The Happy Project Hooker

Doug Hennig

Project objects and ProjectHooks are new in VFP 6. Together, they provide the ability to add to or change the behavior of the Project Manager. Doug's article presents tools that allow you to have project-specific field mappings, perform text search on all files in a project, log activities, and have a toolbar of useful project functions.

Until VFP 6, the only "official" way to work with a project was through the Project Manager. While the Project Manager is a fine tool, it has a fixed feature set and user interface. Often, developers wanted more functionality, and had to resort to hacking the PJX file to get it. Here are some typical examples:

- Many frameworks (such as Codebook, Stonefield AppMaker, and FoxExpress) creating a starting project for an application either by copying a template project file or creating a PJX file from scratch. In either case, the PJX file was opened as a table and the files it referenced programmatically altered by manipulating the records based on what should go in the application.
- Tools like FoxDoc and the Documenting Wizard had to open the PJX file as a table to read the contents so they knew which files to document.
- Many developers have written utilities over the years to do things with the files in a project. For example, I've written utilities to open all the "table-based" source code files (SCX, VCX, MNX, and FRX) in a project and pack them to minimize space for transmission or storage, and to ZIP all the source code files for an application for backup purposes. These utilities had to open the PJX file as a table and process the records in it.

VFP 6 changes our approach to projects. There's no longer a need to hack the PJX file because VFP 6 provides a new interface to projects: whenever a project is opened, a Project object is now instantiated. Interestingly, Project objects are Automation, not native VFP, objects. This has several consequences:

- Errors that occur during Project object manipulation are OLE errors, not VFP errors.
- The Project object model uses more standard property names (for example, Count) than many native VFP objects do (which seem to have a different name for every Count property, such as PageCount, FormCount, ButtonCount, etc.)
- Automation clients and ActiveX controls can access projects through Project objects.
- Properties and members of Project objects don't always show up in the VFP Debugger because for performance reasons, VFP doesn't read the typelib of Automation objects until a property is actually referenced.

There are two ways you can work with a Project object: through the Application (or _VFP) Projects collection and Application.ActiveProject. Similar to VFP's Forms collection and ActiveForm property, there's an object reference to every open project in the Projects collection and a reference to the active project in the ActiveProject property. Like ActiveForm, ActiveProject is undefined (the TYPE function returns "U") when there's no open project.

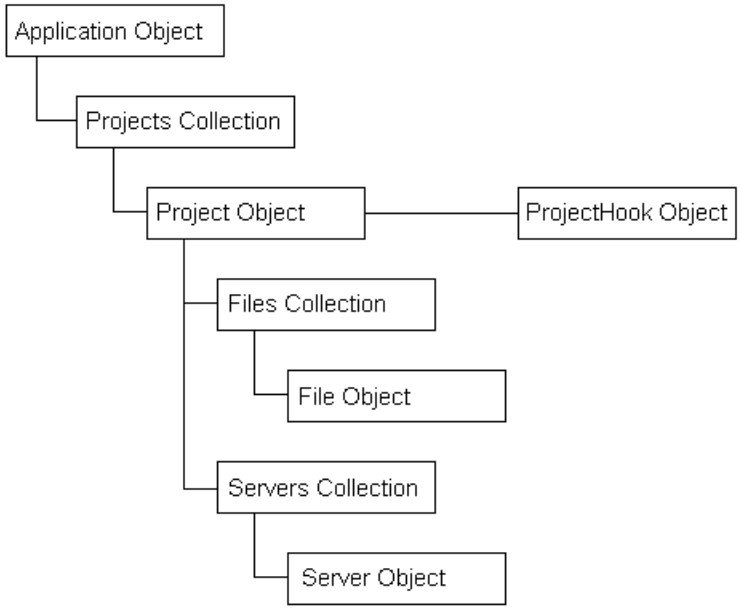
As if Project objects weren't enough, VFP 6 also gives us ProjectHook objects. A ProjectHook object may be automatically instantiated when a project is opened (you have complete control over whether ProjectHooks are used or not), and its events are fired whenever something is done to the project: adding or removing files, modifying or running files, or building the project. ProjectHook is a new VFP base class, so we can create our own ProjectHook classes and put any code into them we want. Thus, we have a hook into the operation of the Project Manager, and can even completely manipulate projects without using the Project Manager at all.

