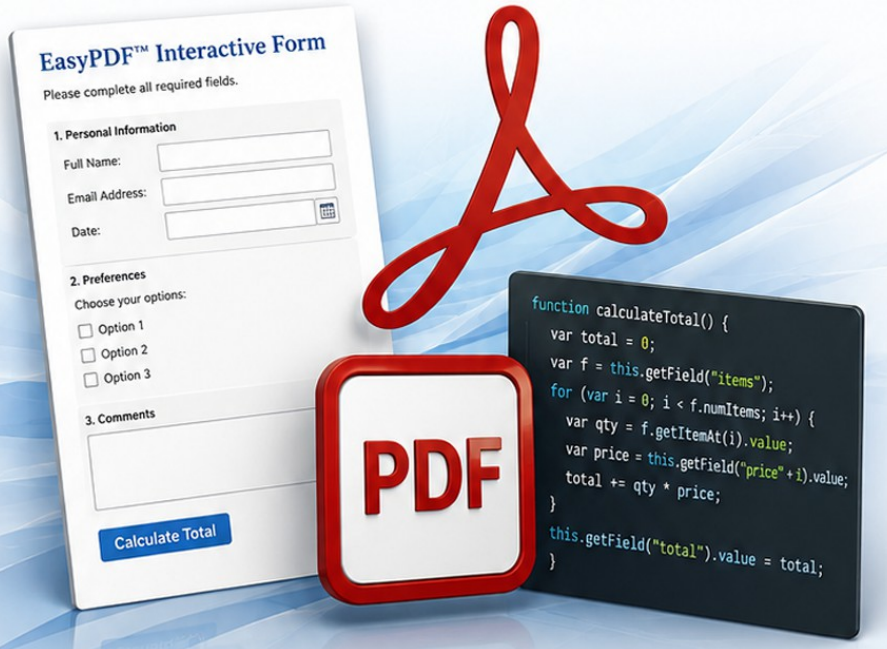
EasyPDF™

Acrobat JavaScript

REFERENCE

THE COMPLETE GUIDE TO DYNAMIC PDF FORMS

Techniques • Examples • Best Practices



Form Calculations
Powerful field calculations and validation

Field Validation
Ensure accurate data entry with custom validation

User Interactions
Create dynamic forms that respond to user actions

Data Processing
Manipulate, format, and process data efficiently

Advanced Topics
Explore advanced techniques and best practices

Everything You Need to Create Smart, Professional PDF Forms with Acrobat JavaScript

TRUSTED • RELIABLE • PROVEN
SINCE 2003

EASYPDF™ FORMS

DOCUMENT AUTOMATION MADE EASY

Preface

A Practical Reference

Documenting the design, evolution, and refinement of the EasyPDF™ family of interactive PDF forms.

Compiled and maintained by

Marty Karr

with the assistance of **ChatGPT (OpenAI)**

Preview Edition

Version 0.2

This publication is released as a Preview Edition and documents practical Acrobat JavaScript design techniques developed through decades of real-world PDF application development. It provides an early look at the EasyPDF™ Acrobat JavaScript Reference while the complete manuscript remains under development. Future publication plans have not yet been finalized.

Copyright

About This Reference

Unlike traditional programming manuals, this reference is based on practical engineering experience gained while designing and maintaining the EasyPDF™ family of interactive PDF forms. Rather than documenting every Acrobat JavaScript feature, it focuses on the techniques, design patterns, and lessons learned that consistently produce reliable, maintainable, professional, real-world interactive PDF applications.

EasyPDF™ Acrobat JavaScript Reference

Table of Contents

Preface ----------

Section 1 - Acrobat UI Techniques ----------

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 - JSON Storage ----------

3.1 Why JSON Was Chosen ----------

3.2 Hidden Data Fields ----------

3.3 Nested JSON Structures ----------

3.4 Maintaining Unique Record Ids ----------

3.5 Restore Deleted Records ----------

3.6 Separating Data from the User Interface ----------

Section 4 – Calculations ----------

4.1 Automatic Line Calculations ----------

4.2 Currency Formatting ----------

4.3 Invoice Totals ----------

4.4 Tax Calculations ----------

4.5 Calculation Design Philosophy ----------

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 ----------

Table of Contents (Cont'd)

