation category. For a detailed analysis by metric, reference Appendix 2.

Framework Comparison			
Criterion	Weight (%)	Delphi	Electron
P1: Development Time	7	3	5
P2: UI Design Approach	7	5	3
P3: Developer Environment Tools	5	4	4
P4: Final Build Time	2	5	3
P5: Code Size	4	5	3
P6: App Store Deployment	4	5	2
Productivity Total	25.2%	1.10	0.89
F1: License	2	3	5
F2: Long-term Feasibility	6	5	3
F3: Development Platforms	3	2	5
F4: Testing Support	6	4	4
F5: Tool Extension	3	5	3
F6: Accessibility	3	3	5
F7: IP Security	4	5	1
Functionality Total	23.5%	0.96	0.83
X1: Target Platforms	7	5	3
X2: Project Variety	3	5	3
X3: Scalability	5	5	5

X4: Unicode Support	4	5	5
X5: Database Access	5	5	4
X6: Hardware Access	6	5	3
X7: Platform Access	6	5	3
Flexibility Total	31.3%	1.57	1.14
R1: Deployment Requirements	4	5	3
R2: Startup Time	4	5	3
R3: Standing Memory Usage	4	3	5
R4: Peak Memory Usage	3	3	5
R5: Network Utilization	4	5	3
R6: Database Utilization	4	3	5
Performance Total	20.0%	0.81	0.79
Weighted Score (5 is best)	100%	4.43	3.64

Table 3 - Weighted Criteria Schema with Framework Scores

4.1. Productivity

Framework productivity was evaluated according to six metrics that sought to capture how frameworks and IDEs improve product time-to-market. Productivity scores are found in *Table 4* and development data in *Table 5*.

Productivity Comparison			
Criterion	Weight (%)	Delphi	Electron
P1: Development Time	7	3	5
P2: UI Design Approach	7	5	3
P3: Developer Environment Tools	5	4	4
P4: Final Build Time	2	5	3
P5: Code Size	4	5	3
P6: App Store Deployment	4	5	2
Productivity Total	25.2%	1.10	0.89

Table 4 - Productivity Scores

Framework Productivity		
	Delphi	Electron
Total Development Time	31.630	67.800
Application Dev	23.300	20.000
Internal Test Dev	8.33	47.8
Final Speedrun (hrs)	1.130	0.550
Total Lines of Code	178	293
Lines of Code for UI	97	115
Non-UI % of Code	46%	61%

Table 5 - Benchmark Productivity Indicators

Product development speed from inception to delivery plays a large role in business flexibility and survival. As a stand-in for “time-to-market”, the Development Time [P1] metric measured developer hours required to bring the RSS reader from specification to tested, accepted product. Based on hours reported to UpWork by the independent contractors, initial development time indicated that each framework required roughly equal effort - 20 hours for the multi-platform Electron application and 23.3 hours for the Delphi application. Once complete, the Electron application was “speedrun” [P4] in half the time of the Delphi application despite requiring almost twice as many lines of developer-typed code. This is largely because Delphi’s IDE provides visual application development [P2] through drag-and-drop components, decreasing the complexity of GUI creation at the cost of increased time configuring components. Delphi exhibited strength in other areas, however. Its database and network code composed only 46% of developer-typed lines compared to 61% for Electron, a clear indication that Delphi’s FireDAC database library and network tools abstract those operations better than Node.JS, reducing developer effort and opportunities for error. Overall, similar results in the initial development phase made Delphi and Electron appear equivalent.

This conclusion changed after amending the specification to add internal unit tests. Once the contractors sufficiently understood the test requirements, the Delphi application was modified and accepted in 8.33 hours of work. The Electron application, on the other hand, took 47.8 hours to add the same functionality. Although the Electron developer implemented the test features in his development environment within 28.6 hours, troubleshooting a variety of Javascript and database library errors occurring on the client machines (Windows, macOS, Linux) nearly doubled the delivery time of the amended RSS reader to 47.8 hours. While it would be unwise to draw strong conclusions from a small sample size, tripled development time and fragility in dissimilar production environments points to the Electron framework’s increased complexity relative to its Delphi competitor. Agile businesses fielding complex or rapidly-changing applications would benefit from a

framework that minimizes development complexity and would be wise to test Delphi against the competition to verify these observations.

A final aspect of product development productivity is the time required to get the application to the user [P6]. Delphi scores top marks in this metric. The RAD Studio IDE automates application deployment to the app stores for all major desktop and mobile applications, eliminating the headache of manual deployment and ensuring the process is consistent. Electron struggles in this regard and can only deploy to the Microsoft Store and Mac App Store with the help of 3rd party tools. Businesses should keep this “last mile” aspect of product development and deployment in mind when selecting a framework for their application.

4.2. Functionality

Framework functionality was examined qualitatively through research on the business aspects of each framework ranging from initial investment through long-term maintenance of the products created. Functionality scores are displayed in *Table 6*.

Functionality Comparison			
Criterion	Weight (%)	Delphi	Electron
F1: License	2	3	5
F2: Long-term Feasibility	6	5	3
F3: Development Platforms	3	2	5
F4: Testing Support	6	4	4
F5: Tool Extension	3	5	3
F6: Accessibility	3	3	5
F7: IP Security	4	5	1
Functionality Total	23.5%	0.96	0.83

Table 6 - Functionality Scores

As a proprietary framework, Delphi requires businesses to purchase commercial-use licenses [F1] and offers optional annual updates for a fee. For this investment, users gain a stable, backward-compatible and growing framework with dedicated support teams and a 25-year history of success [F2]. Companies can be confident that applications developed today will be supported and maintainable in future decades, an outstanding long-term outlook in the rapidly changing world of software. Delphi ships with testing libraries [F4] and also gives businesses the opportunity to develop tools and extensions [F5] for the IDE in its native language. Some drawbacks of the framework include its Windows-only IDE [F3] and limited accessibility support [F6] for FireMonkey cross-platform applications, a shortfall that Embarcadero Technologies is working to remedy.

Electron is a free, open-source platform offering businesses the opportunity to develop applications on any major operating system without upfront costs. The price of zero up-front costs is found in the lack of expert support and amenities. The framework forgoes a native IDE, relying instead on extensions for IDEs like Microsoft's Visual Studio, and lacks conveniences such as integrated compilation, bundled testing libraries, and native language tool development. Compensating slightly for those drawbacks, Electron gains access to the many Javascript and Typescript libraries available and provides excellent accessibility options for all major desktop platforms. Business investigating Electron should keep in mind its uncertain future - at five years old the framework is still in its honeymoon phase.

Intellectual property protection [F7] is fundamentally important to long-term business plans. If a product solves a new problem or utilizes a novel technique, the developers should understand how their choice of framework affects IP vulnerability. Delphi programs compile into platform-native machine code rather than intermediate code. Decompilation using free tools can recover elements of the GUI but only yields assembly code for the logic. Electron has a fundamental IP problem - it gives away source code with each installation by default. Electron application code can be recovered with a simple text editor and only somewhat obfuscated using 3rd party tools. On the broader security front, Electron is vulnerable to cross-site scripting, one of the top ten-most dangerous web application security risks according to the Open Web Application Security Project, among other javascript-specific attack vectors.²⁵ *Appendix 3* describes such vulnerabilities and examines the results of available decompilation tools on the RSS reader applications.

Overall, Delphi provides the most assured long-term outlook, best intellectual property security, and easiest in-house customization at the cost of a one-time license purchase. Electron is absolutely free and can be developed on each of the three major desktop platforms but exacts a cost for that flexibility via its uncertain long-term outlook and reliance on corporate sponsorships and community support for additional development.

4.3. Flexibility

Framework flexibility was examined qualitatively through research and conversation with experts in Delphi and Electron and sought to analyze the application of each framework to business problems and requirements. Functionality scores are displayed in *Table 7*.

Flexibility Comparison			
Criterion	Weight (%)	Delphi	Electron
X1: Target Platforms	7	5	3
X2: Project Variety	3	5	3
X3: Scalability	5	5	5

²⁵ "Top 10 Web Application Security Risks." OWASP Top 10. Open Web Application Security Project, 2020. <https://owasp.org/www-project-top-ten/>.

X4: Unicode Support	4	5	5
X5: Database Access	5	5	4
X6: Hardware Access	6	5	3
X7: Platform Access	6	5	3
Flexibility Total	31.3%	1.57	1.14

Table 7 - Flexibility Scores

Delphi's major advantage in the flexibility category is its ability to deploy one body of source code to any major desktop or mobile platform [X1] as a native binary executable, maximizing application market reach while minimizing maintenance/upgrade headaches due to code duplication. The framework supports projects of every scale [X3] from logic controllers for industrial automation to world-wide inventory management and functions within every tier [X2] from database-heavy back-ends to client-side services. Finally, Delphi's standard libraries provide simplified access to most database products [X5], fully support Unicode [X4] and other modern standards, and broaden access to operating system functionality [X7] on every platform as well as I/O devices and sensors [X6].