Because we'll need some background into Project objects and ProjectHooks before we can build some tools using them, this article will have more background than is usual for my articles. Please bear with me; we'll get to the good stuff soon.

Project Object Model

The object model for the Projects collection and Project objects is shown in Figure 1. As you can see, a Project object contains a reference to a ProjectHook object, and has both Files and Servers collections. We'll look at these objects and collections, as well as their properties, events, and methods.

Figure 1. Project Object Model.



Projects Collection

The Projects collection only has one property, Count (the number of Project objects in the collection), and one method, Item (which returns a reference to the specified project). Here are some examples:

```
? Application.Projects.Count
? Application.Projects.Item[1].Name
```

Like other collections, you can specify an index to the collection itself rather than having to use the Item method:

```
? Application.Projects[1].Name
```

One really cool thing about this collection (and other new collections in VFP) is that you can reference a project by index number or name; the following code works just fine as long as there's an open project called SOURCE (you must specify the extension or you'll get an error, but you don't have to specify the path):

```
? Application.Projects['source.pjx'].BuildDateTime
```

Project Object

The properties of Project objects are listed in Table 1; properties common to all VFP base classes are omitted. You'll recognize many of these properties from the Project Info and EXE Version dialogs available from the Project Manager. In this table, "Access" is either read-only (R/O) or read-write (R/W) and "Type" is the data type of the property ("L" for logical, "C" for character, "O" for object, etc., plus "Coll" for collection).

Table 1. Project Object Properties.

Property	Access	Type	Description

AutoIncrement	R/W	L	.T. to increment the VersionNumber property when an EXE or DLL is built.
BuildDateTime	R/O	T	The date and time of the last build.
Debug	R/W	L	.T. to include debugging information for each compiled file.
Encrypted	R/W	L	.T. to encrypt files as they're compiled.
Files	R/O	Coll	A collection of File objects.
HomeDir	R/W	C	The home directory of the project.
Icon	R/W	C	The icon used when building an EXE.
MainClass	R/O	C	The name of the ActiveDoc class used as the main program (only used when MainFile contains the name of the VCX where this class is defined).
MainFile	R/O	C	The main program in the project, set using the SetMain method.
Name	R/O	C	The name of the project, including the fully qualified path.
ProjectHook	R/W	O	A reference to the ProjectHook object associated with the project.
ProjectHookClass	R/W	C	The name of the class the ProjectHook object is instantiated from.
ProjectHookLibrary	R/W	C	The class library the ProjectHook class is defined in.
SCCProvider	R/O	C	The name of the source code control provider if the project is under source code control.
ServerHelpFile	R/W	C	The name of the help file for the typelib used for server classes.
ServerProject	R/W	C	The first part of the ProgID for the Automation server created by the project; the default is the same as the Name property.
Servers	R/O	Coll	A collection of Server objects.
TypeLibCLSID	R/O	C	The typelib Registry CLSID.
TypeLibDesc	R/W	C	The typelib description. The default is the name of the project + "Type Library".
TypeLibName	R/O	C	The fully qualified path to the typelib for the project.
VersionComments	R/W	C	Version comments.
VersionCompany	R/W	C	Version company name.
VersionCopyright	R/W	C	Version copyright.
VersionDescription	R/W	C	Version description.
VersionNumber	R/W	C	Version number in Major.Minor.Revision format.
VersionProduct	R/W	C	Version product name.
VersionTrademarks	R/W	C	Version trademarks.
Visible	R/W	L	.T. (the default) to display the Project Manager.

Project objects have the following methods (see the VFP documentation for syntax):

- Build: builds the project and returns .T. if successful. Parameters specify the name of the file to build, the type of file (APP, EXE, etc.), whether all files should be compiled, whether errors should be displayed after building, and whether new GUIDs should be generated. The following code builds an APP with the same name as the project:

```
oProject.Build(, 2)
```

- CleanUp: packs the project and optionally removes object code.
- Close: closes and releases the project.
- Refresh: refreshes the project by re-reading DBC and VCX files, and optionally the source code control status of each file.
- SetMain: sets or clears the main file for the project:

```
oProject.SetMain('startup.prg')
```

Files Collection

The Files collection is a collection of File objects, one for each file in a project. It has one property, Count (the number of files in the collection), and two methods, Item (which returns a reference to the specified file) and Add (which adds a file to the project and returns an object reference to it). Here are some examples:

```
oProject = Application.ActiveProject
? oProject.Files.Count
? oProject.Files[1].Name
? oProject.Files['myclasses.vcx'].Description
oFile = oProject.Files.Add('myprogram.prg')
```

File Object

The properties of File objects are listed in Table 2; properties common to all VFP base classes are omitted.