6.1 Hidden vs Display ----------
6.2 Commit Immediately ----------
6.3 setTimeOut() ----------
6.4 Focus Issues ----------
6.5 Combo Box Behavior ----------
6.6 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 update?Btns() ----------
7.1.3 Separating Trial Limits ----------
7.1.4 Excluding Counters from Drop-Down Lists ----------
7.1.5 Eliminating Blank Drop-Down Entries ----------
7.2 Acrobat Event Model ----------
7.2.1 Event Sequence and Execution Order ----------
7.2.2 Validate, Format, and Calculate Events ----------
7.2.3 Using setTimeOut() ----------
7.2.4 Refreshing the User Interface ----------
7.2.5 Common Event Mistakes ----------
7.3 Design Patterns ----------
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 Tracking

EasyPDF™ Acrobat Javascript Reference

2.2 updateNames() Design Pattern

Problem

Nearly every EasyPDF™ form maintains a drop-down list containing user-defined records such as customers, contacts, patients, medications, websites, invoices, or passwords. Whenever a record is added, deleted, restored, or renamed, the associated drop-down list must immediately reflect the current contents of the underlying JSON data store.

Early implementations rebuilt each drop-down list using application-specific code. Although functional, this resulted in duplicated programming logic and increased maintenance whenever improvements or corrections were required.

Investigation

As additional EasyPDF™ forms were developed, it became apparent that every application performed essentially the same task:

- Read the JSON data store.
- Build an array of valid record names.
- Exclude internal application properties.
- Sort the names when appropriate.
- Repopulate the drop-down list.
- Restore the current selection whenever possible.
- Update the navigation buttons.

The only differences between applications were the names of the fields and the specific JSON object being processed.

This observation suggested that rebuilding a drop-down list should follow a standardized design pattern regardless of the application.

Solution

Each EasyPDF™ form implements a dedicated updateNames() routine responsible for rebuilding its associated drop-down list from the current JSON data.

Rather than allowing individual application routines to manipulate the drop-down directly, every operation that changes the underlying data simply calls updateNames().

The updateNames() routine becomes the single authority responsible for rebuilding the list, restoring the current selection when appropriate, excluding internal application properties, and updating the associated navigation buttons.

Why It Works

By assigning all drop-down maintenance responsibilities to a single function, every record-

management operation behaves consistently.

The application no longer needs to remember how to update the user interface after each individual operation. Instead, add(), delete(), restore(), rename(), and similar routines simply update the data and then call updateNames().

This separation of responsibilities reduces duplicated code while making future enhancements considerably easier to implement.

Design Principle

Whenever multiple operations affect the same user interface element, assign responsibility for maintaining that element to a single dedicated routine.

The application should modify its data first and then allow the specialized update routine to refresh the user interface.

Lessons Learned

• A drop-down list should have one authoritative update routine.

• User interface rebuilding should never be duplicated throughout the application.

• Separating data manipulation from user interface updates simplifies maintenance.

• Improvements made to updateNames() automatically benefit every routine that calls it.

Historical Note

The updateNames() design pattern gradually evolved while developing multiple EasyPDF™ forms including the Password Manager, CRM System, Digital Rolodex, Medication Manager, and Smart Invoice. As each application introduced new features, the advantages of centralizing drop-down maintenance became increasingly apparent. The pattern ultimately became one of the fundamental architectural standards adopted throughout the EasyPDF™ family of forms.

EasyPDF™ Engineering Principle

Whenever possible, create one authoritative function responsible for maintaining each user interface component. All application routines should rely on that function rather than attempting to duplicate its responsibilities.

3.3 Nested JSON Structures

Problem

Early EasyPDF™ forms stored records using relatively simple JSON structures. While this approach worked well for standalone applications, it became increasingly restrictive as more sophisticated relationships between records were introduced.

For example, the Smart Invoice application required each customer to own multiple invoices while preserving customer-specific information independently from invoice-specific information. Similar requirements later appeared in other EasyPDF™ forms as the applications continued to evolve.

Attempting to maintain these relationships using separate JSON objects or unrelated hidden fields quickly became difficult to manage and increased the likelihood of data inconsistencies.

Investigation

Initially, independent JSON objects appeared attractive because they were easy to understand and straightforward to implement.

However, as additional features were added, every relationship between records required additional programming to keep multiple data stores synchronized.