Electron is an open-source framework targeting all desktop operating systems through its Chromium base. It typically focuses on web-centric, client-side applications but can accomplish middle-tier and database services using runtimes and libraries like node.js and node-postgres. Hardware access and limited operating system interactions are provided by node.js libraries and Electron's Chromium core ensures compliance with modern Unicode standards.

After reviewing both frameworks, Delphi holds the lead in the flexibility category due to its flexible and automated deployment to all major platforms, scalability to every level of development, and visual design system. Electron enjoys a lower barrier-to-entry and more development tool options but requires manual deployments and lacks the same hardware and operating system access of its competitor.

4.4. Performance

Delphi and Electron applications were evaluated according to startup times, peak and idle memory use, file numbers and sizes, and internal test results. Performance scores are found in *Table 8*.

Performance Comparison			
Criterion	Weight (%)	Delphi	Electron
R1: Deployment Requirements	4	5	3
R2: Startup Time	4	5	3
R3: Standing Memory Usage	4	3	5

R4: Peak Memory Usage	3	3	5
R5: Network Utilization	4	5	3
R6: Database Utilization	4	3	5
Performance Total	20.0%	0.81	0.79

Table 8 - Performance Scores

Framework Startup Times (sec) and Deployment Sizes		
	Delphi	Electron
Deployed File Size (MB)	32.6	296.0
# Deployed Files	2	12,300
Startup (local) (s)	0.240	1.371
Startup (network) (s)	0.473	49.875
Fastest startup (local) (s)	0.094	1.228
Fastest startup (network) (s)	0.221	10.495
Slowest startup (local) (s)	1.202	4.181
Slowest startup (network) (s)	18.830	74.334
Startup Std Dev (local) (s)	0.132	0.135
Startup Std Dev (network) (s)	1.478	3.790
Peak Memory Use (MB) (s)	109.9	98.8
Idle Memory Use (MB) (s)	75.7	42.5

Table 9 - Benchmark Performance Indicators

Application deployment requirements [R1] offer insight into both user application management and performance on networked devices. RAD Studio deployed the Delphi RSS reader to each platform as one executable file averaging 32.6 MB but weighing in as small as 18 MB for Windows. Combined with the initialization SQL script, users manage just two files. These minimal file sizes directly impacted application start-up times on Windows. The Delphi reader was ready for use [R2] in 0.24 seconds from a local hard drive and just 0.473 seconds from a network drive on average, with a slowest time of 18.83 seconds from the network. In comparison, Electron deployed the application as a package of 12,300 files totalling 296 MB - the size of the RSS reader on top of node.js

libraries and the Chromium engine. Ten times the size of the Delphi reader, the Electron RSS reader clocked in nearly an order of magnitude slower with a local startup average of 1.37 seconds and network average of 49.875 seconds. With an observed maximum network startup time of 74.3 seconds, Electron demonstrated a worst-case user experience four times slower than Delphi on Windows 10. In today's world, this longer wait may be the difference between a productive and satisfied patron and a vocally unhappy former-customer.

As seen in *Table 9*, Electron bested Delphi's FMX framework in the memory use metric for the test application [R4], consuming 11% less memory when running storage tests and 44% less at an idle [R3]. Different applications will have different memory footprints and would need to be individually tested and optimized based on the needs of the deployment, the libraries used, and UIs custom to each application. In addition, businesses considering developing Electron applications should focus on market segments that will launch the app locally. Delphi does not share those restrictions and is suitable for both desktop computers and mobile devices, where startup times and lightweight installations are critical.

The RSS reader benchmark application was specifically designed to test database frameworks and network libraries. The built-in storage test ordered after initial product delivery captured elapsed time from the start of the test until every RSS feed had been requested and all articles stored in the PostgreSQL10 database. Because this measurement incorporates network time, HTTP Toolkit simultaneously captured the network traffic and the HTTP GET/response durations subtracted from the total to yield framework processing time.

Windows Performance Tests		
	Delphi	Electron
Storage Time (s)	18.318	51.845
Framework (s)	2.988	25.999
	16.31%	50.15%
Network (s)	15.330	25.846
	83.69%	49.85%
Retrieval Time (s)	0.106	0.1588

Table 10 - Windows Performance Tests

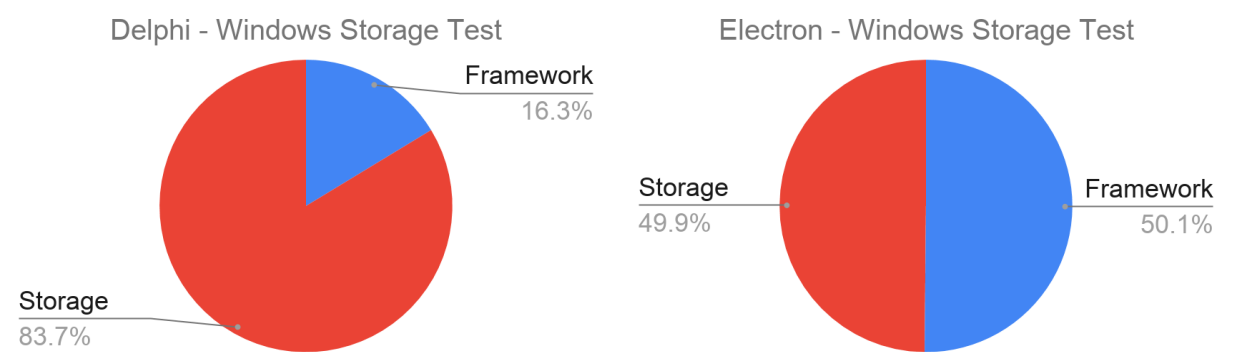


Figure 2 - Windows Framework vs. Network Time

Table 10 and Figure 2 clearly show a vast difference in the absolute storage test averages on Windows [R5]. Electron took nearly three times longer than Delphi to complete the test, averaging 51.8 seconds with a standard deviation of nine seconds. HTTP captures for the same 7.2 MB article payload revealed a 40% network time difference between the frameworks, a factor outside the applications’ control. More telling is that Delphi spent 16.3% of the storage test performing network and database operations whereas Electron required 50.4% of the total time for that purpose. The built-in retrieval test, a simple database query and file output to gauge framework database abstractions [R6], helps further break down the framework time. Ten tests per application on the same data resulted in relative parity - 106 milliseconds for Delphi and 158 milliseconds for Electron. Because both applications used the same database instance, FireDAC and node-postgres seem roughly equivalent with only a slight performance lead for Delphi. Applied to the storage test, the wide gap between framework processing times seems largely attributable to their network library implementations and other processing overhead.

Linux Performance Tests		
	Delphi	Electron
Storage Time (s)	33.823	57.484
Framework (s)	17.052	28.770
	49.59%	50.05%
Network (s)	16.772	28.715
	50.41%	49.95%
Retrieval Time (s)	0.157	0.2484

Table 11 - Linux Performance Tests

macOS Performance Tests		
	Delphi	Electron
Storage Time (s)	21.021	28.935
Retrieval Time (s)	0.323	0.1188

Table 12 - macOS Performance Tests

Performance testing on Linux saw Delphi hold the lead in both the storage test and retrieval test but with an interesting change. Per *Table 11*, Delphi's framework processing increased to 50% of the total storage test time, the same as Electron. This increase in internal processing likely contributed to the smaller performance advantage - Delphi was 64% faster than Electron on Windows but only 41% faster on Linux. This lead shrank to 27.5% on macOS, although HTTP Toolkit was unable to capture network traffic to determine if network performance differences were a significant factor. Despite that setback, the macOS retrieval test results showed Delphi querying the database 70% slower than Electron. Reduced FireDAC efficiency on macOS likely accounts for a component of the reduced Delphi lead seen in *Table 12*.

After completing network and storage testing, Delphi held a clear lead in both categories on Windows and maintained a smaller performance edge on Linux and macOS. Electron struggled to keep up, largely due to increased framework processing requirements. Businesses considering network-heavy applications should consider the operating system preferences of their target audience and lean heavily toward Delphi for the Windows market. If the segment is primarily Linux and macOS users, companies should consider prototyping their application in each framework and gauging whether performance optimizations in Electron can surpass Delphi's capabilities out of the box.