Table 2. File Object Properties.

Property	Access	Type	Description
CodePage	R/O	N	The code page for the file.
Description	R/W	C	The description shown in the Project Manager.
Exclude	R/W	L	.T. if the file is excluded from the project.
FileClass	R/O	C	The class a form is based on.
FileClassLibrary	R/O	C	The library the form's class is defined in.
LastModified	R/O	T	The date and time the file was last modified.
Name	R/O	C	The name of the file, including the fully qualified path.
ReadOnly	R/O	L	.T. if the file is read-only.
SCCStatus	R/O	N	The source code control status of the file. 0 means the file isn't source controlled; see the VFP documentation for other values (such as checked out or not).
Type	R/O	C	The file type: V for VCX, P for program, R for report, B for label, K for form, Q for query, L for API library, D for table, d for database, Z for APP, M for menu, T for text, and x for other.

File objects have the following methods (see the VFP documentation for syntax):

- AddToSCC: adds the file to source code control.
- CheckIn: checks the file in.
- CheckOut: checks the file out.
- GetLatestVersion: gets a local copy of the latest version of the file, but doesn't check it out.
- Modify: open the file in the appropriate editor or designer (specify the class name if the file is a VCX).
- Remove: removes the file from the project, and optionally deletes it.
- RemoveFromSCC: removes the file from source code control.
- Run: runs the file (preview in the case of reports and labels).
- UndoCheckOut: performs an undo checkout.

Servers Collection

The Servers collection is a collection of Server objects, one for each Automation server in a project. It has one property, Count (the number of files in the collection), and one method, Item (which returns a reference to the specified server).

Server Object

The properties of Server objects are listed in the table below; properties common to all VFP base classes are omitted.

Table 3. Server Object Properties.

Property	Access	Type	Description
CLSID	R/O	C	The Registry CLSID for the server.
Description	R/W	C	The server's description in the Registry.
HelpContextID	R/W	N	The help context ID for the typelib.
Instancing	R/W	N	Specifies how the server is instantiated: 1 (the default) for single use, 2 for not creatable outside of VFP, or 3 for multi-instance.
ProgID	R/O	C	The Registry ProgID for the server.
ServerClass	R/O	C	The class name of the server.
ServerClassLibrary	R/O	C	The library the server class is defined in.

ProjectHooks

As I mentioned earlier, VFP has a new ProjectHook base class. A ProjectHook object may be automatically instantiated when a project is opened (you have complete control over whether ProjectHooks are used or not), and its events are fired whenever something is done to the project: adding or removing files, modifying or running files, or building the project. Like Project objects, ProjectHooks are actually ActiveX objects, not native VFP objects.

You can set a ProjectHook for individual projects or globally. To specify which ProjectHook class to use for an individual project, choose the Project page of the Project Info dialog and choose the desired class for the Project Class. You can also set the ProjectHookClass and ProjectHookLibrary properties of the Project object, although this won't instantiate the ProjectHook class until the project is closed and reopened. To set a ProjectHook for all projects that don't have individually specified ProjectHooks, use the Projects page of the Tools Options dialog. You can also instantiate a ProjectHook object and store a reference to it in a Project object's ProjectHook property, but this provides a non-persistent hook to the project.

Once a ProjectHook class has been specified for a project, the ProjectHook object is automatically instantiated whenever the project is opened, and it's destroyed when the project is closed. If you want to open a project but not instantiate the ProjectHook, use the new NOPROJECTHOOK clause of the MODIFY PROJECT command. You can also use NOPROJECTHOOK with CREATE PROJECT to create a project without the global ProjectHook. These two commands have another new clause, NOSHOW, that instantiates the Project and ProjectHook objects, but doesn't display the Project Manager, so you can manipulate a project without any user interface at all.

