

Forum:[Hack Forums](#)

Thread: [HF-Sharp - an open source .NET API Wrapper for HackForums](#)

JUSTIN - May 12, 2019, 5:27 pm

Hello everybody. I was recently given UB3R randomly so I can actually use the API now.

I made a .NET wrapper for the API, to make it easier for anybody here to implement the API into your C#/VB projects.

Official API documentation can be found here: <https://hackforums.net/showthread.php?tid=5665556>
You can generate an API key here: <https://hackforums.net/apikey.php> (you must be UB3R or R3KT)

Here's the repository on github: <https://github.com/ooojustin/HF-Sharp>

Here's the nuget package: <https://www.nuget.org/packages/HF-Sharp>

Here's a quick video demonstrating implementation/basic usage:

[Video: https://www.youtube.com/watch?v=Dtbww4_mht8]

Unrelated to the current forum section, but there's also a PHP version:
<https://github.com/ooojustin/HF-PHP>

Have fun! If you have any questions feel free to ask in this thread.

0xDEAD10CC - May 12, 2019, 9:20 pm

If you're using Newtonsoft I'd highly advise using the json attributes rather than naming the identifiers to match what the json identifiers are supposed to be. Here's a few that I think this API would benefit from:

- https://www.newtonsoft.com/json/help/html/T_Newtonsoft_Json_Converters_StringEnumConverter.htm

- <https://www.newtonsoft.com/json/help/html/SerializationAttributes.htm>

I would also avoid using dynamic to make this framework more type safe. Perhaps some use of async/await would be nice as well rather than forcing these methods to rely on synchronous API calls with an internal webclient that also can't be manipulated. When building a framework, use of properties is also preferred vs public fields (which also aren't immutable), as they make it much easier to avoid breaking changes to future versions of an API. Encapsulation is the key to most frameworks, and I would expose more of this implementation through the use of interfaces and allow some of these classes to be more friendly to inheritance to modify and/or extend existing behavior.

The one thing to keep in mind is that this API is meant to be RESTful. The representation of these classes should represent the equivalent of the resources and collections that the API was designed to expose.

I'm not sure how much of the API changed when I initially was going to write something for this, but the design itself didn't follow RESTful API guidelines too well, so I abandoned that thought.

JUSTIN - May 12, 2019, 10:08 pm

(May 12, 2019 09:20 PM)0xDEAD10CC Wrote: [\[CITE\]](#) If you're using Newtonsoft I'd highly advise using the json attributes rather than naming the identifiers to match what the json identifiers are

supposed to be. Here's a few that I think this API would benefit from:

- https://www.newtonsoft.com/json/help/html/T_Newtonsoft_Json_Converters_StringEnumConverter.htm
- <https://www.newtonsoft.com/json/help/html/SerializationAttributes.htm>

I would also avoid using dynamic to make this framework more type safe. Perhaps some use of async/await would be nice as well rather than forcing these methods to rely on synchronous API calls with an internal webclient that also can't be manipulated. When building a framework, use of properties is also preferred vs public fields (which also aren't immutable), as they make it much easier to avoid breaking changes to future versions of an API. Encapsulation is the key to most frameworks, and I would expose more of this implementation through the use of interfaces and allow some of these classes to be more friendly to inheritance to modify and/or extend existing behavior.

The one thing to keep in mind is that this API is meant to be RESTful. The representation of these classes should represent the equivalent of the resources and collections that the API was designed to expose.

I'm not sure how much of the API changed when I initially was going to write something for this, but the design itself didn't follow RESTful API guidelines too well, so I abandoned that thought. tbh i just wrote this in a very short period of time because i realized i could use the API now and i was bored. i may update it, but im definitely keeping the data deserialization to hard coded objects because it makes it much easier to install and use the API without reading any docs. i may add a function that returns a simple deserialized object instead, or raw json data, so the developer can handle it however they'd like in a scenario where the API updates and the wrapper is outdated. can you explain how using properties rather than public fields makes it easier to avoid breaking changes in future API versions? and the only function i really see benefit in overriding is GET. i also don't really see a single instance in the project in which an interface would be useful. i may replace web requests with an async solution though, not really sure yet, but mainly because i don't trust people to write thread safe code

edit: actually after reading how the api names some vars i might use JsonProperty attributes... the api is a bit inconsistent