Simple operations such as deleting a customer, restoring an invoice, assigning unique record identifiers, or updating customer information required multiple objects to remain perfectly synchronized. As application complexity increased, maintaining these relationships became unnecessarily complicated.

It became apparent that the data itself should define the relationships rather than relying upon program logic to reconstruct them.

Solution

EasyPDF™ applications gradually adopted nested JSON structures in which related records became part of the parent object.

For example, each customer object maintains its own collection of invoices. Rather than storing invoices independently and attempting to associate them later, every invoice naturally belongs to its customer within the same JSON hierarchy.

This approach places related information together, allowing the application's structure to mirror the logical relationships between the records it manages.

Why It Works

Nested JSON structures significantly reduce application complexity because relationships are preserved automatically by the storage model itself.

Retrieving all invoices for a customer no longer requires searching multiple independent objects.

Deleting or restoring records becomes more reliable because related information remains grouped together.

The resulting code is easier to understand, easier to maintain, and considerably less prone to synchronization errors.

Design Principle

Whenever records possess a natural parent-child relationship, the storage structure should reflect that relationship.

The organization of the data should simplify the application rather than forcing the application to compensate for an unnecessarily complicated storage model.

Lessons Learned

• Store related information together whenever practical.

• Allow the JSON structure to represent real-world relationships.

• Eliminate unnecessary synchronization between multiple data stores.

• Well-designed data structures simplify programming far more than additional application logic.

Historical Note

The transition to nested JSON structures occurred during development of EasyPDF™ Smart Invoice and quickly influenced the design of several additional EasyPDF™ forms. Once the advantages became apparent, nested storage replaced several earlier approaches that relied upon maintaining separate collections of related information. The result was a more maintainable architecture capable of supporting future expansion with considerably less programming effort.

EasyPDF™ Engineering Principle

Design the data structure first.

When the underlying data accurately represents the relationships being modeled, much of the application's complexity disappears naturally.

7.1.1 Selecting the Most Reliable Acrobat Property

Problem

While implementing show/hide navigation arrow buttons for several EasyPDF™ forms, including Smart Invoice, CRM, Digital Rolodex, and Medication Manager, the Acrobat display property produced inconsistent visual results. Although the underlying JavaScript executed correctly, the navigation buttons occasionally failed to update their visible state after records were added, deleted, or restored.

Because the script logic itself was functioning correctly, the inconsistent user interface behavior made the problem particularly difficult to diagnose.

Investigation

Several revisions of the navigation button scripts were tested using the Acrobat display property. Although each revision improved portions of the implementation, the button visibility remained inconsistent under certain conditions.

Further testing indicated that the navigation logic was not at fault. Instead, the problem appeared to be related to Acrobat's handling of the display property for dynamically changing button visibility.

Replacing the display property with the hidden property immediately produced consistent and repeatable behavior across all EasyPDF™ forms.

Solution

The navigation buttons are now shown and hidden using the Acrobat hidden property instead of the display property.

Example

```
var bHide = (fNames.numItems <= 1);

fPrev.hidden = bHide;
fNext.hidden = bHide;
```

The button update routine should be called whenever the contents of the associated drop-down list are rebuilt.

Why It Works

Although both properties control field visibility, practical testing demonstrated that the hidden

property provides more reliable results when dynamically updating Acrobat button fields.

Using the hidden property also simplifies the implementation by eliminating unnecessary display state management.

Design Principle

Whenever possible, use the simplest Acrobat property that consistently produces the desired result.

For dynamically updating button visibility, the hidden property proved to be both simpler and more reliable than the display property.

Lessons Learned

• Do not assume Acrobat properties with similar purposes behave identically.

• A simpler solution is often the more reliable solution.

• When a user interface behaves inconsistently, consider the Acrobat property being used before rewriting otherwise correct program logic.

• Once a reliable implementation has been verified, standardize its use throughout the entire application.

Historical Note

This behavior was discovered while implementing show/hide navigation arrow buttons for multiple EasyPDF™ forms. After extensive testing, the hidden property was adopted as the standard implementation for all future EasyPDF™ projects.

7.1.2 Centralizing update?Btns()

Problem

While implementing show/hide navigation buttons throughout the EasyPDF™ family of forms, the initial approach called update?Btns() from numerous locations within the application. Button updates were performed after adding records, deleting records, restoring deleted records, clearing fields, and in several other routines whenever it seemed necessary.

