Best Practices

Introduction

As in all things, you are free to choose your own policies; however, here are some suggested best practices for various development tasks in FireBreath. This is a work in progress; if you have items that should go on this list, please add them. We'll remove them if we don't like them =] Feel free to leave comments if you disagree with something on this list.

Submitting feature requests or issue reports

- Don't complain about something unless you have a constructive suggestion.
    - For example, "CMake is awful and shouldn't be used!" is only helpful if you understand why we are using it and have a better suggestion
    - Similarly "Boost is big and shouldn't be used!" is only helpful if you understand where and why we use boost and have a better suggestion
- Don't just assume we'll fix your bug; if you have a problem with FireBreath please tell us about it! Most serious issues are fixed within a very short period of time after being reported, but we may not have encountered your issue yet.
- Patches are always welcome! Pull requests even more so!

Dealing with JSAPI objects

- Never use a JSAPI object by reference or by raw pointer; always use it wrapped in a boost::shared_ptr or boost::weak_ptr
    - If this feels like a lot of typing to you, consider adopting the FireBreath convention of typedef boost::shared_ptr<ClassName> ClassNamePtr and boost::weak_ptr<ClassName> ClassNameWeakPtr;
- Never store a shared_ptr to a JSAPI object except inside a class that owns that object. Otherwise, use a boost::weak_ptr. This will help prevent memory leaks due to cyclic dependencies
- Avoid using multiple inheritance on a JSAPI object! Consider using a "has a" instead of an "is a" relationship.
- Unless you are an exception to this rule (you'll know), always pass the shared_ptr const and by reference: void setPtr(const boost::shared_ptr<MyAPI>& api) { ... }

Dynamic typing (FB::variant) and you

The key thing to remember here is that premature optimization is the root of all evil, but premature pessimization should be avoided as well!

- Avoid repeated calls to variant::cast and particularly variant::convert_cast, as these calls have a cost; save the output of the first call and reuse it.

```
// Wrong!
if (val.convert_cast<FB::JSObjectPtr>()->HasProperty("left") |
    val.convert_cast<FB::JSObjectPtr>()->HasProperty("right")) { ... }
// Right!
FB::JSObjectPtr ptr(val.convert_cast<FB::JSObjectPtr>());
if (ptr->HasProperty("left") | ptr->HasProperty("right")) { ... }
```

- Don't use variant::cast unless you absolutely only want to allow one input type! If you just want everything to be a given type, for example a string, use variant::convert_cast<std::string>() which will convert numeric types to string! Similarly, variant::convert_cast<long>() will convert a string "23" to (long)23

```
// Wrong!
int out(-1);
if (val.is_of_type<int>()) out == val.cast<int>();
else if (val.is_of_type<long>()) out == val.cast<long>()) out = val.cast<long>();
// else if (...) etc

// Right!
try {
    int out(val.convert_cast<int>());
} catch (const FB::bad_variant_cast &ex) {
    // Oh, no! Couldn't convert it to an int
    // We better donate to FireBreath so they can change the laws of physics!
}
```

- Whenever possible, use a `const FB::variant&` for passing variants into a function; this accomplishes two things:
    1. You won't accidently change the variant thinking that will do something useful
    2. Passing by reference, you won't duplicate the variant (and thus the data inside it). With some data types this can be expensive.

Plugin project management - files, dependencies, targets, build machines

- Don't add files, dependencies, or link libraries to your project from inside your IDE
    - All project management should be done inside the CMake files
    - "But wait!" many cry, "Our build server doesn't have CMake!" Regardless of your reason for not liking CMake, get over it and just install it and use it as intended. It's up to you -- but please don't come complaining to us when you find that you have caused yourself far more work than you saved by not learning to use CMake.
- Don't move your build directory. Delete it and run the prep script again.
- Don't put your build directory in source control. Run the prep script on each computer, each platform