0xDEAD10CC - May 12, 2019, 10:43 pm

(May 12, 2019 10:08 PM)JUSTIN Wrote: [\[CITE\]](#) tbh i just wrote this in a very short period of time because i realized i could use the API now and i was bored. i may update it, but im definitely keeping the data deserialization to hard coded objects because it makes it much easier to install and use the API without reading any docs. i may add a function that returns a simple deserialized object instead, or raw json data, so the developer can handle it however they'd like in a scenario where the API updates and the wrapper is outdated. can you explain how using properties rather than public fields makes it easier to avoid breaking changes in future API versions? and the only function i really see benefit in overriding is GET. i also don't really see a single instance in the project in which an interface would be useful. i may replace web requests with an async solution though, not really sure yet, but mainly because i don't trust people to write thread safe code

edit: actually after reading how the api names some vars i might use JsonProperty attributes... the api is a bit inconsistent

I'm not saying to not expose objects of the appropriate RESTful API types. This is actually quite contrarily what I was suggesting to do rather than relying on dynamic for extensibility/compatibility. With Newtonsoft, you're leaving a lot of its functionality on the table too. This framework does nothing more than serialize/deserialize objects in a very non-scalable way without any opportunity for others to expand or manipulate desired functionality. Additionally, parts of your framework have

XML docs, and there are some classes that neglect documentation entirely.

Good framework design is more difficult than most people can begin to realize... You don't want things tightly coupled, but you want a good enough design that will scale easily without introducing breaking changes to those that implement your framework in their custom solutions, and maintaining backwards compatibility as best as possible so that they don't have to re-write their entire program to use V2.

As for my comment about properties...

Properties are a combination of methods that allow you to expose fields in an indirect way. This means they allow you to expose that data in any way you need to, without changing the public interface. Fields should typically be kept hidden because there's no way to provide any kind of data validation on them either, unlike properties. The entire point of encapsulation most of the time is to allow for easier interchangeability, while maintaining the validity of an internal state; making things harder to use incorrectly and easy to use correctly. Properties are a layer of abstraction which allows you to do this behind the scenes while avoiding breaking changes if there's a case you need to manipulate the way this field is set or accessed.

>> and the only function i really see benefit in overriding is GET

Abstraction isn't solely based on the ability to override functions. There may be other use cases, but these are the kinds of things you need to think about when writing a library.

>> i also don't really see a single instance in the project in which an interface would be useful

Well, i'm not saying interfaces would be the best design choice here, but one example off the top of my head would be to represent request functionality. This would enable you to build up a pipeline of requests that can flow through some kind of processing class object for more automated tasks.

edit: Here's another post about properties: <https://csharpindepth.com/articles/PropertiesMatter>

Maintainable code is something that developers learn especially once they start writing larger projects. You can't really maintain code that was written to be just a collection of wrapper classes for functionality. If you ever need to scale out, it just means you have to re-write more and more of your existing functionality.

JUSTIN - May 13, 2019, 5:16 am

(May 12, 2019 10:43 PM)0xDEAD10CC Wrote: [\[CITE\]](#) ~ snip ~

i know what properties are and how they work, i just didn't really understand the benefit of using them in this scenario. i could set up the fields as properties with default getters/setters but since the data is being deserialized using reflection and not set from a developer stand point, i'm not sure how using properties would make it harder to screw up the code? and i've worked on large projects with maintainable and scale-able code, but it's a small open source project that i put together in an hour so i didn't really care for specifics regarding abstraction. i'm going to make some logical changes that will allow the code to be used on api funcs that don't return consistent output that can be deserialized & reflected to structs/that'll also allow the code to be usable if the api updates. thanks for the suggestions.

Dotexe - May 13, 2019, 5:21 am

Amazing release Justin, keep it up ?

JUSTIN - May 13, 2019, 5:28 am

(May 13, 2019 05:21 AM)Dotexe? Wrote: [\[CITE\]](#) Amazing release Justin, keep it up ?
Thanks for the positive feedback ?

edit: started adding async support in a new branch. haven't added the individual funcs yet but it's basically completed already.

file in branch: <https://github.com/ooojustin/HF-Sharp/blob/feat/async/HF-API.cs>

non-async version:

```
Code public UserInformation GetUserInfo(int uid) {  
    string path = "user/" + uid;  
    return GET<UserInformation>(path);  
}
```

async version:

```
Code public async Task<UserInformation> GetUserInfoAsync(int uid) {  
    string path = "user/" + uid;  
    return await GETAsync<UserInformation>(path);  
}
```

0xDEAD10CC - May 13, 2019, 10:03 am

(May 13, 2019 05:16 AM)JUSTIN Wrote: [\[CITE\]](#) i know what properties are and how they work, i just didn't really understand the benefit of using them in this scenario. i could set up the fields as properties with default getters/setters but since the data is being deserialized using reflection and not set from a developer stand point, i'm not sure how using properties would make it harder to screw up the code? and i've worked on large projects with maintainable and scale-able code, but it's a small open source project that i put together in an hour so i didn't really care for specifics regarding abstraction. i'm going to make some logical changes that will allow the code to be used on api funcs that don't return consistent output that can be deserialized & reflected to structs/that'll also allow the code to be usable if the api updates. thanks for the suggestions.

