

EasyPDF™

The Acrobat

ENCYCLOPEDIA

THREE DECADES OF PRACTICAL EXPERIENCE
DESIGNING SOPHISTICATED INTERACTIVE
PDF APPLICATIONS WITH ADOBE ACROBAT

Architectures • Real-World Solutions • Best Practices

EasyPDF™ Interactive Application

Please complete all required fields.

1. Customer Information

Full Name:

Email Address:

Date:

2. Application Data

Choose your options:

☐ Option 1

☐ Option 2

☐ Option 3

3. Notes

Save & Calculate



```
// Example: Store application data
function saveRecord(data) {
  var store = this.getField("dsData");
  var obj = JSON.parse(store.value || "{}");
  obj[data.id] = data;
  store.value = JSON.stringify(obj);
  this.calculateTotals();
  this.refreshView();
}
```



Application Architecture
Design maintainable,
self-contained PDF
applications



Event Engineering
Master Acrobat's
event model and
execution order



**JSON & Persistent
Storage**
Store and manage
structured data
inside the PDF



Design Patterns
Proven patterns for
reusable, scalable,
and reliable code



Maintainability
Engineer for clarity,
performance, and
long-term success

*The Definitive Guide to Designing Sophisticated,
Self-Contained Interactive PDF Applications
Using Adobe Acrobat*

EASYPDF™ FORMS
DOCUMENT AUTOMATION MADE EASY

by Marty Karr



EasyPDF™ Acrobat Encyclopedia

Foreword

When I first began working with Adobe Acrobat more than three decades ago, I viewed it much as most people did—as an exceptional tool for creating electronic documents and interactive forms.

Over time, however, something unexpected happened.

Each new project presented another opportunity to solve a practical problem. One solution led to another. Simple forms evolved into increasingly sophisticated applications. New ideas emerged through experimentation, experience, failure, and refinement. Looking back, I now realize that the greatest lessons I learned were seldom about JavaScript syntax or individual Acrobat features. They were about software design, application architecture, and discovering better ways to organize information and solve real-world problems.

Eventually, a realization changed the way I viewed interactive PDF development forever.

The breakthrough was not discovering JavaScript objects.

It was recognizing that a JavaScript object could fulfill the same organizational role as a traditional database record.

Once that connection became clear, everything else began to fall into place.

Interactive PDF forms were no longer merely electronic versions of paper documents.

They had quietly become something much more.

That single insight ultimately transformed the way I approached every application I designed thereafter.

The pages that follow are not intended to be a comprehensive reference for every Acrobat JavaScript command, nor are they intended to replace Adobe's own documentation.

Instead, this Encyclopedia documents the engineering principles, architectural concepts, design patterns, debugging techniques, and practical lessons learned while developing commercial interactive PDF applications over more than thirty years.

Some of the techniques described here evolved gradually over decades. Others emerged only after repeated experimentation and countless revisions. Many were discovered only because practical experience forced me to question assumptions I had accepted for years.

If there is one central message throughout this work, it is this:

Technology occasionally advances through entirely new inventions.

Just as often, however, progress occurs because someone begins looking at familiar technology from a different perspective.

That is my hope for this Encyclopedia.

EasyPDF™ Acrobat Encyclopedia

Whether you are creating your first interactive PDF application or have spent years working with Acrobat JavaScript, I hope these pages encourage you to think beyond individual scripts and begin thinking about application architecture, maintainability, reusable design, and solving problems that extend far beyond electronic forms.

If, after reading this Encyclopedia, you find yourself looking at Acrobat differently than you did before opening the first page, then it will have accomplished exactly what I set out to achieve.

Marty Karr

Author

EasyPDF™ Acrobat Encyclopedia

Declaration of Purpose

To demonstrate, through decades of practical experience, that interactive PDF applications created with Adobe Acrobat remain a powerful, capable, yet often misunderstood and underrated desktop application platform, whose most powerful capabilities are revealed when Acrobat's features are combined to create sophisticated, self-contained desktop applications.

EasyPDF™
Acrobat Encyclopedia

Preface

Compiled and maintained by

Marty Karr

with the assistance of ChatGPT (OpenAI)

For more than three decades, I have designed and developed interactive PDF applications using Adobe Acrobat. During that time, what initially began as a pursuit of creating more capable electronic forms gradually evolved into something far more significant. Through years of experimentation, problem solving, and practical application development, I came to recognize that Acrobat possesses capabilities extending well beyond those traditionally associated with interactive PDF forms.