Although this approach worked, it resulted in duplicated code, increased maintenance, and made it more difficult to determine whether every possible update path had been covered.

Investigation

A review of the application flow revealed that navigation buttons do not actually depend on whether a record is added, deleted, restored, or edited.

Instead, the button visibility depends upon only one condition:

The current contents of the associated drop-down list.

Further analysis showed that every operation capable of changing the drop-down list ultimately calls updateNames() to rebuild the list.

This observation suggested that the responsibility for updating the navigation buttons belonged inside updateNames() rather than throughout the individual application routines.

Solution

The update?Btns() function was moved into updateNames().

Whenever updateNames() rebuilds the drop-down list, it immediately updates the navigation buttons before returning control to the calling routine.

As a result, add(), delete(), restore(), and similar functions no longer require individual calls to update?Btns().

This centralizes all button updates into a single location.

Why It Works

The visibility of the navigation buttons is determined solely by the contents of the drop-down list.

Since updateNames() is responsible for rebuilding that list, it naturally becomes the single location

responsible for updating the associated navigation buttons.

This approach eliminates redundant function calls while ensuring that button visibility always remains synchronized with the current drop-down contents.

Design Principle

Whenever multiple routines produce the same user interface update, centralize that update into the single function responsible for maintaining the affected user interface component.

This reduces duplicated code, simplifies future maintenance, and minimizes the possibility of inconsistent behavior.

Lessons Learned

• Eliminate duplicate function calls whenever a single centralized location can perform the same task.

• A function should be responsible for maintaining the user interface element that it creates or modifies.

• Simplifying program flow generally improves long-term maintainability.

• Well-defined responsibilities make debugging considerably easier.

Historical Note

This design improvement was discovered while integrating show/hide navigation buttons into multiple EasyPDF™ forms, including the CRM System, Digital Rolodex, Smart Invoice, and Medication Manager.

Once the update routine was centralized, the implementation became both simpler and more reliable, and the technique was adopted as the EasyPDF™ standard for all future projects.

EasyPDF™ Engineering Principle

When a user interface element is maintained by a single function, all related updates should occur within that function rather than being scattered throughout the application.

7.1.3 – Separating Trial Limits

Problem

During development of the EasyPDF™ Forms product line, trial-version restrictions were sometimes added directly inside the application functions they affected.

For example, a customer-management application might limit the trial version to three customer records. The quickest way to enforce that restriction would be to place the trial check directly inside the addCustomer() function:

```
function addCustomer() {
if (bTrial && getCustomerCount(oCustomers) >= 3) {
app.alert(
"The trial version allows a maximum of three customers."
);
return;
}

// Continue adding the customer...

}
```

Although this works, the addCustomer() function now performs two unrelated responsibilities.

Its primary purpose is to add a customer record. However, it has also become responsible for determining whether the user's license permits that operation.

As additional trial restrictions are introduced, the same problem can spread throughout the application. Add, edit, delete, save, restore, validation, and navigation functions may gradually become cluttered with licensing conditions.

This makes the application more difficult to read, test, maintain, and convert into a licensed version.

Investigation

The underlying problem was not the trial limit itself. The problem was where the trial-limit logic had been placed.

Trial restrictions are not part of the application's core business logic.

A customer-management function should manage customers. An invoice function should manage invoices. A medication function should manage medications.

Whether the user is allowed to perform an operation under a trial license is a separate concern.

EasyPDF™ Acrobat Javascript Reference

Embedding trial restrictions directly inside the application's primary functions creates several maintenance problems:

• The same trial condition may be repeated in multiple locations.

• Trial-limit values may become hard-coded throughout the document.

• Licensing messages may become inconsistent.

• Changing a trial limit may require editing several unrelated functions.

• Removing trial restrictions from the licensed version becomes more difficult.

• Application functions become responsible for both record management and license enforcement.

The investigation therefore led to a broader design question:

Should the application functions themselves enforce trial restrictions, or should they ask a separate licensing function whether an operation is permitted?

The second approach proved cleaner and more maintainable.

Solution

The trial restriction should be moved into a dedicated helper function whose only responsibility is to determine whether the operation is permitted.