The data isn't being deserialized using reflection, i'm curious why you think that :/.

This is a library/wrapper around the HF API implementation though. If you read the links I provided it wasn't a statement about "making it harder to screw up the code", it was about maintainability and scalability for library design infrastructure. You may not care but if a developer is going to invest time into using this, then they may have a hard time supporting any of the changes you make to this library if they intend on supporting future revisions of your wrapper. That's what i'm getting at -- If you're not using properties and other levels of encapsulation, it just means that ANY change you make will break their code.

Not understanding the benefits of properties in projects like this is common for people that haven't had much experience with library design. If for instance, the HF API changes its requirements for a certain property in terms of the kind of data that is valid/invalid, you would be able to reflect this easily without any implications to the public interface; thus, the importance or advantage of the encapsulation properties provide. They allow you to protect your users from any changes to the API, while still allowing your interface to modify its core functionality to accommodate more API changes without incurring breaking changes to your library.

Think about it from an interface standpoint, because any public interface of any library has similar problem(s) in retrospect: If you implement an interface, what happens when that interface gets modified? You now have to go back and edit all of your code that implements this interface to make sure that it matches the desired code contract. That is a breaking change that is on a global scale, which may cause tons of rework depending on the size of the project or how many times that interface is used. Additionally, if the interface usage is deeply nested in your core code, perhaps this means changes all the way up your inheritance chain too.

JUSTIN - May 13, 2019, 10:43 am

(May 13, 2019 10:03 AM)0xDEAD10CC Wrote: [\[CITE\]](#) The data isn't being deserialized using reflection, I'm curious why you think that :/.

This is a library/wrapper around the HF API implementation though. If you read the links I provided it wasn't a statement about "making it harder to screw up the code", it was about maintainability and scalability for library design infrastructure. You may not care but if a developer is going to invest time into using this, then they may have a hard time supporting any of the changes you make to this library if they intend on supporting future revisions of your wrapper. That's what I'm getting at -- If you're not using properties and other levels of encapsulation, it just means that ANY change you make will break their code.

Not understanding the benefits of properties in projects like this is common for people that haven't had much experience with library design. If for instance, the HF API changes it's requirements for a certain property in terms of the kind of data that is valid/invalid, you would be able to reflect this easily without any implications to the public interface; thus, the importance or advantage of the encapsulation properties provide. They allow you to protect your users from any changes to the API, while still allowing your interface to modify its core functionality to accommodate more API changes without incurring breaking changes to your library.

Think about it from an interface standpoint, because any public interface of any library has similar problem(s) in retrospect: If you implement an interface, what happens when that interface gets modified? You now have to go back and edit all of your code that implements this interface to make sure that it matches the desired code contract. That is a breaking change that is on a global scale, which may cause tons of rework depending on the size of the project or how many times that interface is used. Additionally, if the interface usage is deeply nested in your core code, perhaps this means changes all the way up your inheritance chain too. I read somewhere that JavaScriptSerializer in .NET uses reflection so I assumed Newtonsoft.Json would too but never looked into it.

also regarding the point of using properties to support future updates to the API, that does make more sense. but can you provide a more specific example? even if the HF API "changes it's requirements for a certain property in terms of the kind of data that is valid/invalid" - properties still have a type. are you just referring to allowing people to change getter/setter methods to validate data? and why not just allow them to retrieve the deserialized object itself without converting it to pre-existing structs? it'd be great if you could link an example of a library that uses properties like this, unless you're simply suggesting I change the fields to properties with default get/set.

0xDEAD10CC - May 13, 2019, 10:56 am

(May 13, 2019 10:43 AM)JUSTIN Wrote: [\[CITE\]](#) I read somewhere that JavaScriptSerializer in .NET uses reflection so I assumed Newtonsoft.Json would too but never looked into it.

also regarding the point of using properties to support future updates to the API, that does make more sense. but can you provide a more specific example? even if the HF API "changes it's requirements for a certain property in terms of the kind of data that is valid/invalid" - properties still have a type. are you just referring to allowing people to change getter/setter methods to validate data? and why not just allow them to retrieve the deserialized object itself without converting it to pre-existing structs? it'd be great if you could link an example of a library that uses properties like this, unless you're simply suggesting I change the fields to properties with default get/set.