The EasyPDF™ Acrobat Encyclopedia is the result of that journey.

Unlike traditional programming references that primarily document JavaScript syntax, objects, properties, methods, and events, this encyclopedia focuses upon the engineering principles, architectural patterns, development methodologies, and practical lessons learned while designing, testing, refining, and maintaining the EasyPDF™ family of interactive PDF applications. Rather than documenting Acrobat's individual features in isolation, it demonstrates how those features combine to create sophisticated, self-contained desktop applications capable of solving real-world problems.

Throughout these chapters, readers will discover that successful Acrobat development depends upon considerably more than writing JavaScript. Reliable applications are built upon thoughtful architecture, sound engineering practices, reusable helper functions, structured data organization, disciplined event management, maintainability, and a thorough understanding of how Acrobat actually behaves as applications grow in size and complexity.

Perhaps most importantly, this encyclopedia introduces a fundamentally different way of thinking about interactive PDF development. By combining JavaScript objects, JSON serialization, hidden form fields, reusable helper functions, and Acrobat's event model, a single PDF document can become a remarkably capable desktop application requiring no external database, additional software, or Internet connection. This architectural philosophy forms the foundation underlying every EasyPDF™ application presented throughout this work.

The accompanying **Declaration of Purpose** explains why this encyclopedia was written. The chapters that follow demonstrate how those principles have been applied through more than three decades of practical development experience to create dependable, maintainable, and professional interactive PDF applications.

This free abbreviated Preview Edition of the EasyPDF™ Acrobat Encyclopedia contains selected material from the completed work, providing readers an opportunity to explore the architectural concepts, engineering philosophies, and development techniques that define the EasyPDF™ approach to Acrobat application development. The complete Encyclopedia is available in both digital PDF and printed editions.

EasyPDF™ Acrobat Encyclopedia

It is my sincere hope that this encyclopedia encourages developers not only to write better Acrobat JavaScript, but also to reconsider what interactive PDF applications created with Adobe Acrobat are truly capable of becoming.

Copyright

Copyright © 2026 Marty Karr. All rights reserved.

This Preview Edition may be freely distributed in its original, unmodified PDF format. No portion of this publication may be reproduced, modified, translated, incorporated into another publication, or sold without the prior written permission of the copyright holder.

EasyPDF™
Acrobat Encyclopedia
Table of Contents

Title

Foreword

Declaration of Purpose

Preface

Section 1 - Acrobat UI Techniques (Refer to Chapter 7.3.2)