For example:

var TRIAL_CUSTOMER_LIMIT = 3;

function canAddCustomer(oCustomers) {
if (!bTrial)
return true;

```
if (getCustomerCount(oCustomers) >= TRIAL_CUSTOMER_LIMIT) {
    app.alert(
        "The trial version allows a maximum of " +
        TRIAL_CUSTOMER_LIMIT +
        " customer records.\n\n" +
        "Please purchase the licensed version to continue."
    );

    return false;
}

return true;

}
```

The addCustomer() function can then remain focused on its intended responsibility:

```
function addCustomer() {
if (!canAddCustomer(oCustomers))
return;

// Continue adding the customer...

}
```

The addCustomer() function no longer needs to know:

• Whether the document is a trial version.

• What the trial customer limit is.

• How customer records are counted.

• What message should be displayed when the limit is reached.

Those responsibilities now belong entirely to canAddCustomer().

The same pattern can be applied to other record types:

```
if (!canAddInvoice(oInvoices))
return;

if (!canAddContact(oContacts))
return;

if (!canAddMedication(oMedications))
return;

if (!canAddPassword(oPasswords))
return;
```

Each helper function should enforce one clearly defined licensing rule.

Applications containing more than one stored-record type should also use separate helper functions for each trial restriction.

For example, an EasyPDF™ Medication Manager trial version might allow one patient record while permitting several medication records for that patient.

Those limits should not be combined into one general trial-limit function because they apply to different data collections and represent different licensing rules.

```
var TRIAL_PATIENT_LIMIT = 1;
var TRIAL_MEDICATION_LIMIT = 5;

function canAddPatient(oPatients) {
if (!bTrial)
return true;

if (getPatientCount(oPatients) >= TRIAL_PATIENT_LIMIT) {
    app.alert(
        "The trial version allows a maximum of " +
        TRIAL_PATIENT_LIMIT +
        " patient record."
    );

    return false;
}

return true;

}

function canAddMedication(oMedications) {
if (!bTrial)
return true;

if (getMedicationCount(oMedications) >= TRIAL_MEDICATION_LIMIT) {
    app.alert(
        "The trial version allows a maximum of " +
        TRIAL_MEDICATION_LIMIT +
        " medication records."
    );

    return false;
}

return true;

}
```

The application functions then call only the helper associated with the operation being performed:

```
function addPatient() {
if (!canAddPatient(oPatients))
return;

// Continue adding the patient...

}

function addMedication() {
if (!canAddMedication(oMedications))
```

return;

```
// Continue adding the medication...
```

}

This keeps the patient limit independent of the medication limit and prevents one trial restriction from unintentionally affecting another part of the application.

Design Principle

Trial-version restrictions are licensing logic, not application logic.

Every trial restriction should be isolated inside a dedicated, clearly named helper function.

The application's primary functions should ask whether an operation is permitted, but they should not contain the detailed rules used to make that decision.

A function such as addCustomer() should add a customer.

A function such as canAddCustomer() should determine whether the license permits another customer to be added.

Keeping those responsibilities separate produces code that is easier to understand, modify, test, and reuse.

Lessons Learned

The quickest place to add a trial restriction is not necessarily the correct place.

Embedding trial checks directly inside add, edit, delete, save, restore, validation, or navigation functions may work initially, but it gradually mixes licensing concerns with application behavior.

The better approach is to:

• Store trial limits in named constants.

• Create one helper function for each licensing rule.

• Keep trial messages inside the licensing layer.

• Return a simple true or false result to the calling function.

• Keep application functions independent of trial-version details.

• Use separate helpers for separate record types and limits.

This design also makes licensed-version development easier.

When the licensed version does not require trial restrictions, the licensing helpers can return true, the trial flag can be disabled, or the trial-control layer can be removed without rewriting the application's core functions.

Historical Note

This principle became increasingly important as the EasyPDF™ Forms product line expanded beyond simple forms into JavaScript-driven PDF applications containing customer records, contacts, invoices, medications, passwords, and other stored data.

Early trial restrictions could be added quickly to individual functions. However, as applications became more sophisticated, that approach risked spreading trial-related conditions throughout the codebase.

Separating trial limits into dedicated helper functions established a cleaner boundary between the application and its licensing controls.

That boundary made it easier to maintain both trial and licensed editions while preserving one stable body of application logic.

The lesson was straightforward but important:

Do not redesign the application around the trial version.

Build the application correctly first, and place the trial restrictions around it as a separate licensing layer.