Sure - but the implementation for Newtonsoft JSON using reflection in a very sparingly way, in which is also fully transparent to the actual usage of this library. To illustrate that point - your code is not required to use reflection anywhere.

Changing a type within a type safe language is going to create a breaking change no matter what. You can however change your internal type if the API changes, and convert it to the type that you

expose via your public interface if possible too (and this would allow your library to maintain compatibility and avoid breaking changes). If you need to change validation for an int or provide validation for an API endpoint that didn't previously require validation, OR for instance you need to invoke or change a callback event for when a piece of information changes -- you can do all of this easily with properties.

>> and why not just allow them to retrieve the deserialized object itself without converting it to pre-existing structs

Well the point of a library is to abstract things and make it easier to use, so the functionality in most API's for the most part is hidden from the user. Libraries that expose raw data do this on the outside of the library, not as part of the core implementation, as a way to provide custom extensibility where needed. The goal for most libraries is to provide the majority (if not all) of the base functionality within the abstract implementation though.

The .NET framework is an example too - imagine not having access to anything but P/Invoke. P/Invoke is really just there in case you need to do something custom or you need to invoke a function in a version of Windows that isn't exposed by the framework through a wrapper class. Most of those functions aren't necessary in most cases though because there's a .NET alternative 99% of the time, and .NET is already essentially a Win32 wrapper on Windows, but allows you to write code even if you hardly know anything about Win32 API's.

For a reference, take a look at this:

<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/designing-for-extensibility>

Also look up the terms "tight coupling" and maybe IoC might be something worthwhile looking into as well. Framework design is mostly about creating building blocks in an interchangeable way.

Here's a link that better describes some of the things I was talking about:

<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/field>

Quote: The principle of encapsulation is one of the most important notions in object-oriented design. This principle states that data stored inside an object should be accessible only to that object.

A useful way to interpret the principle is to say that a type should be designed so that changes to fields of that type (name or type changes) can be made without breaking code other than for members of the type. This interpretation immediately implies that all fields must be private.

JUSTIN - May 13, 2019, 2:32 pm

(May 13, 2019 10:56 AM)0xDEAD10CC Wrote: [\[CITE\]](#) Changing a type within a type safe language is going to create a breaking change no matter what. You can however change your internal type if the API changes, and convert it to the type that you expose via your public interface if possible too (and this would allow your library to maintain compatibility and avoid breaking changes). If you need to change validation for an int or provide validation for an API endpoint that didn't previously require validation, OR for instance you need to invoke or change a callback event for when a piece of information changes -- you can do all of this easily with properties.

For a reference, take a look at this:

<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/designing-for-extensibility>

Also look up the terms "tight coupling" and maybe IoC might be something worthwhile looking into as well. Framework design is mostly about creating building blocks in an interchangeable way.

Here's a link that better describes some of the things I was talking about:

<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/field>

i understand the concept of encapsulation, and i've used properties before. i was just trying to understand why it's so important for longevity of an api wrapper like this. from what i've read, it seems to be extra important when writing code for meant to interact with some data source that the developer doesn't always have control of, and i guess an API is a perfect example of this. in the past i've only used properties over fields in situations where data validation is necessary and i think i understand why thats so important in library design too now. thanks for the advice, i'll release a version tomorrow favoring properties over public fields and support for asynchronous requests.

edit: does this look a bit better?

<https://github.com/ooojustin/HF-Sharp/blob/784104180194de24fe03a16287f36b81f75a841a/Serialized/UserInformation.cs>

i've opted to use the JsonProperty attribute so i can follow standard naming conventions and make the property names make a bit more sense in scenarios where they're poorly named by the api or automatically handled differently by the content of the getter/setter logic

+Haze - May 14, 2019, 11:10 am

(May 13, 2019 02:32 PM)JUSTIN Wrote: [\[CITE\]](#) i understand the concept of encapsulation, and i've used properties before. i was just trying to understand why it's so important for longevity of an api wrapper like this. from what i've read, it seems to be extra important when writing code for meant to interact with some data source that the developer doesn't always have control of, and i guess an API is a perfect example of this. in the past i've only used properties over fields in situations where data validation is necessary and i think i understand why thats so important in library design too now. thanks for the advice, i'll release a version tomorrow favoring properties over public fields and support for asynchronous requests.

edit: does this look a bit better?