Section 2 - Drop-Down Lists

2.1 Building Dynamic Drop-Down Lists

2.2 updateNames() Design Pattern

2.3 Navigating Drop-Down Lists Using Arrow Buttons

2.4 Alphabetical vs. Sorting by Date

2.5 Eliminating Blank Drop-Down Entries

2.6 Record Index Indicators

2.7 Updating Drop-Down Lists After Add/Delete/Restore

Section 3 – Data Architecture and Storage

3.1 The JavaScript Object: The Foundation of the Mini-Database

3.2 Why JSON Was Chosen

3.3 Hidden Data Fields

3.4 Nested JSON Structures

3.5 Restore Deleted Records

Section 4 – Calculations

4.1 Why I Stopped Relying on Acrobat Calculated Fields

4.2 Currency Formatting

Section 5 - Trial Version Techniques

5.1 Designing Trial Versions

5.2 Limiting Records Without Breaking Features

5.3 Trial Version Messaging

5.4 Maintaining One Code Base

Section 6 - Acrobat Quirks

6.1 Hidden vs. Display

EasyPDF™
Acrobat Encyclopedia
Table of Contents

6.2 Commit Immediately

6.3 Beyond Combo Boxes

6.3.1 Why Combo Boxes Became My First Choice

6.3.2 Growing Pains

6.3.3 The Straw That Broke the Camel's Back

6.3.4 Discovering Popup Menus

6.3.5 Hidden List Boxes Change Everything

6.3.6 Lessons Learned

6.4 Acrobat Reader vs Acrobat Pro

Section 7 Lessons Learned

7.1 Things Never to Do Again

7.1.1 Selecting the Most Reliable Acrobat Property

7.1.2 Centralizing updateNavigationBtns()

7.1.3 Separating Trial Limits

7.1.4 Excluding Counters from Drop-Down Lists

7.1.5 The Hidden Cost of Reusing Existing PDF Forms

7.1.5.1 Why We Copy Existing PDF Forms

7.1.5.2 Every Copied PDF Has a History

7.1.5.3 Understanding Application Baggage

7.1.5.4 Hidden Dependencies

7.1.5.5 The Biggest Revelation of All

7.1.5.6 Knowing What to Keep

7.1.5.7 A New Development Philosophy

7.2 Acrobat Event Model

7.2.1 Event Sequence and Execution Order

7.2.2 Validate, Format, and Calculate Events

7.2.2.1 Validate Event

7.2.2.2 Format Event

7.2.2.3 Calculate Event

**EasyPDF™
Acrobat Encyclopedia
Table of Contents**

7.2.2.4 Blur Event

7.2.3 Deferring Code Execution with setTimeout()

7.2.4 Refreshing the User Interface

7.2.5 Common Event Mistakes

7.3 Acrobat Application Design Patterns

7.3.1 Interactive PDF Forms as Mini-Databases

7.3.2 Separating the User Interface from Business Logic

7.3.3 Designing for Maintainability

7.3.4 Using Hidden Fields for Persistent Storage

7.3.5 Building Self-Contained Applications

Appendix

A.1 Future Publication Notes

A.2 Using the HTML5 download Attribute for PDF Downloads

A.3 Creating a Professional PDF Title Page Illustration Cover

A.4 Local Preview Delays Caused by Matomo Analytics Tracking Script

A.5 Why I Switched From Button Fields to the Document Link Tool for my TOC

A.6 Acrobat Automatically Creates Unexpected Bookmarks

A.7 Adding a Clickable Website Link to an Acrobat About Dialog

A.8 Check boxes vs. Radio Buttons: Choosing the Right Field Type

3.1 The JavaScript Object: The Foundation of the Mini-Database

Problem

As EasyPDF™ forms evolved beyond conventional interactive forms, simply storing and retrieving values from individual form fields was no longer sufficient.

Applications such as customer managers, digital rolodexes, medication managers, password managers, and invoicing systems needed to maintain collections of related records. Each record could contain numerous values, and the application needed a practical way to add, retrieve, update, delete, sort, and restore those records.

A conventional solution would have been to connect the PDF to an external database. However, for relatively modest record-management applications, the additional infrastructure of an external database was unnecessary.

What was needed was a structure within JavaScript itself that could organize and manage the application's records.

Investigation

A JavaScript object provides that structure.

Rather than treating every form field as an isolated piece of stored information, an object allows related values to be grouped together as properties of a record. Multiple records can then be maintained within a larger application data structure.

Conceptually, a customer record might contain:

```
{  
  customerID: 1001,  
  name: "ABC Company",  
  phone: "555-1234",  
  email: "contact@example.com"  
}
```

The application can create additional records using the same structure and use Acrobat JavaScript to locate and manipulate individual records as required.

This changes the role of the PDF form fields. The visible fields primarily become the user interface through which information is entered, displayed, and edited, while the JavaScript object becomes the working structure used to manage the records themselves.

Solution

EasyPDF™ forms use JavaScript objects as the foundation of their internal record-management architecture.

Records can be created as objects containing properties and values. Those records can then be maintained within a larger object or collection that JavaScript can search, modify, sort, delete,

EasyPDF™ Acrobat Encyclopedia

restore, and display through the PDF interface.

Instead of requiring a database query to retrieve a particular record, JavaScript can locate the appropriate object and populate the corresponding form fields.

When information is changed, JavaScript can update the appropriate record within the object.

When a new record is created, it can be added to the existing collection.

When a record is deleted, the application can remove or relocate that record according to the application's design.

The JavaScript object therefore performs much of the organizational work that might otherwise have required an external database for applications of this scale.

Why It Works

JavaScript objects are well suited for representing structured information because properties can describe the individual pieces of information belonging to a record.

The application logic can work directly with those objects rather than continually treating individual PDF fields as the application's underlying data structure.

This creates an important separation between the user interface and the stored record structure.

PDF form fields provide the interface.

Acrobat JavaScript provides the application logic.