ProjectHook Events

ProjectHook objects don't have any properties beyond those of other VFP base classes, but have several specific events that automatically fire when something is done to the project (see the VFP documentation for parameters these events receive):

- AfterBuild: fires after the project build is complete.
- BeforeBuild: fires before the project build is started, either by clicking on the Build button in the Project Manager, calling the Build method of a Project object, or using one of the BUILD commands. Putting NODEFAULT in the BeforeBuild event of the ProjectHook prevents the build from continuing.
- OLEDragOver: this event fires when a file is dragged over the TreeView area of the Project Manager window.
- OLEDragDrop: this event fires when a file is dropped on the TreeView area of the Project Manager window. After this event fires, the QueryAddFile event fires.
- QueryAddFile: fires when a file is added to the project, either by clicking on the Add button in the Project Manager (after the user has chosen the file in the Open File dialog), by clicking on the New button (after the file is saved and the appropriate editor or designer window is closed), when a file is dropped on the Project Manager window (after the OLEDragDrop event fires), or when the Add method of the Project object's File collection is called. Since this fires after the file has been chosen or created, it can't be used to display a dialog of which class to create a form from, for example. Putting NODEFAULT in the QueryAddFile method prevents the file from being added to the project.
- QueryModifyFile: fires upon clicking the Modify button in the Project Manager or calling the Modify method of a File object. Putting NODEFAULT in the QueryModifyFile method prevents the file from being modified.
- QueryRemoveFile: fired when the Remove button in the Project Manager is clicked or the Remove method of a File object is called. Putting NODEFAULT in the QueryRemoveFile method prevents the file from being removed.
- QueryRunFile: fired when the Run button in the Project Manager is clicked or the Run method of a File object is called. Putting NODEFAULT in the QueryRunFile method prevents the file from being executed.

Extending Functionality

Because a ProjectHook object is automatically instantiated when a project is opened from a class we defined, and its events are automatically called when something is done with the project, we have the ability to define what happens when the user works with a project. I'm sure you can think of all kinds of things you could do with this capability; we'll discuss some of the ideas I've come up with.

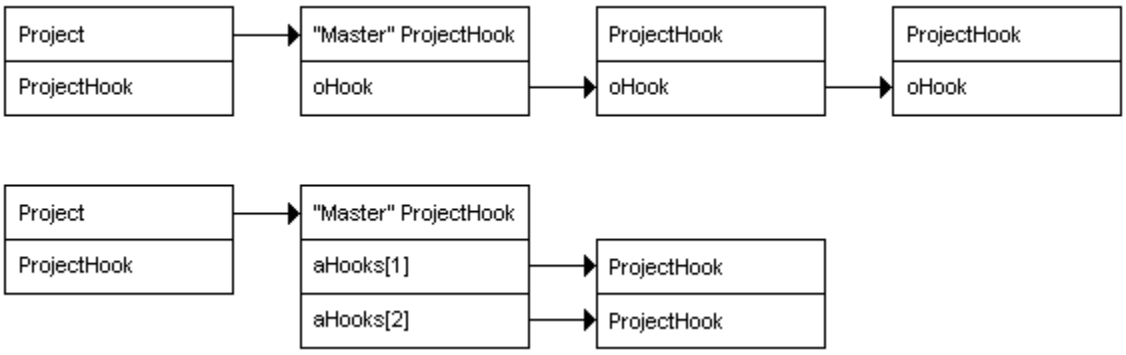
Before we discuss what type of tools you can build, however, keep one thing in mind: avoid the temptation to build a ProjectHook with everything including the kitchen sink in it. There's a number of good reasons for making your ProjectHook classes granular:

- Not all functionality may be appropriate for all projects. You may want to vary the behavior of some functions for certain types of projects (for example, typical data entry applications versus Automation server projects versus ActiveDoc applications). It'd be a ugly class with lots of CASE statements that had to implement this in one place.
- Using dynamic composition instead of subclassing is cleaner, makes a simpler class hierarchy, and may be the only means possible if someone else wrote a ProjectHook you want to use.
- You may want to give someone else some but not all the functionality of ProjectHooks you've created, or use ProjectHooks they've created to add new tools to your arsenal without having to modify your own.