<https://github.com/ooojustin/HF-Sharp/blob/784104180194de24fe03a16287f36b81f75a841a/Serialized/UserInformation.cs>

i've opted to use the JsonProperty attribute so i can follow standard naming conventions and make the property names make a bit more sense in scenarios where they're poorly named by the api or automatically handled differently by the content of the getter/setter logic

Yes, you've got the right idea. Your code could be simplified a bit more like this

<https://pastebin.com/RMkmiUmi>

You can use the shorthand auto property syntax because you're not really doing any processing or logic with the get and set methods. The only time you'd explicitly declare the get and set methods would be when you need to handle some logic when a property has been accessed or changed. Getters and setters were something more prevalent in Java due to its verbose syntax, but C# has a more syntactically less verbose option which is properties. Behind the scenes, auto properties still generate the full get and set methods, but the nicer syntax allows for cleaner code.

In regards to the JsonProperty, it's not all that necessary to explicitly put the JsonProperty attribute on all properties because Json.Net will recognize the ones that have a matching name, but ones with different names do have to use the JsonProperty attribute.

JUSTIN - May 14, 2019, 4:04 pm

(May 14, 2019 11:10 AM)+Haze Wrote: [\[CITE\]](#) Yes, you've got the right idea. Your code could be simplified a bit more like this

<https://pastebin.com/RMkmiUmi>

i chose to include the internal values now so it'll be easier if i need to add logic to the getters or

setters in the future.

are u just suggesting i simply exclude the private values in favor of using public properties with auto get/set until i actually need to add some type of logic to the getters/setters? i don't really see a point. if i do need to add logic to any individual property the code will become inconsistent until i add private values to all of them.

+Haze - May 14, 2019, 4:11 pm

(May 14, 2019 04:04 PM)JUSTIN Wrote: [\[CITE\]](#) using that example, in which there is no independent internal value, somewhat defeats the purpose in this scenario. i'm pretty sure the reason for having internal values stored separately is so in a situation in which the API itself is updated and you need to make the library adapt to those changes, you can do so without necessarily breaking projects dependent on the API wrapper. with separate internal values, you can also use validate values before storing them and call events among other things. info here:

<https://stackoverflow.com/a/29938136/5699643>

i chose to include the internal values now so it'll be easier if i need to add logic to the getters or setters in the future.

unless ur suggesting i simply exclude the private values in favor of using autos until i actually need to add some type of logic to the getters/setters?

I am aware of why it's done, I am just saying in your case you're only deserializing the JSON data from the API to the object, so 9/10 times you won't need to have custom logic or events when a value is changed. That class is only to represent the JSON data in a strongly typed way, that is all. In all my life of deserializing JSON data to an object, the need to create full get and setters has never appeared, auto properties have always been fine.

JUSTIN - May 14, 2019, 4:28 pm

(May 14, 2019 04:11 PM)+Haze Wrote: [\[CITE\]](#) I am aware of why it's done, I am just saying in your case you're only deserializing the JSON data from the API to the object, so 9/10 times you won't need to have custom logic or events when a value is changed. That class is only to represent the JSON data in a strongly typed way, that is all. In all my life of deserializing JSON data to an object, the need to create full get and setters has never appeared, auto properties have always been fine.

yeah i updated my post after realizing the point u were making. im just doing this in preparation, and it doesn't really add much code. also other than changing access modifier, is there any benefit to using a public property over a public field?

+Haze - May 14, 2019, 4:41 pm

(May 14, 2019 04:28 PM)JUSTIN Wrote: [\[CITE\]](#) yeah i updated my post after realizing the point u were making. im just doing this in preparation, and it doesn't really add much code. also other than changing access modifier, is there any benefit to using a public property over a public field?

Yes. A public property can have getters and setters whilst a public field cannot. Also, in all .NET libraries I've seen, properties are almost always used. Field variables are normally only used within the scope of the class.

It may seem pointless to use a property over a public field when you don't need to have the full {get; set;} syntax, but it all derives from the olden days when getters and setters were a huge part of object-oriented programming. C# has evolved since then and has provided a more succinct syntax.

JUSTIN - May 14, 2019, 5:35 pm

(May 14, 2019 04:41 PM)+Haze Wrote: [\[CITE\]](#) Yes. A public property can have getters and setters whilst a public field cannot. Also, in all .NET libraries I've seen, properties are almost always used. Field variables are normally only used within the scope of the class.