5. Conclusions

This paper sought to compare Delphi and Electron - two competing frameworks for desktop application development - in the areas of developer productivity, functionality for decision-makers, flexibility for product development, and product performance. The benchmark RSS reader application developed by experts in each framework along with qualitative research resulted in several salient conclusions for business decision-makers: First, Delphi and its RAD Studio IDE profoundly enhance development flexibility and product time-to-market in agile markets or changing projects. Second, Delphi's runtime and network libraries exceed the performance capabilities of the competition and impose substantially less overhead in terms of dependency file sizes and startup times. Lastly, Electron beats Delphi on entry costs and offers cross-platform capability but falls short in application performance and IP security.

Overall, the frameworks this paper evaluated showed their strengths in different areas of product development and performance but Delphi demonstrated consistent strength across each

evaluation category and scored 4.43 out of 5 points, outperforming Electron's 3.64 points. Based on this comparison, businesses seeking to build performance-oriented products with long lifecycles and wide market reach should strongly consider investing in Delphi and its RAD Studio IDE.

6. Future Work

This study and its benchmark application examined two of the common cross-platform frameworks for desktop operating systems and focused on productivity related to GUI design, database access, and HTTP use. Future papers should work to round-out this comparison research by studying other cross-platform and mobile frameworks with capabilities similar to Delphi and RAD Studio. Website interactions through REST services and APIs, mobile development support, and operating system interactions are major functions frameworks must handle and should be considered as benchmarking tasks.

References

- Biørn-Hansen, Andreas, Tim A. Majchrzak, and Tor-Morten Grønli. "**Progressive web apps: The possible web-native unifier for mobile development.**" In *International Conference on Web Information Systems and Technologies*, vol. 2, pp. 344-351. SCITEPRESS, 2017.
- Dalmasso, Isabelle, Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. "**Survey, comparison and evaluation of cross platform mobile application development tools.**" In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 323-328. IEEE, 2013.
- Delia, Lisandro, Nicolas Galdamez, Pablo Thomas, Leonardo Corbalan, and Patricia Pesado. "**Multi-platform mobile application development analysis.**" In *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS)*, pp. 181-186. IEEE, 2015.
- "**Desktop Operating System Market Share Worldwide**". Statcounter Global Stats. Accessed 20 January 2021. <https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-201012-202012>.
- Majchrzak, Tim, and Tor-Morten Grønli. "**Comprehensive analysis of innovative cross-platform app development frameworks.**" In *Proceedings of the 50th Hawaii International Conference on System Sciences*. 2017.
- McKeeth, Jim, Adam Leone, and Eli M. "**Discovering the best development framework through benchmarking.**" Austin: Embarcadero Technologies, 2020.
- Rieger, Christoph, and Tim A. Majchrzak. "**Weighted evaluation framework for cross-platform app development approaches.**" In *EuroSymposium on Systems Analysis and Design*, pp. 18-39. Springer, Cham, 2016.
- "**Top 10 Web Application Security Risks.**" OWASP Top 10. Open Web Application Security Project, 2020. <https://owasp.org/www-project-top-ten/>.
- Willocx, Michiel, Jan Vossaert, and Vincent Naessens. "**A quantitative assessment of performance in mobile app development tools.**" In *2015 IEEE International Conference on Mobile Services*, pp. 454-461. IEEE, 2015.

Appendix 1. Benchmark Application Specification

The following specification was provided to the independent contractors that developed the benchmark application in Delphi and Electron. Embarcadero project managers answered questions as needed, updated the specification with additional details, and strove to keep each application true to the specification.

Unicode Reader

October 29, 2020

Overview

The goal of the Unicode Reader is to build an RSS reader that brings in multiple RSS feeds which contain diverse Unicode characters and save the individual posts to a table in a PostgreSQL database. The posts can then be viewed and read in the Unicode Reader. This is a prototype! The emphasis of this project is on the lessons learned during the development process and documentation phase. The end result does not require a huge amount of polish but should look reasonably similar and function as closely as possible unless otherwise noted in this specification.

The goal of building the Unicode Reader is to explore the strengths and weaknesses of the framework you are using. The app should be built once to figure out your approach. Once complete, build the app again while recording your screen. Finally, document the app creation process in a step-by-step manner (similar to a recipe - what would someone else need to know to build the app in its entirety - configuration, code, testing, etc.).

Key Features

- Database Usage
- XML Parsing
- Unicode Character IO

PostgreSQL

The PostgreSQL database should be installed on the same machine as the desktop client. The RSS reader should be loading from localhost. The drivers should all be correctly configured to support Unicode and the database and tables should also be configured to support Unicode.

Default Feeds

- <https://blogs.embarcadero.com/feed/>
- <https://blogs.embarcadero.com/ja/feed/>
- <https://blogs.embarcadero.com/de/feed/>
- <https://blogs.embarcadero.com/ru/feed/>
- <https://blogs.embarcadero.com/pt/feed/>
- <https://blogs.embarcadero.com/es/feed/>

Schemas

The schemas aren't set in stone. If you think something doesn't make sense or needs to be changed (added or removed), let us know so we can keep the schema in sync across different platforms.

Channels Schema

```
create table channels (  
    id serial,  
    title varchar(1024) not null,  
    description text not null,  
    link varchar(2048) not null,  
    constraint channels_pk primary key (id)  
);
```

Articles Schema

```
create table articles (  
    id serial,  
    title varchar(1024) not null,  
    description text not null,  
    link varchar(2048) not null,  
    is_read boolean default false,  
    timestamp timestamp default now(),  
    channel integer not null,  
    constraint articles_pk primary key (id),  
    constraint channels_fk foreign key (channel)  
    references channels (id)  
    on delete cascade  
);
```

Login Info

Username: postgres Password: postgres Server: 127.0.0.1 DBName: postgres (or just the default?)

Requirements

Layout

The Unicode Reader is broken down into three sections. The left section contains the list of feeds. The middle section contains the list of posts from the selected Feed (or All Feeds if selected). The left and middle sections are a fixed width. The right section takes up the rest of the client and displays the contents of the post itself. In theory the contents of the post should be loaded into an HTML frame since they will contain HTML. It should not load the URL of the post but load the post itself. Any links should open in the Desktop browser (target=_blank).

Light And Dark Theme

The Unicode Reader should offer a Light theme and a Dark theme. The Dark theme should be default.





「この本を読めば、Delphiの開発スピード向上が可能です。Delphiの初心者か、バージョン1からのユーザーかは関係ありません。あらゆる種類のヒント、トリック、テクニックを学ぶことができ、生産性を大幅にアップできます。」

<https://leanpub.com/codefasterindelphi>

AlisterChristieによるツイート

🇯🇵 Japanese

Deliverables

Project Items

1. Complete source code for your working Unicode Reader. Include a compiled executable if applicable or instructions for executing the code if not.
2. A video capture of the second build process. This must be in real-time (not sped up) and executed manually (without auto-typing or other speed features). The intent is to get a realistic view of the effort required to make this Unicode Reader by a competent programmer.
3. A document with step-by-step instructions that walk someone unfamiliar with your development environment, tools, and language through the process of building this Unicode Reader to its full functionality. This document can be a .docx, .pdf, or Google Document format. Markdown usage is preferred.

Iterative Feedback

Please provide feedback to us during the development process so we can help speed up the development. We have many many years of experience and are here to help you get the project done as fast as possible.

After working Unicode Readers were finished and analysis began, Embarcadero project managers realized they were unable to test important aspects of each framework's database implementation performance. The following tests were specified to measure framework speed in database storage and retrieval *assuming the PostgreSQL database performance remained constant between applications*.

Test #1 - Database Storage

This test will measure the performance of the framework's database storage implementation. Network usage time will be eliminated by monitoring using a separate network monitoring tool and the database performance will be assumed equivalent for each application.

The test will start with a database configured with many RSS channels (20+) and an empty articles table.

Test steps:

1. Start a timer
2. Download and store all articles available from all RSS feeds in the database
3. Stop the timer and display the elapsed time

Test #2 - Database Retrieval

This test will measure the performance of the framework's database query implementation by touching each record in the articles table and saving them to a flat file with simple HTML. The database performance is assumed to be equivalent for each application and the output file operations are basic in order to reveal differences in database access implementation efficiency through completion time disparities.