7.1.4 Excluding Counters from Drop-Down Lists

Problem

Several EasyPDF™ forms maintain internal counters within their JSON data store to automatically assign unique record identifiers and maintain application state. Examples include customer IDs, invoice numbers, patient IDs, contact IDs, and similar values that are required internally but should never appear as selectable items within user drop-down lists.

During development it became apparent that rebuilding a drop-down list by simply looping through every property contained within the JSON object unintentionally included these internal counters as selectable entries. Although the application continued to function, the user interface displayed information intended solely for internal program use.

Investigation

Initially, the drop-down rebuilding routines assumed every top-level JSON property represented a valid user record. This assumption proved incorrect as additional application features introduced internal properties such as nextCustomerID, nextInvoiceNo, nextPatientID, nextContactID, and similar counters.

Because these values existed alongside the user records within the JSON structure, they were inadvertently treated as record names whenever the drop-down list was rebuilt.

As additional EasyPDF™ forms adopted the same storage architecture, it became evident that filtering these internal properties individually within each application was neither efficient nor maintainable.

Solution

The updateNames() routines were modified to distinguish between actual user records and internal application properties before populating the drop-down list.

Only valid record objects are added to the list. Internal counters and other application variables are ignored during the rebuilding process.

By centralizing this filtering logic within updateNames(), every EasyPDF™ form benefits from the same reliable behavior without requiring application-specific exceptions elsewhere in the code.

Why It Works

Drop-down lists exist solely for user interaction.

Internal counters exist solely to support application logic.

Separating these responsibilities ensures the user interface presents only meaningful record names while preserving all application metadata within the JSON storage structure.

The result is a cleaner interface, more maintainable code, and a reduced likelihood of future programming errors whenever additional internal properties are added to the data store.

Design Principle

Internal application data should remain internal.

Whenever JSON storage contains both user records and application metadata, user interface routines should explicitly filter the data presented to the user rather than assuming every stored property represents a selectable record.

Lessons Learned

• Never assume every JSON property belongs in a user interface control.

• Internal counters should remain invisible to the user.

• Centralize filtering logic within updateNames() rather than scattering exceptions throughout the application.

• Separating application metadata from user-visible data produces a cleaner and more maintainable design.

Historical Note

This design improvement originated during the development of several EasyPDF™ forms after global counters were introduced to automatically generate unique record identifiers. Rather than redesigning the storage architecture, the updateNames() routines were enhanced to recognize and exclude internal counters while rebuilding the associated drop-down lists. The solution proved reliable and was subsequently adopted as the EasyPDF™ standard across all projects.

EasyPDF™ Engineering Principle

User interface controls should display only information intended for the user. Internal application data should remain hidden regardless of how it is stored.

7.2.1 Event Sequence and Execution Order

Problem

One of the most common sources of frustration encountered by Acrobat JavaScript developers is determining why a script that appears to be correct does not always execute when expected. A field may contain the proper value while a calculated total remains unchanged, a validation routine may seem to execute too early or too late, or an interface update may not occur until after the user has completed another action.

These situations often lead developers to conclude that their JavaScript contains a programming error when, in reality, the script itself is functioning exactly as written. The problem lies not in the code, but in understanding when Acrobat chooses to execute that code.

Unlike traditional programming environments where statements generally execute in the order they are encountered, Acrobat is an event-driven application. Every user action initiates one or more events, and Acrobat processes those events according to its own predefined execution sequence. Understanding both the sequence of events and the order in which Acrobat executes them is essential to designing predictable, reliable interactive PDF applications.

Investigation

As increasingly sophisticated EasyPDF™ Forms were developed, it became apparent that Acrobat performs far more work than simply accepting a field value and immediately executing every related script.

Instead, Acrobat divides document processing into a series of individual stages. Depending upon the document design and field type, those stages may include:

- Keystroke processing
- Validation
- Formatting
- Value commitment
- Calculations
- User interface refresh

Each stage serves a specific purpose and must complete before Acrobat advances to the next. This processing model ensures that every event receives valid information from the previous stage before continuing.

For example, calculations cannot reliably execute until Acrobat has determined that user input is valid and committed the final field value. Likewise, formatting cannot display the final appearance of a value until Acrobat knows that value has been accepted. User interface updates naturally occur after the document has reached a consistent state.

Recognizing this sequence explains why calculations, validation routines, or visual updates sometimes appear to execute later than expected even though the JavaScript itself is operating correctly.