It may seem pointless to use a property over a public field when you don't need to have the full {get; set;} syntax, but it all derives from the olden days when getters and setters were a huge part of object-oriented programming. C# has evolved since then and has provided a more succinct syntax.

Well what I really meant is "also other than changing access modifier, is there any benefit to using a public property with auto get/set over a public field". Since I've read it's bad practice, I won't use them in my API wrappers from now on. I've written code with properties after I gravitated away from creating my own functions for getting/setting variables (like in Java) due to how verbose it is. In my own projects I typically use public fields in scenarios where the get/set won't be used. Anyway you're definitely right about .NET libs using properties, I've noticed that tendency. According to this post on StackOverflow (

<https://stackoverflow.com/questions/1180860/public-fields-versus-automatic-properties>) there are some benefits other than simply being able to easily implement get/set. I'll be sure to use them in my code more frequently from now on.

0xDEAD10CC - May 14, 2019, 6:44 pm

(May 14, 2019 11:10 AM)+Haze Wrote: [\[CITE\]](#) Yes, you've got the right idea. Your code could be simplified a bit more like this

<https://pastebin.com/RMkmiUmi>

You can use the shorthand auto property syntax because you're not really doing any processing or logic with the get and set methods. The only time you'd explicitly declare the get and set methods would be when you need to handle some logic when a property has been accessed or changed. Getters and setters were something more prevalent in Java due to its verbose syntax, but C# has a more syntactically less verbose option which is properties. Behind the scenes, auto properties still generate the full get and set methods, but the nicer syntax allows for cleaner code.

In regards to the JsonProperty, it's not all that necessary to explicitly put the JsonProperty attribute on all properties because Json.Net will recognize the ones that have a matching name, but ones with different names do have to use the JsonProperty attribute.

For naming consistency I would still apply JsonPropertyAttribute to all of the properties. If you don't, you'll have some lowercase and the rest will match the property names, which in this case are PascalCase and not all lowercase. Additionally, if you ever decided to change a property name, this won't break the serialization/deserialization naming when your identifier changes.

@OP: Going from auto-properties to fully defined ones is not a breaking change. Modifying the accessibility or including/excluding an accessor or mutator is however. This is why using properties is preferred in any kind of situation like this. They can (for the most part) behave exactly like your public field, but it's a lot more scalable if you need to use the getter or setter to do things in the future, and your users won't be mad that they have to re-write a bunch of code because doing so won't change anything on their end. When you get into writing more complex frameworks this is far more important to understand, especially if you're including properties as part of an interface.

This topic extends further than just framework design, even in your own code you can control immutability too using properties to prevent yourself from using your own classes incorrectly. Protected or private setters, or no setter at all, for properties that return a private readonly field that is set within a constructor -- just as one example. As another example, say you need to return a value depending on a certain condition... With a property, you can use the getter to conditionally branch that out without editing the field, and without having to write inline code everywhere you use that field to conditionally branch to return the special case value. They have all the same benefits that you could apply to a method, but in the context of specific pieces of information.

JUSTIN - May 14, 2019, 11:49 pm

(May 14, 2019 06:44 PM)0xDEAD10CC Wrote: [\[CITE\]](#) @OP: Going from auto-properties to fully

defined ones is not a breaking change. Modifying the accessibility or including/excluding an accessor or mutator is however. This is why using properties is preferred in any kind of situation like this. They can (for the most part) behave exactly like your public field, but it's a lot more scalable if you need to use the getter or setter to do things in the future, and your users won't be mad that they have to re-write a bunch of code because doing so won't change anything on their end. When you get into writing more complex frameworks this is far more important to understand, especially if you're including properties as part of an interface.

i'm aware of that, that's pretty much what i said. i stated that i included the private fields to maintain consistency if i need to add logic to a getter/setter in the future. it doesnt make the code any more complicated as is and its just slightly lmore verbose for the time being.

(May 14, 2019 06:44 PM)0xDEAD10CC Wrote: [\[CITE\]](#) This topic extends further than just framework design, even in your own code you can control immutability too using properties to prevent yourself from using your own classes incorrectly. Protected or private setters, or no setter at all, for properties that return a private readonly field that is set within a constructor -- just as one example. As another example, say you need to return a value depending on a certain condition... With a property, you can use the getter to conditionally branch that out without editing the field, and without having to write inline code everywhere you use that field to conditionally branch to return the special case value. They have all the same benefits that you could apply to a method, but in the context of specific pieces of information.

well yes i've used properties for these things, just not in the context of anticipating changes for something like a library where (without properties) a change to an API can break your project alongside those of everybody else using your library