The output file should be named combinedRSS.html and adhere to the following format:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Combined RSS Feeds</title>
  </head>
  <body>
    <h2><a href="channel link">Channel Name</a> - <a href="article
    link">Article Title</a></h2>
    <h3>Date</h3>
    <p>Article content</p>
    <br>
    <hr>
```

```
<h2><a href="channel link">Channel Name</a> - <a href="article
link">Article Title</a></h2>
<h3>Date</h3>
<p>Article content</p>
<br>
<hr>

<h2><a href="channel link">Channel Name</a> - <a href="article
link">Article Title</a></h2>
<h3>Date</h3>
<p>Article content</p>
<br>
<hr>
</body>
</html>
```

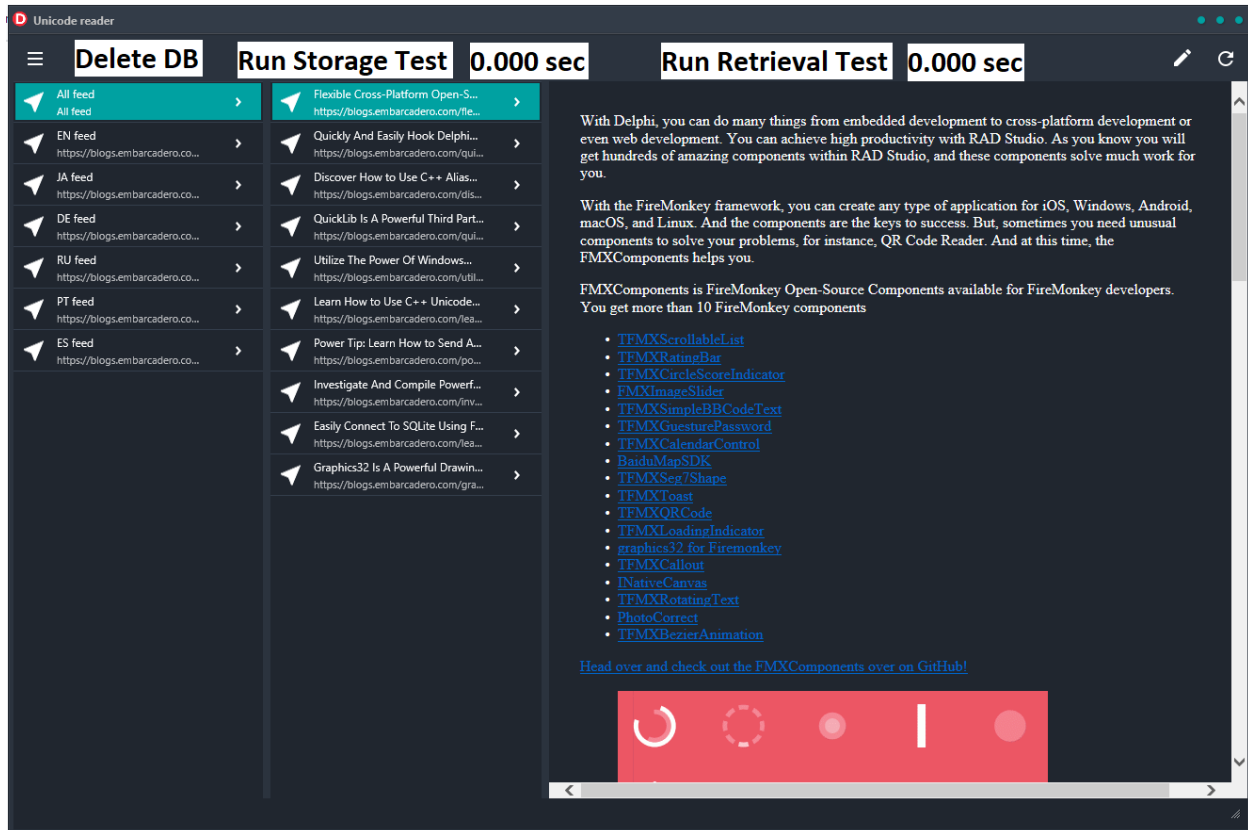
The test will start with a database configured with many RSS channels (20+) and a populated articles table from Test 1.

Test steps:

1. Start a timer
2. Access each article in the database
3. Sort articles in reverse chronological order
4. Concatenate articles and save as *combinedRSS.html* in the executable directory
5. Stop the timer and display the elapsed time

GUI Modifications

Add a button and timer for each test to the top bar of the application. If able in less than an hour, add a button to “reset” the database according to the given SQL script that drops and re-initiates tables. See the following image for a mockup of the modified Unicode Reader.



Appendix 2. Detailed Framework Analysis

Framework	Evaluation	Score	
		D	E
P1: Development Time			
Delphi	One expert Delphi developer completed the Unicode Reader in 23.3 hours using the RAD Studio IDE. Application modification with internal tests took 8.33 hours for a total development time of 31.63 hours. Five other Delphi developers gave estimates for the original application ranging from 24 to 50 hours, averaging 38.8 hours.	3	
Electron	One expert Electron developer completed the Unicode Reader in 20 hours using Angular for the RSS reader GUI and node-postgres, a collection of node.js modules, for the database interactions. Application modification with internal tests took an additional 47.8 hours - 28.6 hours to code the tests and 19.2 hours to troubleshoot issues on three platforms until acceptance criteria were met - for a total of 67.8 hours. Three other Electron estimates for the Unicode Reader ranged from 80 to 120 hours with a mean of 100 hours.		5
P2: UI Design Approach			
Delphi	Delphi's RAD Studio IDE offers a What-You-See-Is-What-You-Get (WYSIWYG) design experience with drag-and-drop components for visual GUI design. The designed GUI can be viewed using native Android/iOS/Windows/macOS styling or custom styles and can simulate application appearances within mobile devices of varying screen sizes. Components can also be resized and have their properties adjusted in the Object Inspector without touching code, allowing rapid prototyping through visual development. Delphi also offers the ability for a developer to edit the UI using a simple YAML style language definition.	5	
Electron	Electron lacks a native IDE but can be developed using text editors and command line tools, Electron doesn't include a WYSIWYG design experience or drag-and-drop components by default. The UI can be created using HTML5 and CSS styling. Unless the developer chooses an IDE like Visual Studio, Electron applications must be compiled and run to view the project's GUI.		3
P3: Developer Environment Tools			
Delphi	Delphi's IDE, RAD Studio, offers a plethora of developer tools including Code Insight (suggestions, completion, etc.), advanced debugger, code formatting, refactoring assistance, keystroke macros, and integration with common software version control systems. RAD Studio provides an Android emulator feature and can tie into an iOS simulator on a macOS machine. RAD Studio is the only IDE available for Delphi and the only method of compiling Delphi projects, however, both the code and UI definitions can be edited using standard text editors.	4	

<i>Electron</i>	Electron applications can be written in code editors such as Visual Studio, Atom, and WebStorm as well as full IDEs. All offer robust features and tools to enhance developer productivity. Electron must be compiled, run, and packaged using the command line - integration with Visual Studio Code hasn't been completed. Third party solutions may be available.		4
P4: Final Build Time			
<i>Delphi</i>	One expert Delphi developer completed the RSS reader speedrun in 1.13 hours.	5	
<i>Electron</i>	One expert Electron developer completed the RSS reader speedrun in 1 hours.		3
P5: Code Size			
<i>Delphi</i>	The Delphi FMX RSS reader required 178 lines of typed code total and 97 lines for the GUI (54%).	5	
<i>Electron</i>	The Electron RSS reader required 293 lines of typed code total and 115 lines for the GUI (39%).		3
P6: App Store Deployment			
<i>Delphi</i>	Delphi's FMX framework can deploy applications for the Microsoft Store, Apple App Store, and Google Play app store for Android. In some cases this deployment results in a platform package such as an APK or IPA which must be uploaded.	5	
<i>Electron</i>	Electron applications can be packaged for the Microsoft Store but will not be deployed there directly by default. Third party options are available. Electron apps can also be packaged for the Apple App Store but the process lacks automation help.		2
F1: License			
<i>Delphi</i>	Delphi is a proprietary software with three paid license tiers and a free Community Edition and Academic Program. The free tier allows for development as long as annual revenue does not exceed \$5,000 USD per year. The first license for full commercial use costs \$1,599 USD and the tier that fully unlocks the software suite is priced at \$5,999 USD at the time of this writing. An annual subscription is offered at one-third the initial license cost in order to receive updates and new software versions.	3	
<i>Electron</i>	Electron is a free and open-source (MIT license) framework allowing full commercial use without any licenses or fees. It is not tied to an IDE but can be developed in Visual Studio to take advantage of the IDE's tools and 3rd party ecosystem.		5
F2: Long-term Feasibility			
<i>Delphi</i>	The Delphi language has been growing, maturing, and expanding since 1995. It's multi-platform desktop and mobile framework (FireMonkey) was released in 2011 and constantly expands access to new hardware and operating systems while maintaining backward compatibility. Comprehensive documentation aids maintenance and a full support team is available for upgrade, migration, or troubleshooting help.	5	
<i>Electron</i>	Released in April 2016, Electron is actively developed and maintained by GitHub and has rapidly provided support for emerging technologies like Apple Silicon (circa Nov		3

	2020). It lacks the history and stable longevity needed to determine if Electron apps built in 2020 will survive through 2030.		
F3: Development Platforms			
<i>Delphi</i>	Delphi can only be developed within RAD Studio on Windows machines. Virtualization solutions such as VMware, Parallels, and Virtual Box with a virtual Windows machine can facilitate Delphi development on other platforms.	2	
<i>Electron</i>	Electron can be developed on Windows, macOS, and Linux.		5
F4: Testing Support			