Solution

Rather than assuming every operation should occur immediately after a field changes, design every script around the Acrobat event specifically intended to perform that task.

Use each event for the responsibility it was designed to perform.

For example:

- **Use Keystroke events while the user is actively entering data.**
- **Use Validate events to determine whether the completed input should be accepted.**
- **Use Format events to control how accepted data is displayed.**
- **Use Calculate events to update dependent values only after the necessary information is available.**
- **Use Mouse Up events to perform operations initiated by push buttons after the user intentionally activates the control.**

Avoid relocating code simply because another event appears to execute sooner. Instead, determine where the operation logically belongs within Acrobat's processing model and allow Acrobat to complete each stage in its normal execution order.

Applications designed around Acrobat's event architecture require fewer programming workarounds, are easier to debug, and produce far more predictable behavior.

Why It Works

Acrobat's event architecture was intentionally designed to separate document processing into individual, well-defined stages instead of attempting to perform every operation simultaneously.

By completing one event before beginning the next, Acrobat ensures that each stage operates on reliable information. Validation occurs before calculations depend upon user input. Formatting reflects committed values rather than temporary keystrokes. Interface updates occur only after the document has reached a consistent state.

This disciplined execution model protects document integrity while eliminating many timing-related problems that developers mistakenly attribute to JavaScript itself.

Once developers begin thinking in terms of Acrobat's processing sequence rather than individual scripts, debugging becomes significantly easier. Many seemingly complex problems disappear simply because the code has been moved to the event responsible for performing that operation.

EasyPDF™ Acrobat Javascript Reference

Design Principle

Never design Acrobat applications around the assumption that code executes immediately after a field changes.

Instead, design every routine with the understanding that Acrobat—not the JavaScript—controls the sequence and timing of event execution.

Assign each programming task to the event specifically designed to perform that responsibility, then allow Acrobat to complete its normal processing sequence without interference.

Working with Acrobat's event model rather than attempting to bypass it produces applications that are simpler, more reliable, and considerably easier to maintain.

Lessons Learned

- **Many apparent JavaScript failures are actually misunderstandings of Acrobat's event timing.**
- **Correct JavaScript executed during the wrong event frequently produces incorrect or unpredictable results.**
- **Acrobat—not the script—determines when each event executes.**
- **Selecting the proper event often eliminates unnecessary programming workarounds.**
- **Reliable interactive PDF applications are built by working with Acrobat's execution sequence rather than attempting to force operations to occur prematurely.**
- **Understanding when code executes is often just as important as understanding what the code does.**

EasyPDF™ Engineering Principle

One of the most valuable lessons learned during the development of EasyPDF™ Forms is that Acrobat's event model should never be treated as an obstacle to overcome. It is the framework that governs every interactive document.

Before rewriting JavaScript that appears to be malfunctioning, first determine whether the routine is executing during the proper Acrobat event. More often than not, the solution is not additional code, but placing existing code in the event Acrobat intended for that purpose.

Understanding Acrobat's event sequence and execution order transforms debugging from a process of trial and error into one of logical analysis. Once developers learn to work with Acrobat's event architecture instead of against it, their applications become more predictable, more maintainable, and significantly easier to develop.

EasyPDF™ Acrobat JavaScript Reference

A.1 Future Publication Notes

This reference was not written from theory.

Every technique, design pattern, and solution documented in this reference was developed, tested, and refined while creating the EasyPDF™ family of interactive PDF forms.

Many of the topics discussed here are not found in standard Acrobat JavaScript references because they were discovered through years of practical development, experimentation, debugging, and refinement while solving real-world problems.

The purpose of this reference is to document those lessons—not only to preserve them for future EasyPDF™ development, but also to help others avoid rediscovering the same solutions through trial and error.

A.2 Using the HTML5 `download` Attribute for PDF Downloads

Subject: Missing HTML download Attribute Prevented Matomo Download Tracking

Problem:

Following a Windows 11 / Microsoft Edge update, PDF download links that previously behaved as expected began opening PDFs directly in the browser instead of downloading them. As a result, Matomo Analytics failed to record PDF downloads as download events.

Cause:

The HTML anchor (<a>) tags linking to PDF files did not include the HTML5 download attribute. Browser behavior apparently changed, causing Edge to display PDFs inline rather than treating them as downloadable files.

Solution:

Add download attribute to all PDF download links.