0xDEAD10CC - May 15, 2019, 3:43 pm

(May 14, 2019 11:49 PM)JUSTIN Wrote: [\[CITE\]](#) i'm aware of that, that's pretty much what i said. i stated that i included the private fields to maintain consistency if i need to add logic to a getter/setter in the future. it doesnt make the code any more complicated as is and its just slightly lmore verbose for the time being.

well yes i've used properties for these things, just not in the context of anticipating changes for something like a library where (without properties) a change to an API can break your project alongside those of everybody else using your library

What you said is not what I said at all. I stated that developers don't worry about consistency when it comes to automatic properties vs the more verbose but semantically identical equivalent--where you as the programmer manually define the backing field. Thus, there's essentially no discernible difference between these two snippets of code:

```
Code public Type PropertyName { get; set; }
Code private Type _propertyName;
public Type PropertyName {
    get { return _propertyName; }
    set { _propertyName = value; }
}
```

Depending on your code formatting preference, you can be wasting up to almost 10 lines of code for something that should really only be one. Evidently, people care about succinct code simplifications like these too, otherwise Microsoft wouldn't have created more features to compact property assignment all within the same line (among other things) in recent C# language versions.

Subsequently, there's no implications in terms of breaking changes when switching between the two, AND, with automatic properties, another important point is that you're less likely to interchangeably use the field and property within your own code which complicates the logic flow. So why are you worrying about that in your code? ? You're only unknowingly needlessly over-complicating what can be simplified. When we do peer code reviews at my company we look for all these things. This is why people use automatic properties over fields, because they can go

from a { get; set; } which is hardly different from the implementation of a field, however switching to a custom getter or setter doesn't break anything for anyone implementing the framework.

With that clarified - I'm not sure why you even have private backing fields at all if you're not using it for anything more than what an automatic property would already do for you.

You also only updated one class but you left the rest of them using fields. :/ And your HF_API class has IDisposable members so it should therefore implement the IDisposable interface to reflect the appropriate disposable pattern for cleanup.

JUSTIN - May 15, 2019, 5:49 pm

(May 14, 2019 06:44 PM)0xDEAD10CC Wrote: [\[CITE\]](#) @OP: Going from auto-properties to fully defined ones is not a breaking change. Modifying the accessibility or including/excluding an accessor or mutator is however. This is why using properties is preferred in any kind of situation like this. They can (for the most part) behave exactly like your public field, but it's a lot more scalable if you need to use the getter or setter to do things in the future, and your users won't be mad that they have to re-write a bunch of code because doing so won't change anything on their end. When you get into writing more complex frameworks this is far more important to understand, especially if you're including properties as part of an interface.

JUSTIN pid=" dateline=" Wrote:i'm aware of that, that's pretty much what i said. i stated that i included the private fields to maintain consistency if i need to add logic to a getter/setter in the future. it doesnt make the code any more complicated as is and its just slightly lmore verbose for the time being.

(May 15, 2019 03:43 PM)0xDEAD10CC Wrote: [\[CITE\]](#) What you said is not what I said at all. I stated that developers don't worry about consistency when it comes to automatic properties vs the more verbose but semantically identical equivalent--where you as the programmer manually define the backing field.I think you misunderstood what Haze was talking about in that first post. We weren't talking about going from auto-properties to fully defined ones being breaking or not, he was just pointing out that unless I'm actually adding logic to get/set there's no point of not using the autos.

(May 15, 2019 03:43 PM)0xDEAD10CC Wrote: [\[CITE\]](#) With that clarified - I'm not sure why you even have private backing fields at all if you're not using it for anything more than what an automatic property would already do for you.This is exactly what Haze was saying. I'll switch to using the autos for now until I have reason to use the get/set funcs.

(May 15, 2019 03:43 PM)0xDEAD10CC Wrote: [\[CITE\]](#) You also only updated one class but you left the rest of them using fieldswell yeah, it's not the master branch. i'll turn the rest into properties and merge to master before updating the nuget package.

(May 15, 2019 03:43 PM)0xDEAD10CC Wrote: [\[CITE\]](#) And your HF_API class has IDisposable members so it should therefore implement the IDisposable interface to reflect the appropriate disposable pattern for cleanup.good point, I'll add that

0xDEAD10CC - May 15, 2019, 5:54 pm

(May 15, 2019 05:49 PM)JUSTIN Wrote: [\[CITE\]](#) he was just pointing out that unless I'm actually adding logic to get/set there's no point of not using the autos.