<i>Delphi</i>	Delphi ships with the DUnitX unit testing package but lacks a native integration testing system. Numerous 3rd party unit and integration testing tools are available but may not be free.	4	
<i>Electron</i>	Electron does not install with a native unit or integration testing package. Open-source projects and libraries are available for both functions.		4
F5: Tool Extension			
<i>Delphi</i>	The RAD Studio IDE for Delphi is written in Delphi. Users can build their own extensions and tools in Delphi, eliminating the need to learn a new language and handle language boundary problems. Additionally, extensions and tools can be built in C++ via the C++Builder side of RAD Studio.	5	
<i>Electron</i>	Electron lacks a native IDE but can use plug-ins available in IDEs such as Visual Studio Code. Additional Electron tools might have to be developed in-house from scratch or integrated with a 3rd party tool such as Visual Studio Code. There are a large number of open source projects around tooling and functionality for Electron.		3
F6: Accessibility			
<i>Delphi</i>	Delphi FMX applications are not accessible-friendly in the latest release but work is being done to re-issue a free accessibility package for Windows applications. Delphi VCL applications are fully accessibility-compatible for Windows.	3	
<i>Electron</i>	Chromium supports many accessibility tools such as screen readers and magnifiers and is functional on all desktop platforms. Electron provides support for Win32, Win64, macOS, and Linux..		5
F7: IP Security			
<i>Delphi</i>	Delphi compiles to native machine code, eliminating much of the source code structure and metadata necessary for accurate decompilation and interpretation. Decompilation using a tool like <i>DeDe</i> will provide some details about the UI but only assembly code for the logic/back-end.	5	
<i>Electron</i>	Electron source code is packaged and deployed to the end-user's system. Unless a developer uses 3rd party tools to obfuscate code, the source code can be read verbatim using a simple text editor or by unpacking with a tool like <i>asar</i> .		1
X1: Target Platforms			

<i>Delphi</i>	Delphi can compile to native 32-bit or 64-bit code for Windows, macOS, Android, iOS, and Linux using the FMX framework. In the latest versions 32-bit support is being phased out for platforms that have dropped 32-bit support.	5	
<i>Electron</i>	Electron packages for cross-platform desktop use within the Chromium browser rather than compiling to native code.		3
X2: Project Variety			
<i>Delphi</i>	Delphi can be used to create applications on all levels from Windows services to Programmable Systems-on-Chip (PSOC) to enterprise applications with database, UI, and network components. Third party tools extend Delphi applications to the web.	5	
<i>Electron</i>	Electron applications mimic desktop applications by running in the Chromium browser and are typically web-centric (i.e. collaboration, messaging, etc.). Electron uses node.js for native services, utilities, and back-end applications and can integrate a wide variety of Javascript and Typescript modules.		3
X3: Scalability			
<i>Delphi</i>	Delphi applications can be separated according to a chosen design pattern. Delphi supports client, middle-tier, and back-end applications and each tier can be divided and owned by different teams.	5	
<i>Electron</i>	Electron can be developed and tested modularly for projects of any size. Electron uses node.js for middle-tier and back-end functions.		5
X4: Unicode Support			
<i>Delphi</i>	Delphi added Unicode support to the Windows-only Visual Component Library (VCL) framework in 2009 and designed the multi-platform Firemonkey(FMX) framework to support Unicode from its very inception. The UnicodeString type is the language default for strings and propagates universal language and character support throughout each component and toolkit in the IDE and runtime library. Third-party vendors offer packages of components that provide Unicode backward-compatibility for operating systems older than 2009.	5	
<i>Electron</i>	JavaScript natively supports UTF-16 for the original Unicode standard and has introduced new conventions to handle Unicode characters beyond the original set. Electron and the Chromium browser support Unicode as fully as the JavaScript language and benefit from multiple open-source projects that ease character conversions among different formats.		5
X5: Database Access			
<i>Delphi</i>	Delphi ships with multiple database libraries that connect to nearly every database type on the market. Database access, queries, and data display are smoothly integrated through components accessible in the free Community Edition and at the first commercial license tier. These components can execute pure SQL commands if desired but also provide an abstracted interface to simplify database interactions.	5	

<i>Electron</i>	Electron does not include a native database access library. Multiple open source libraries are available to harness server and server-less databases, including JavaScript implementations for nearly every database system in use.		4
X6: Hardware Access			
<i>Delphi</i>	Delphi's FMX framework includes libraries that allow interaction with a device's peripheral sensors and components regardless of platform. These libraries compile into native code. The Delphi RTL, direct memory access, and other low level features give it full access to the hardware platform, including inline assembly code on x86 desktop platforms.	5	
<i>Electron</i>	Electron can access operating system functions and hardware peripherals through node.js libraries. It's cross-platform Chromium base facilitates high level hardware access on all major desktop platforms. Electron's access to hardware is through managed code rather than native code and can only access features exposed through libraries.		3
X7: Platform Access			
<i>Delphi</i>	Delphi FMX is fully capable of accessing and using native OS APIs and features on all major desktop and mobile platforms. Delphi applications can push native OS messages and notifications and access such platform functions as storage, contacts, battery status, etc.	5	
<i>Electron</i>	Electron applications are unable to utilize operating system functions without bridging libraries developed with other tools and frameworks.		3
R1: Deployment Requirements			
<i>Delphi</i>	Delphi compiled into one executable binary file and one SQL script. The executable measured 18 MB in size for 64-bit Windows, 36 MB for macOS, and 31 MB for Linux.	5	
<i>Electron</i>	Electron compiled into 12,300 files totalling 296 MB in size for 64-bit Windows, 297 MB for macOS, and 206MB for Linux.		3
R2: Startup Time			
<i>Delphi</i>	The Delphi RSS reader averaged a startup time of 0.240 seconds from a local file system and 0.473 seconds from a network hard drive with a standard deviation of 0.132 and 1.478 seconds respectively. The slowest startup times were 1.202 seconds locally and 18.830 seconds over the network.	5	
<i>Electron</i>	The Electron RSS reader averaged a startup time of 1.371 seconds from local storage and 49.875 seconds from a network folder with a standard deviation of 0.135 and 3.789 seconds respectively. The slowest startup times were 4.181 seconds locally and 74.334 seconds networked.		3
R3: Standing Memory Usage			
<i>Delphi</i>	The Delphi RSS reader idled at 71.6 MB of memory.	3	