Before:

```
<a href="licensed forms/easypdf-medication-manager-licensed-copy.pdf"
   title="Download FREE Lifetime License (Limited-Time Offer)">
   EasyPDF™ Medication Manager
</a>
```

After:

```
<a href="licensed forms/easypdf-medication-manager-licensed-copy.pdf" download
   title="Download FREE Lifetime License (Limited-Time Offer)">
   EasyPDF™ Medication Manager
</a>
```

Result:

PDF files download correctly instead of opening inline in Microsoft Edge.

Matomo Analytics once again records PDF downloads correctly.

No changes to Matomo configuration or tracking code were required.

Lesson Learned:

Do not rely on browser default behavior for PDF links. Always include the HTML5 download attribute on EasyPDF™ Forms download links to ensure consistent browser behavior and reliable Matomo Analytics download tracking.

A.3 Creating a Professional PDF Title Page Illustration Cover

Subject

Creating a Professional PDF Title Page Illustration Cover

Problem

After creating a custom title page separately from the manuscript and inserting it into the completed PDF, Acrobat displayed the title page correctly at the selected magnification (Fit Width). However, advancing to the next page caused Acrobat to automatically change the zoom level, resulting in an inconsistent viewing experience throughout the remainder of the document.

Cause

The title page and manuscript pages were created using different applications and later combined into a single PDF. Although the pages appeared to be the same physical size, they retained different internal page definitions (such as page geometry and page boxes). Acrobat recalculated the page magnification when transitioning from the title page to the remaining pages.

Solution

Rather than creating the title page as a separate PDF and inserting it into the completed manuscript, insert the title page artwork directly into the word processor document (OpenOffice Writer in this case) as the first page. Export the entire manuscript as a single PDF so all pages are generated using the same PDF export engine.

Before

- Title page created separately.
- Separate PDFs combined into a single document.
- Acrobat changed magnification when advancing beyond the title page.
- Inconsistent viewing experience.

After

- Title page inserted directly into the manuscript.
- Entire document exported as one PDF.
- All pages share consistent page geometry.
- Acrobat maintains a consistent viewing magnification throughout the document.

Result

The zoom change between the title page and the remaining manuscript pages was completely eliminated. The document now opens and navigates consistently, providing a more professional reading experience.

Lesson Learned

Rather than creating the cover or title page separately and inserting it into an existing PDF, insert the artwork directly into the source manuscript before exporting the document to PDF. Generating the entire publication as a single PDF ensures consistent page geometry, prevents unexpected page magnification changes, and produces a more polished, professional publication.

A.4 Local Preview Delays Caused by Matomo Tracking

Symptom

After adding the Matomo tracking code to every page, local website previews from CoffeeCup HTML Editor (or opening pages directly from a local `file:///` path in Edge) may pause for several seconds before displaying the page.

The page eventually loads correctly, but the delay can make it appear that CoffeeCup HTML Editor or the browser is malfunctioning.

Cause

The standard Matomo tracking code executes immediately when the page loads and attempts to retrieve the tracking script from the Matomo server.

When previewing pages locally using the `file:///` protocol, the browser still attempts to initialize the tracking code even though the page is not being served from a web server. This unnecessary network activity can significantly delay local previews.

The issue does not affect the live website.

Solution

Execute the Matomo tracking code only when the page is served using HTTP or HTTPS.

Example

```
if (location.protocol !== "file:") {
   var _paq = window._paq = window._paq || [];
   _paq.push(['trackPageView']);
   _paq.push(['enableLinkTracking']);
   (function () {
      var u = "https://www.easypdfeforms.com/matomo/";
      _paq.push(['setTrackerUrl', u + 'matomo.php']);
      _paq.push(['setSiteId', '1']);
      var d = document,
         g = d.createElement('script'),
         s = d.getElementsByTagName('script')[0];
      g.async = true;
      g.src = u + 'matomo.js';
      s.parentNode.insertBefore(g, s);
   })();
}
```

Benefits

- **Restores fast local previews.**
- **Prevents unnecessary network requests during development.**
- **Prevents local testing from appearing in Matomo analytics.**
- **Leaves live website tracking completely unchanged.**

Lesson Learned

Third-party analytics scripts should generally be disabled during local `file:///` development unless they are specifically required for testing. Doing so improves development performance and keeps analytics data free of local testing activity.