Well then that's what I mean, yes. :)

(May 15, 2019 05:49 PM)JUSTIN Wrote: [\[CITE\]](#) But then again, even when switching from public fields to public properties with get/set, this wouldn't be a breaking change unless the property name was changed.

If you developed a more complex framework this wouldn't hold true. For instance, if you had

interfaces that needed to use the properties/fields in question. You can't substitute one for the other without the interface implementation changing.

In your case however, this may be true, as long as people aren't using reflection on your fields for whatever reason. (Properties can't be treated the same way as normal fields in reflection.)

If you want your wrapper to be the best framework available out there though for now, you still have time to make corrections and improvements.

JUSTIN - May 15, 2019, 5:58 pm

(May 15, 2019 05:54 PM)0xDEAD10CC Wrote: [\[CITE\]](#) In your case however, this may be true, as long as people aren't using reflection on your fields for whatever reason. (Properties can't be treated the same way as normal fields in reflection.)

ok well

- 1.) properties have diff membetypes during reflection
- 2.) data binding on properties
- 3.) out/ref args on fields
- 4.) access modifiers

aside from these, are there any other differences? just asking out of curiosity, even though it's a little out of context of this thread now

0xDEAD10CC - May 15, 2019, 6:35 pm

Within the context of library/framework design, all you need to think about is encapsulation for the most part and in a lot of cases you can avoid problems if your skillful at appropriately implementing those concepts.

JUSTIN - May 15, 2019, 6:41 pm

(May 15, 2019 06:35 PM)0xDEAD10CC Wrote: [\[CITE\]](#) Within the context of library/framework design, all you need to think about is encapsulation for the most part and in a lot of cases you can avoid problems if your skillful at appropriately implementing those concepts.

I was just asking in general but yeah auto public properties should be good for now

JUSTIN - May 19, 2019, 7:46 am

Just released version 1.2.0 using properties.

Also included a feature allowing you to bypass the captcha system that's enabled on the API (for some reason), so you can run your programs on a VPS/VPN. More information about that feature here: <https://hackforums.net/showthread.php?tid=5970613&pid=58590656#pid58590656>

incelsubhuman - June 7, 2019, 12:03 pm

(May 14, 2019 06:44 PM)0xDEAD10CC Wrote: [\[CITE\]](#) For naming consistency I would still apply JsonPropertyAttribute to all of the properties. If you don't, you'll have some lowercase and the rest will match the property names, which in this case are PascalCase and not all lowercase.

Additionally, if you ever decided to change a property name, this won't break the serialization/deserialization naming when your identifier changes.

@OP: Going from auto-properties to fully defined ones is not a breaking change. Modifying the accessibility or including/excluding an accessor or mutator is however. This is why using properties is preferred in any kind of situation like this. They can (for the most part) behave exactly like your public field, but it's a lot more scalable if you need to use the getter or setter to do things in the future, and your users won't be mad that they have to re-write a bunch of code because doing so won't change anything on their end. When you get into writing more complex frameworks this is far more important to understand, especially if you're including properties as part of an interface.

This topic extends further than just framework design, even in your own code you can control immutability too using properties to prevent yourself from using your own classes incorrectly. Protected or private setters, or no setter at all, for properties that return a private readonly field that is set within a constructor -- just as one example. As another example, say you need to return a value depending on a certain condition... With a property, you can use the getter to conditionally branch that out without editing the field, and without having to write inline code everywhere you use that field to conditionally branch to return the special case value. They have all the same benefits that you could apply to a method, but in the context of specific pieces of information.

Teaching basic SE principles to beginners in the .NET section is a waste of your time

JUSTIN - June 8, 2019, 7:40 am

(Jun 7, 2019 12:03 PM)incelsubhuman Wrote: [\[CITE\]](#) Teaching basic SE principles to beginners in the .NET section is a waste of your time

This is a matter of library design, not SE principles, and I'm not a beginner. When it comes down to it, normal variables can be changed to properties whenever logic needs to be implemented and that wont be a breaking change unless the project using your library implemented one of the concepts I mentioned in this post:

<https://hackforums.net/showthread.php?tid=5969375&pid=58579244#pid58579244>

doxerhacker - June 8, 2019, 7:39 pm

(Jun 8, 2019 07:40 AM)JUSTIN Wrote: [\[CITE\]](#) This is a matter of library design, not SE principles, and I'm not a beginner. When it comes down to it, normal variables can be changed to properties whenever logic needs to be implemented and that wont be a breaking change unless the project using your library implemented one of the concepts I mentioned in this post:

SE principles -> SOLID