<i>Electron</i>	The Electron RSS reader used 38.9 MB when idling.		5

R4: Peak Memory Usage			
<i>Delphi</i>	The Delphi RSS reader demonstrated a peak memory use of 103.5 MB during RSS retrieval.	3	
<i>Electron</i>	The Electron RSS reader peaked at 94.7 MB on startup.		5
R5: Network Utilization Time			
<i>Delphi</i>	The Windows Delphi RSS reader spent 83.7% of the storage test time waiting for network responses and just 16.3% - on framework processing. The Linux application showed a much more even split with 50.4% of the storage test time attributable to network responses and 49.6% to framework processing.	5	
<i>Electron</i>	The Electron RSS reader spent 49.9% of the Windows storage test time waiting for network responses and 50.1% - on framework processing, substantially more than Delphi. Similarly to Delphi, though, the Linux test showed an even split with 50% of the storage test time attributable to network responses and 50% to framework processing.		3
R6: Database Utilization Time			
<i>Delphi</i>	Delphi retrieved and output all RSS articles in 106 milliseconds in Windows, 157 milliseconds in Linux, and 323 milliseconds in macOS.	3	
<i>Electron</i>	Electron posted retrieval test times of 159 milliseconds in Windows, 248 milliseconds in Linux, and 119 milliseconds in macOS, slightly edging out Delphi when averaged.		5

Appendix 3. Framework Decompilation Analysis

Overview

The goal of this decompilation exercise was to determine the feasibility of retrieving *both* the UI and the original code from each framework's calculator application using open-source or free tools. The frameworks assessed were Delphi FMX and Electron (with Angular).

When the Delphi FMX RSS reader was decompiled, all UI elements were successfully extracted and the logic code was presented as assembly. This exercise wasn't able to extract function and procedure structure or re-create the GUI layout from the binary file but it may be possible with more experience.

The UI elements and Javascript code of the Electron RSS reader are easily exposed using a standard text editor. The Typescript code was transpiled into Javascript and could not be recovered. Overall, Electron's packaging provided a very limited level of obfuscation.

Tools

Delphi

[DeDe](#)²⁶ - a free Delphi decompiler.

[Interactive Delphi Reconstructor](#)²⁷ - a decompiler for Delphi executables and dynamic libraries.

[MiTeC DFM Editor](#)²⁸ - a standalone editor for Delphi Form files (*.dfm) in both binary and text format.

Electron

[TextPad](#)²⁹ - a general purpose text editor for plaintext files.

²⁶ <https://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissassemblers/DeDe.shtml>

²⁷ <https://github.com/crypto2011/IDR>

²⁸ <https://www.mitec.cz/dfm.html>

²⁹ <https://www.textpad.com/home>

Decompiler Results

Delphi VCL

DeDe 3.50.02 (c) 1999-2002 by DaFixer				
File Dumpers Tools Options About				
C:\Users\Adam\Documents\Embarcadero\E... Process UnicodeReader				
Classes Info Units Info Forms Procedures Project Exports				
Class Name	Unit Name	SelfPtr	DFM Offset	Version: unknown version
TList1.Pack[0](\$BInfcFMX...		0084D2A8	00000000	Unit List (from PACKAGEINFO)
TList1.Pack[0](\$BInfcFMX...		0084CC4C	00000000	Data.Bind.Components
TList1.Pack[0](\$BInfcFMX...		0084E66C	00000000	Data.Bind.Consts
TList1.Pack[0](\$BInfcFMX...		008496CC	00000000	Data.Bind.DBScope
TList1.Pack[0](\$BInfcFMX...		00562AD8	00000000	Data.Bind.Editors
TList1.Pack[0](\$BInfcFMX...		0055EA10	00000000	Data.Bind.Ext
TList1.Pack[0](\$BInfcFMX...		0055F118	00000000	Data.Bind.Links
TList1.Pack[0](\$BInfcFMX...		0055F118	00000000	Data.Bind.ObjectScope
TList1.Pack[0](\$BInfcFMX...		00966D38	00000000	Data.Bind.ObserverLinks
TList1.Pack[0](\$BInfcFMX...		008EB938	00000000	Data.DB
TList1.Pack[0](\$BInfcFMX...		00967970	00000000	Data.DB.CommonTypes
TList1.Pack[0](\$BInfcFMX...		0096A760	00000000	Data.DB.Consts
TList1.Pack[0](\$BInfcFMX...		00C6980C	00000000	Data.FmtBcd
TList1.Pack[0](\$BInfcFMX...		005D8EE4	00000000	Data.SqlFmtBcd
TList1.Pack[0](\$BInfcFMX...		005D7158	00000000	FireDAC.Comp.BatchMove
TList1.Pack[0](\$BInfcFMX...		0096ADD8	00000000	FireDAC.Comp.BatchMove.DataSet
TList1.Pack[0](\$BInfcFMX...		006997F4	00000000	FireDAC.Comp.BatchMove.SQL
TList1.Pack[0](\$BInfcFMX...		0092BAAB	00000000	FireDAC.Comp.BatchMove.Text
TList1.Pack[0](\$BInfcFMX...		006102B0	00000000	FireDAC.Comp.Client
TList1.Pack[0](\$BInfcFMX...		00610994	00000000	FireDAC.Comp.DataSet
TList1.Pack[0](\$BInfcFMX...		006164DC	00000000	FireDAC.Comp.Script
TList1.Pack[0](\$BInfcFMX...		005DD4C	00000000	FireDAC.Comp.ScriptCommands
TList1.Pack[0](\$BInfcFMX...		00617888	00000000	FireDAC.Comp.UI
TList1.Pack[0](\$BInfcFMX...		008C25BC	00000000	FireDAC.DApt
TList1.Pack[0](\$BInfcFMX...		007182DC	00000000	FireDAC.DApt.Column
TList1.Pack[0](\$BInfcFMX...		005D7158	00000000	FireDAC.DApt.Init
TList1.Pack[0](\$BInfcFMX...		00617888	00000000	FireDAC.DApt.Params
TList1.Pack[0](\$BInfcFMX...		008C25BC	00000000	FireDAC.DApt.Wait
TList1.Pack[0](\$BInfcFMX...		007182DC	00000000	FireDAC.Phys
TList1.Pack[0](\$BInfcFMX...		0053F7B8	00000000	FireDAC.Phys.Intf
TList1.Pack[0](\$BInfcFMX...		005328C0	00000000	FireDAC.Phys.Meta
TList1.Pack[0](\$BInfcFMX...		00ACBEE8	00000000	FireDAC.Phys.PG
TList1.Pack[0](\$BInfcFMX...		00ACB36C	00000000	FireDAC.Phys.PGCLI
TList1.Pack[0](\$BInfcFMX...		00A75758	00000000	FireDAC.Phys.PGDel
TList1.Pack[0](\$BInfcFMX...		00A66AEC	00000000	FireDAC.Phys.PGMeta
TList1.Pack[0](\$BInfcFMX...		00A8A0CC	00000000	FireDAC.Phys.PGWrapper
TList1.Pack[0](\$BInfcFMX...		00A88D1C	00000000	FireDAC.Phys.SQLGenerator
TList1.Pack[0](\$BInfcFMX...		00A88628	00000000	FireDAC.Phys.SQLPreprocessor
TList1.Pack[0](\$BInfcFMX...		00B24544	00000000	FireDAC.Stan.Async
TList1.Pack[0](\$BInfcFMX...		00B0427C	00000000	FireDAC.Stan.Consts
TList1.Pack[0](\$BInfcFMX...		00B002B4	00000000	FireDAC.Stan.Def
TList1.Pack[0](\$BInfcFMX...		00B0FFC8	00000000	FireDAC.Stan.Error
TList1.Pack[0](\$BInfcFMX...		00B09C58	00000000	FireDAC.Stan.Expr
TList1.Pack[0](\$BInfcFMX...		00A6E768	00000000	FireDAC.Stan.Factory
TList1.Pack[0](\$BInfcFMX...		006CD690	00000000	FireDAC.Stan.Intf
TList1.Pack[0](\$BInfcFMX...		005D09CC	00000000	FireDAC.Stan.Option
TList1.Pack[0](\$BInfcFMX...		008EA150	00000000	FireDAC.Stan.Param
TList1.Pack[0](\$BInfcFMX...		00D1A8B0	00000000	FireDAC.Stan.Pool
TList1.Pack[0](\$BInfcFMX...		005D09CC	00000000	FireDAC.Stan.RectStr
TList1.Pack[0](\$BInfcFMX...		00D1A8B0	00000000	FireDAC.Stan.SQLTimeInt
TList1.Pack[0](\$BInfcFMX...		005D09CC	00000000	FireDAC.Stan.UI
TList1.Pack[0](\$BInfcFMX...		00D1A8B0	00000000	FireDAC.UI
TList1.Pack[0](\$BInfcFMX...		005D09CC	00000000	FireDAC.UI.Intf
TList1.Pack[0](\$BInfcFMX...		00D1A8B0	00000000	FMX.AcceleratorKey
TList1.Pack[0](\$BInfcFMX...		005D09CC	00000000	FMX.AcceleratorKey.Win
TList1.Pack[0](\$BInfcFMX...		005D09CC	00000000	FMX.AchtrList
TList1.Pack[0](\$BInfcFMX...		005D09CC	00000000	FMX.AchtrList

Table 1 - DeDe Decompile of Delphi FMX

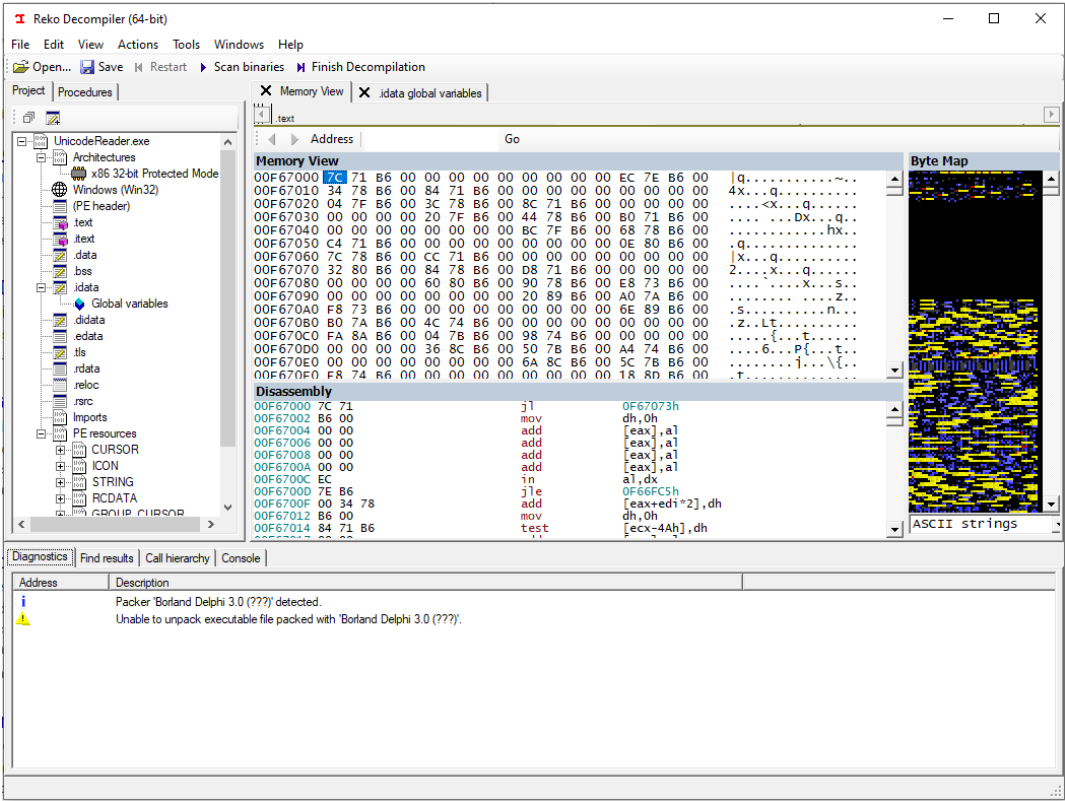


Table 2 - Reko Binary Decompilation of Delphi FMX

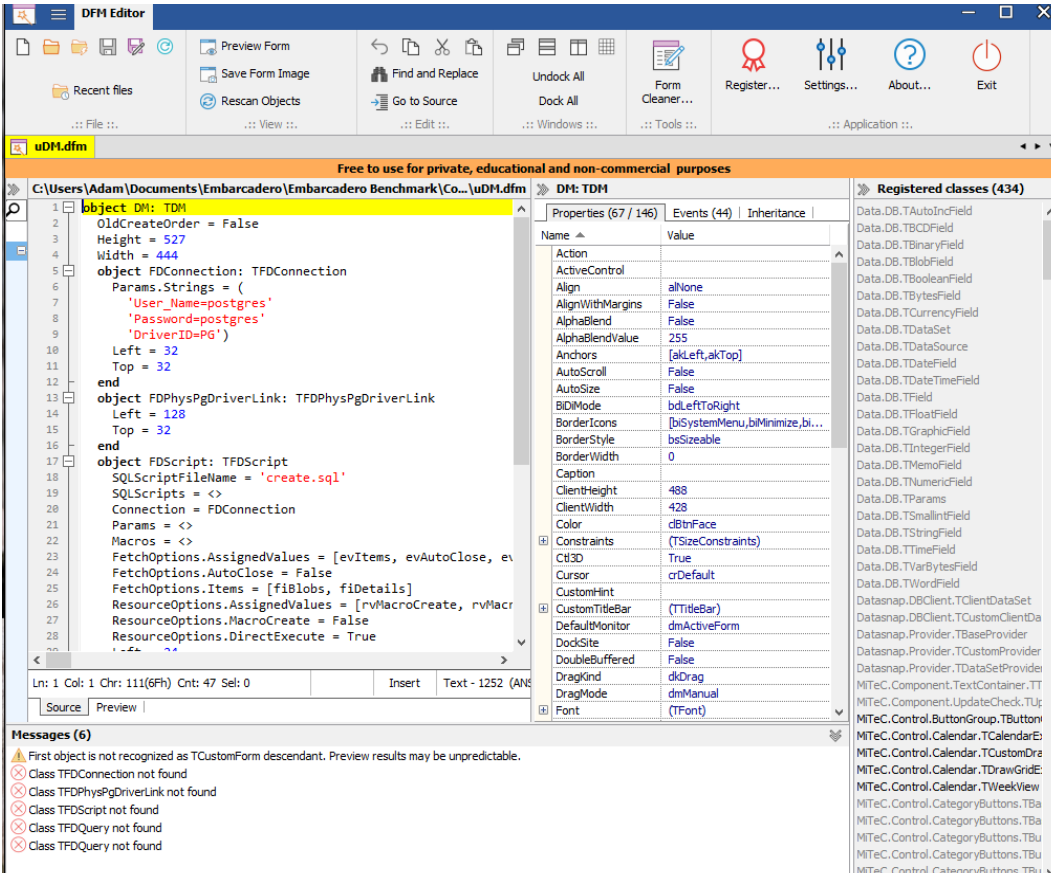


Table 2 - DFM Editor GUI Code View of Delphi FMX

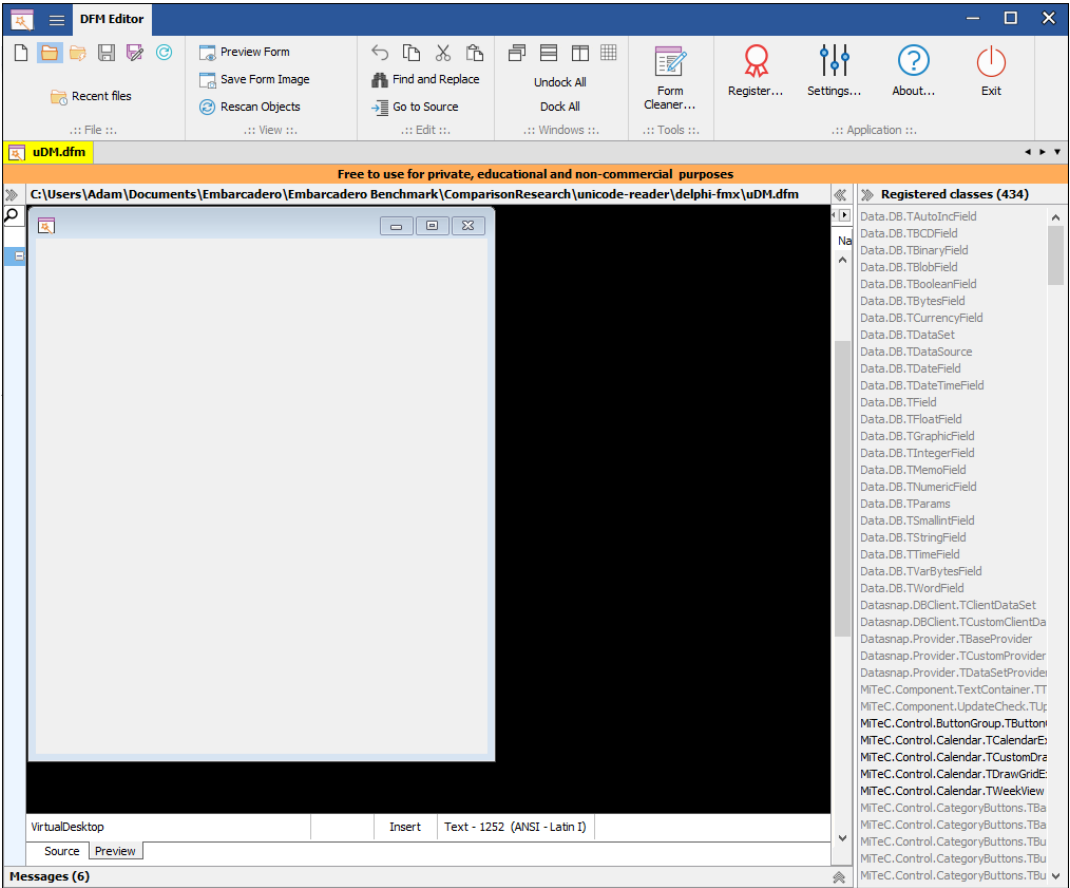
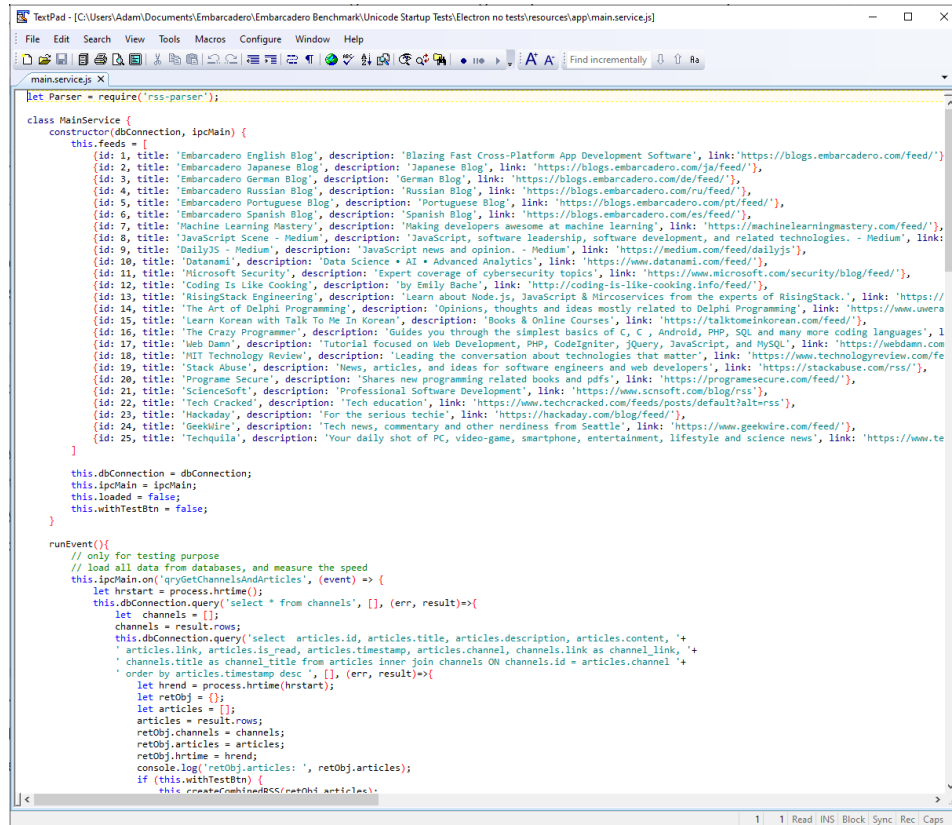


Table 3 - DFM Editor GUI Design View of Delphi FMX

Electron