JavaScript objects organize the working records.

That architecture allows the PDF to behave less like a collection of independent form fields and more like a small record-management application.

There is, however, one important limitation: a JavaScript object exists in memory while the application is running. By itself, the object does not provide the persistent storage needed to preserve those records after the PDF is saved, closed, and reopened.

Solving that problem required another piece of the architecture: JSON.

Design Principle

Separate the application's interface from its record structure.

Use PDF form fields to enter and display information, while using JavaScript objects to organize and manage the underlying records.

For applications with relatively modest record requirements, this approach can provide the organizational capabilities needed without introducing an external database where one is unnecessary.

EasyPDF™ Acrobat Encyclopedia

Lessons Learned

- A JavaScript object can serve as the working record structure for a self-contained PDF application.
- Related values can be grouped into individual records rather than managed as unrelated form-field values.
- Multiple records can be created, retrieved, updated, sorted, deleted, and restored through Acrobat JavaScript.
- PDF form fields can function primarily as the user interface while JavaScript objects manage the underlying information.
- Separating the interface from the record structure makes more sophisticated PDF applications practical.
- The JavaScript object provides the structure, but an additional mechanism is required to preserve that structure between document sessions.

Historical Note

The JavaScript object became an important part of the evolution of EasyPDF™ forms from conventional interactive forms into self-contained record-management applications.

The concept addressed one of the fundamental problems encountered during that evolution: how to organize collections of related information without depending upon an external database.

The object provided the working structure needed to create and manage records, but it did not by itself solve the problem of persistence. The information still needed to survive when the PDF was saved, closed, and later reopened.

In 2020, Thom Parker brought JSON to my attention. That proved to be the final piece of the puzzle.

JSON made it possible to convert the JavaScript object into text that could be stored within the PDF and later reconstruct that text as a usable JavaScript object. The combination of the JavaScript object, JSON serialization, Acrobat JavaScript, and hidden PDF fields ultimately provided the architecture required for the self-contained mini-database approach used in later EasyPDF™ applications.

The JavaScript object therefore provided the foundation, while JSON provided the means to preserve that foundation between document sessions.

EasyPDF™ Engineering Principle

When an interactive PDF must manage multiple related records, use JavaScript objects to provide the working data structure, then use JSON and hidden PDF fields to preserve that structure within the document.

EasyPDF™ Acrobat Encyclopedia

3.2 Why JSON Was Chosen

Problem

As EasyPDF™ forms became more sophisticated, individual Acrobat form fields were no longer sufficient for managing collections of related information such as customers, contacts, medications, invoices, passwords, and other stored records.

A method was needed to organize multiple records and their associated values within the PDF itself while allowing JavaScript to add, retrieve, update, delete, sort, and restore those records without requiring an external database.

Investigation

Acrobat JavaScript objects provide a natural way to organize related information while the form is running. A single object can contain multiple records, with each record containing its own collection of properties and values.

The problem is persistence. JavaScript objects exist in memory while the document is being used, but the information must also survive when the PDF is saved, closed, and reopened.

JSON provides the connection between these two requirements. JavaScript objects can be converted into JSON text for storage and later reconstructed from that text when the data is needed again.

Solution

EasyPDF™ forms use JavaScript objects to organize application data and JSON to serialize those objects for storage within the PDF.

A JavaScript object can be converted to JSON using:

`JSON.stringify()`

The resulting text can then be stored in a designated PDF form field.

When the stored information is needed again, the JSON text can be converted back into a JavaScript object using:

`JSON.parse()`

This allows structured application data to remain embedded within the PDF rather than requiring a separate database, external data file, or Internet connection.

Why It Works

JSON preserves structured information in a text format that JavaScript can easily create and reconstruct.

Instead of attempting to treat every visible form field as an independent storage location, related

EasyPDF™ Acrobat Encyclopedia

values can be grouped into records within an object. Those records can then be serialized together, stored within the PDF, and reconstructed whenever the application needs them.

The PDF therefore becomes capable of maintaining its own structured data while the JavaScript provides the logic required to manage that data.

Design Principle

Use structured data for structured information.

When an interactive PDF application must manage multiple related records, organize the information as JavaScript objects and use JSON serialization to preserve those objects rather than scattering the application's stored data across unrelated visible fields.