There are several ways you can build modular ProjectHook classes and use dynamic composition to put them together:

- You can chain ProjectHook objects together using hook operations. For example, the MyActivityTracker class in the sample files for this session is a subclass of the Activity_Tracker ProjectHook included with the VFP samples (subclassed because I wanted activities logged to a table rather than a cursor). Rather than copying the code from this class into SFProjectHook (my ProjectHook base class), I simply add it to the ProjectHook chain when a project is opened and can thus take full advantage of its behavior without even knowing how it does its job. Interestingly, only the first object in the chain needs to be based on the ProjectHook class; other objects in the chain can be of any class.
- You can use a "master" ProjectHook object and pass messages to a collection of other ProjectHooks (this is a parallel approach as opposed to the serial approach used by hook operations). The public domain ProjectHookX tool designed by Toni and Mike Feltman of F1 Technologies (developers of Visual FoxExpress), and written by Toni, uses this technique to create a simple "tree" of ProjectHooks.
- You can use both techniques to build more complex trees.

Figure 2. Chaining ProjectHooks.



Examples

ProjectHooks allow you to create simple tools to boost your productivity without a lot of effort. Let's look at some examples of ProjectHook classes to give you an idea of the kinds of tools we can build. Before trying these out yourself, open the Projects table and ensure that the paths you see in the Project field match the location where you installed the sample files on your system. Do the same with the mHookLib field in the PJXHooks table in the Project3 subdirectory.

The Activity_Tracker sample that comes with VFP (located in _samples + "Solution\Tahoe\Project_Hook.vcx") logs project activities to a cursor. Every event in this class has a call to This.WriteLog(<action>, <filename>). WriteLog simply adds a record to the cursor indicating what activity took place. The Destroy method has code that asks if you want to view the log, and if so, browses the cursor. As I mentioned earlier, I've subclassed Activity_Tracker into MyActivityTracker so it can log to a table rather than a cursor.

For a more complex example that comes with VFP, look at AppBldr.scx and the AppHook class in AppHook.vcx. They are parts of VFP's Application Wizard that provide options for pieces of an application and update the Project object and meta data tables with information entered in the Application Builder dialog.

SFProjectHook (in SFProj.vcx, available from the Subscriber Downloads site) is my ProjectHook base class. This class doesn't have a lot of functionality; it'll usually be subclassed to provide the behavior you want. It supports hook operations with an oHook property that points to the next object in the chain. ProjectHook events, such as QueryAddFile and BeforeBuild, are passed on to the hooked object, so all objects in the chain have a chance to process the event. It stores a reference to the project it maintains in its oProject property, the project's directory in cProjectDirectory, and its own directory (so other pieces can be found if necessary) in cDirectory. It has an oToolbar property so a toolbar can be associated with the ProjectHook if desired (the class and library for the toolbar are specified in the cToolbarClass and cToolbarClassLibrary properties). If SFProjectHook is associated with a new project (Files.Count = 0), the custom NewProject method is fired. This allows you to, for example, create a wizard in which you can specify the type of project or what library or framework files should be added to it automatically. See its About method for documentation on this class.

SFProjectHookRegistry is a subclass of SFProjectHook that saves and restores field mappings, the template class used for new forms, and the _INCLUDE setting. This allows you to have project-specific settings that are automatically restored when the project is opened. It uses a free table called Projects to store this information, and collaborates with the FoxReg class that comes with VFP (home() + "FFC\Registry.vcx") to read from and write to the Registry settings where these properties are stored.

SFProjectHookFind is another subclass of SFProjectHook that provides a text search capability. The advantage of this class over the Filer tool that comes with VFP is that Filer checks all files in a specific directory, while SFProjectHook searches all files in the project (which may be spread over many directories). The FindText method of this class searches the project's files for the specified string and puts hits into its aFind array property. Another class, SFFindText, provides a visual interface to SFProjectHookFind: it allows you to specify the text to search for, displays the hits in a grid, and allows you to edit the selected file.