```
let Parser = require('rss-parser');

class MainService {
  constructor(dbConnection, ipcMain) {
    this.feeds = [
      (id: 1, title: 'Embarcadero English Blog', description: 'Blazing Fast Cross-Platform App Development Software', link: 'https://blogs.embarcadero.com/feed/'),
      (id: 2, title: 'Embarcadero Japanese Blog', description: 'Japanese Blog', link: 'https://blogs.embarcadero.com/ja/feed/'),
      (id: 3, title: 'Embarcadero German Blog', description: 'German Blog', link: 'https://blogs.embarcadero.com/de/feed/'),
      (id: 4, title: 'Embarcadero Russian Blog', description: 'Russian Blog', link: 'https://blogs.embarcadero.com/ru/feed/'),
      (id: 5, title: 'Embarcadero Portuguese Blog', description: 'Portuguese Blog', link: 'https://blogs.embarcadero.com/pt/feed/'),
      (id: 6, title: 'Embarcadero Spanish Blog', description: 'Spanish Blog', link: 'https://blogs.embarcadero.com/es/feed/'),
      (id: 7, title: 'Machine Learning Mastery', description: 'Making developers awesome at machine learning', link: 'https://machinelearningmastery.com/feed/'),
      (id: 8, title: 'JavaScript Scene - Medium', description: 'JavaScript, software leadership, software development, and related technologies. - Medium', link: 'https://medium.com/feed/dailyjs'),
      (id: 9, title: 'DailyJS - Medium', description: 'JavaScript news and opinion. - Medium', link: 'https://medium.com/feed/dailyjs'),
      (id: 10, title: 'DataNami', description: 'Data Science • AI • Advanced Analytics', link: 'https://www.datanami.com/feed/'),
      (id: 11, title: 'Microsoft Security', description: 'Expert coverage of cybersecurity topics', link: 'https://www.microsoft.com/security/blog/feed/'),
      (id: 12, title: 'Coding Is Like Cooking', description: 'by Emily Bache', link: 'http://coding-is-like-cooking.info/feed/'),
      (id: 13, title: 'RisingStack Engineering', description: 'Learn about Node.js, JavaScript & Microservices from the experts of RisingStack.', link: 'https://risingstack.com/feed/'),
      (id: 14, title: 'The Art of Delphi Programming', description: 'Opinions, thoughts and ideas mostly related to Delphi Programming', link: 'https://www.uwera.com/feed/'),
      (id: 15, title: 'Learn Korean with Talk To Me In Korean', description: 'Books & Online Courses', link: 'https://talktomeinkorean.com/feed/'),
      (id: 16, title: 'The Crazy Programmer', description: 'Guides you through the simplest basics of C, C++, Android, PHP, SQL and many more coding languages', link: 'https://www.crazyprogrammer.com/feed/'),
      (id: 17, title: 'Web Damm', description: 'Tutorial focused on web development, PHP, CodeIgniter, jQuery, Javascript, and MySQL', link: 'https://webdamm.com/feed/'),
      (id: 18, title: 'MIT Technology Review', description: 'Leading the conversation about technologies that matter', link: 'https://www.technologyreview.com/feed/'),
      (id: 19, title: 'Stack Abuse', description: 'News, articles, and ideas for software engineers and web developers', link: 'https://stackabuse.com/feed/'),
      (id: 20, title: 'Programme Secure', description: 'Shares new programming related books and pdfs', link: 'https://programmesecure.com/feed/'),
      (id: 21, title: 'ScienceSoft', description: 'Professional Software Development', link: 'https://www.scsoft.com/blog/feed/'),
      (id: 22, title: 'Tech Cracked', description: 'Tech education', link: 'https://www.techcracked.com/feeds/posts/default?alt=rss'),
      (id: 23, title: 'Hackaday', description: 'For the serious techie', link: 'https://hackaday.com/blog/feed/'),
      (id: 24, title: 'Geekwire', description: 'Tech news, commentary and other nerdiness from Seattle', link: 'https://www.geekwire.com/feed/'),
      (id: 25, title: 'Techquila', description: 'Your daily shot of PC, video-game, smartphone, entertainment, lifestyle and science news', link: 'https://www.techquila.com/feed/'),
    ];