Lessons Learned

- JSON provides a practical bridge between JavaScript objects and persistent PDF storage.
- `JSON.stringify()` converts structured JavaScript data into text that can be stored within the PDF.
- `JSON.parse()` reconstructs the stored text into usable JavaScript objects.
- Related records can be maintained together without requiring an external database.
- The resulting application can remain self-contained within the PDF.

Historical Note

The use of JSON became increasingly important as EasyPDF™ forms evolved from conventional interactive forms into record-oriented PDF applications.

Applications such as the EasyPDF™ Digital Rolodex, Password Manager, Medication Manager, and Smart Invoice required more than simply retaining values in individual visible fields. They needed to maintain collections of records that could be selected, edited, sorted, deleted, restored, and displayed through the user interface.

JSON provided a practical means of preserving those structured records within the PDF itself while allowing Acrobat JavaScript to reconstruct and manipulate them as ordinary JavaScript objects.

This approach became one of the fundamental architectural building blocks used throughout later EasyPDF™ form development.

EasyPDF™ Engineering Principle

When an interactive PDF must maintain structured records, use JavaScript objects to organize the data and JSON to preserve that structure within the document.

EasyPDF™ Acrobat Encyclopedia

3.3 Hidden Data Fields

Problem

JSON provides a practical way to convert structured JavaScript objects into text, but the resulting JSON still requires a location where it can be stored permanently within the PDF.

The storage location also needs to remain out of the way of the user interface. Displaying raw JSON inside a visible form field would serve no useful purpose to the user and could allow stored application data to be accidentally modified or deleted.

A method was therefore needed to provide persistent internal storage while keeping the stored data separate from the visible form interface.

Investigation

An Acrobat form field does not have to be visible to the user in order to store a value.

A designated field can therefore be used strictly as an internal data container. JavaScript can write serialized JSON to the field and retrieve it later even though the user never directly interacts with the field.

This makes a hidden field particularly useful for storing structured application data because the information remains part of the PDF and is saved along with the document.

Solution

EasyPDF™ forms use designated hidden fields as internal data stores for serialized JSON.

For example, a field such as:

`dsContact`

can contain the JSON representation of all records used by a contact-management application.

Other applications can use separate data-store fields appropriate to their purpose, such as:

`dsPatient`, `dsMed`, `dsCus`, etc.

The application converts its JavaScript object into JSON using `JSON.stringify()` and assigns the resulting text to the designated data-store field.

When the information is needed again, JavaScript retrieves the stored field value and reconstructs the object using `JSON.parse()`.

The hidden field therefore acts as the persistent storage container while the JavaScript object remains the working representation of the data.

Why It Works

Acrobat saves form-field values as part of the PDF. A field used exclusively for internal storage

EasyPDF™ Acrobat Encyclopedia

can therefore preserve serialized JSON between document sessions without requiring an external file or database.

When the document is reopened, the stored JSON remains available in the hidden data field. JavaScript can read that value, reconstruct the corresponding object, and continue working with the previously stored records.

The user does not need to see or interact with the data-store field because the visible form fields provide the interface for viewing and modifying the information.

Design Principle

Keep internal application storage separate from the visible user interface.

Visible fields should present information the user needs to view or edit. Hidden data fields should preserve the structured information required internally by the application.

Lessons Learned

- Hidden fields provide a practical location for storing serialized JSON within a PDF.
- The stored data remains part of the document when the PDF is saved and reopened.
- Raw JSON does not need to be exposed through the visible user interface.
- Different types of application data can be maintained in separate designated data-store fields.
- Hidden data fields allow an interactive PDF to maintain persistent structured information without requiring an external database.

Historical Note

Hidden data fields became increasingly important as EasyPDF™ forms evolved into applications capable of maintaining multiple records.

The EasyPDF™ Digital Rolodex used a designated data store for contact information, while later applications expanded the same approach to customers, invoices, patients, medications, passwords, and other structured records.

This architecture allowed increasingly sophisticated data-management capabilities to remain contained within the PDF itself while keeping the underlying JSON storage invisible during normal use.

The combination of JavaScript objects, JSON serialization, and hidden data fields consequently became a fundamental part of the EasyPDF™ record-management architecture.

EasyPDF™ Engineering Principle

Use hidden PDF fields as internal storage containers when structured application data must remain persistent within the document but should not become part of the visible user interface.