To see how some of these tools can be chained together to provide a full set of functionality for a project, look at the MyProjectHook class in SFTest.vcx. It's a subclass of SFProjectHookFind (so it has the text search feature) that simply instantiates and hooks SFProjectHookRegistry and MyActivityTracker objects, and specifies which toolbar class and library to use (the MyProjectToolbar class in SFTest.vcx). Open the X project in the Project1 subdirectory of the sample files directory and notice the project toolbar that appears. It provides speed buttons to build an APP or EXE (avoiding the seemingly endless array of options and mouse clicks necessary to build a project from the Project Manager), perform a text search, and save project-specific settings.

Open the Tools Options dialog, activate the Field Mappings page, and notice all data types are mapped to a class in the VFP FFC _Base.vcx. Choose the Forms page and notice the template form is set to _Form in this same vcx. Finally, type WAIT WINDOW _INCLUDE in the Command window and notice the default include file is FOXPRO.H. Now close the project and open the Y project in the Project2 subdirectory, and look at the Field Mappings and Forms pages of the Tools Options dialog and the value of _INCLUDE. These project-specific settings were restored from the record for this project in Projects.dbf by the SFProjectHookRegistry object in the ProjectHook chain. While the Y project is still open, click on the Find button in the toolbar, enter "classlibrary" as the text to search, and click on the Find button. When the list of hits appears in the grid, either double-click on a item in the grid or select an item and click on the Edit button to edit the file.

When you close the project, notice the toolbar doesn't disappear immediately. This is because the ProjectHook wasn't destroyed when the project was closed. VFP tried to destroy it, but because the toolbar contains an object reference to it, the ProjectHook stays alive. A timer is used in the toolbar to destroy the ProjectHook and toolbar when the project is no longer open.

ProjectHookX is a utility generously placed in the public domain by Toni Feltman. The main class for this utility, cPJXHookX (in PJXHook.vcx in the ProjectHookX subdirectory), has an aHooks array property that contains references to other ProjectHook objects. Each ProjectHook event has code that sends a message to every ProjectHook in the "tree" of hooks. The class has an AddHooks methods that displays a form in which you can define which ProjectHooks to instantiate when the master ProjectHook is

instantiated (these are stored in a PJXHooks table). You can also call the Add2HooksArray method to programmatically (but not persistently) add an instantiated ProjectHook object to the tree. An object reference to the master ProjectHook is stored in the public variable _oPJXHook so you can call its methods from the Command window without having to type Application.ActiveProject.ProjectHook in front of everything.

To see ProjectHookX in action, open the Z project in the Project3 subdirectory, then type _oPJXHook.AddHooks() in the Command window to display a list of ProjectHooks instantiated. To search for text, type _oPJXHook.aHooks[1].FindText("error") in the Command window, then display _oPJXHook.aHooks[1].aFind in the Debugger. Modify a file, then select and browse the ProjectLog table to see how project logging is done.

Other Ideas

Here are some other things you could use ProjectHooks for:

- "Test" vs. "release" builds: test builds don't increment the build number (or just a minor part of it) and have the Debug property set on. Release builds increment a major part of the build number and have the Debug property set off.
- A New Project Wizard could allow you to choose from a list of templates (such as an ActiveDoc application, a data entry application, a tool application, etc.) and would automatically add library or framework files to the new project based on the template. It could also have checkboxes to include certain files such as a graphing library.
- Poor person's version control: the Activity_Tracker VFP sample could be expanded to log changes to the project into a table, including things such as datetime, user, project, file, and class. After a file is modified, a dialog would appear asking the user to describe in detail what changes were made (this would be done in the QueryModifyFile method).
- Utilities: I mentioned earlier that I've created utilities that hack the PJX file for packing, zipping, documentation, etc. These utilities will be much easier to build in VFP 6 because they don't require knowledge of the PJX file and can be executed while the Project Manager is open. Another useful tool would be a project audit utility; it would ensure every file has its Description property filled in, that the project has the Version properties filled in, etc.

Conclusion

Thanks to the new Project objects and ProjectHooks in VFP 6, we can now add to or change the behavior of the Project Manager. This allows us to create tools that give us more capabilities or more productivity than we get with the Project Manager. I expect we'll be seeing a lot of such tools in the months ahead.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit and Stonefield Query. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997 and 1998 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP). CompuServe 75156,2326 or dhennig@stonefield.com.