    this.dbConnection = dbConnection;
    this.ipcMain = ipcMain;
    this.loaded = false;
    this.withTestBtn = false;
  }

  runEvent() {
    // only for testing purpose
    // load all data from databases, and measure the speed
    this.ipcMain.on('qryGetChannelsAndArticles', (event) => {
      let hrstart = process.hrtime();
      this.dbConnection.query('select * from channels', [], (err, result) => {
        let channels = [];
        channels = result.rows;
        this.dbConnection.query('select articles.id, articles.title, articles.description, articles.content, '+
          ' articles.link, articles.is_read, articles.timestamp, articles.channel, channels.link as channel_link, '+
          ' channels.title as channel_title from articles inner join channels ON channels.id = articles.channel '+
          ' order by articles.timestamp desc', [], (err, result) => {
          let hrend = process.hrtime(hrstart);
          let retObj = {};
          articles = result.rows;
          retObj.channels = channels;
          retObj.articles = articles;
          retObj.hrtime = hrend;
          console.log('retObj.articles: ', retObj.articles);
          if (this.withTestBtn) {
            this.createChannelDescFromArticles();
          }
        });
      });
    });
  }
}
```

Table 12- Textpad Displaying Electron Logic Code



Appendix 4. Contributors & Sponsor

This paper was made possible through the efforts of multiple contributors. All the sources are available on GitHub in the [ComparisonResearch](https://github.com/Embarcadero/ComparisonResearch/tree/main/unicode-reader) project github.com/Embarcadero/ComparisonResearch/tree/main/unicode-reader

Embarcadero Technologies, Inc.

Marco Cantù	RAD Studio Senior Product Manager
Jim McKeeth	Chief Developer Advocate & Engineer
David Millington	RAD Studio Senior Product Manager
Stephen Ball	delphiaball.co.uk
Hagop Panosian	Product Marketing Manager

Embarcadero Most Valuable Professionals (MVPs)

Olaf Monien	developer-experts.net
-------------	--

Independent Contractors

Adam Leone	repo-me-this.com
Eli M.	fmxexpress.com
Dhiraj Sakariya	upwork.com/freelancers/~01139eb7cc53906988
Heru Susanto	upwork.com/freelancers/~0195227b473e36e942
Victor Vkulenko	upwork.com/freelancers/~01393e5253b24c66e9

About Embarcadero Technologies

Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster, and run them better regardless of platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance, and accelerate innovation. Founded in 1993, Embarcadero is headquartered in Austin, Texas with offices located around the world.



About Delphi

Delphi is Embarcadero's flagship development tool supporting native application and server development for Windows, macOS, Linux, Android, and iOS. It includes a variety of libraries including a robust framework for database applications, REST services, and visual application development. It is available in multiple editions, including a free Community Edition, an Academic Edition, and Professional, Enterprise, and Architect Editions.

www.embarcadero.com/products/delphi



Download a free trial www.embarcadero.com/products/delphi/start-for-free