Integrating Haxe

In the Unity3D + node.js stack

About me

Dan Korostelev

@nadako

- Mobile game developer at Plamee
- Contributor to various stuff in Haxe

Haxe as a part of technology stack

- Game client: Unity3D and C#
- Game server: node.js and JavaScript
- Game logic (that runs on client AND server): ???

Haxe as a part of technology stack

- Game client: Unity3D and C#
- Game server: node.js and JavaScript
- Game logic (that runs on client AND server): HAXE

Alternatives

- JavaScript on client: embedding it in a Unity3D game is hard and requires a lot of boilerplate code, also JavaScript sucks (we actually tried that before).
- C# on the server:
    - Write everything in C# - could be a solution, but not for our team (our server guys are good with node.js, and making them learn .NET server infrastructure would be risky and impractical).
    - Running .NET within node.js (e.g. ember.js) - same concerns as embedding JS in client - the complexity raises and becomes risky.
    - Cross-compile C# to .js - every cross-compiler I checked out back then produced very heavyweight code and it was very hard to reason about its performance and debugability (is that a word?)

Haxe fits just right in

- Modern, strictly typed and easy-to-learn language
- Clean and optimized JS code for node.js server
- Clean and optimized (slightly worse than JS, but still very good) code for Unity3D client
- No hand-handwritten glue code and easy interop thanks to macros

Make it easy for skeptics and non-haxers

Integrate Haxe into the project seamlessly, making it as easy to use as possible and don’t give skeptical people more grounds for arguing than they already have.

- haxe compiler included in the project repo (no installation needed, easy updating!)
- easy-to use build scripts (single command to build everything)
- button that builds everything in a Unity editor (for non-programmers)

31
Chilistas
Battle ends in:
3m 32sec
Retreat
Not enough energy!
Not enough energy!
x4
x6
x16
13
19

What went good: cross-compilation

- main reason to use Haxe in our case
- readable generated code (useful in some rare cases)
- good performance (still some unnecessary overhead in C# target)

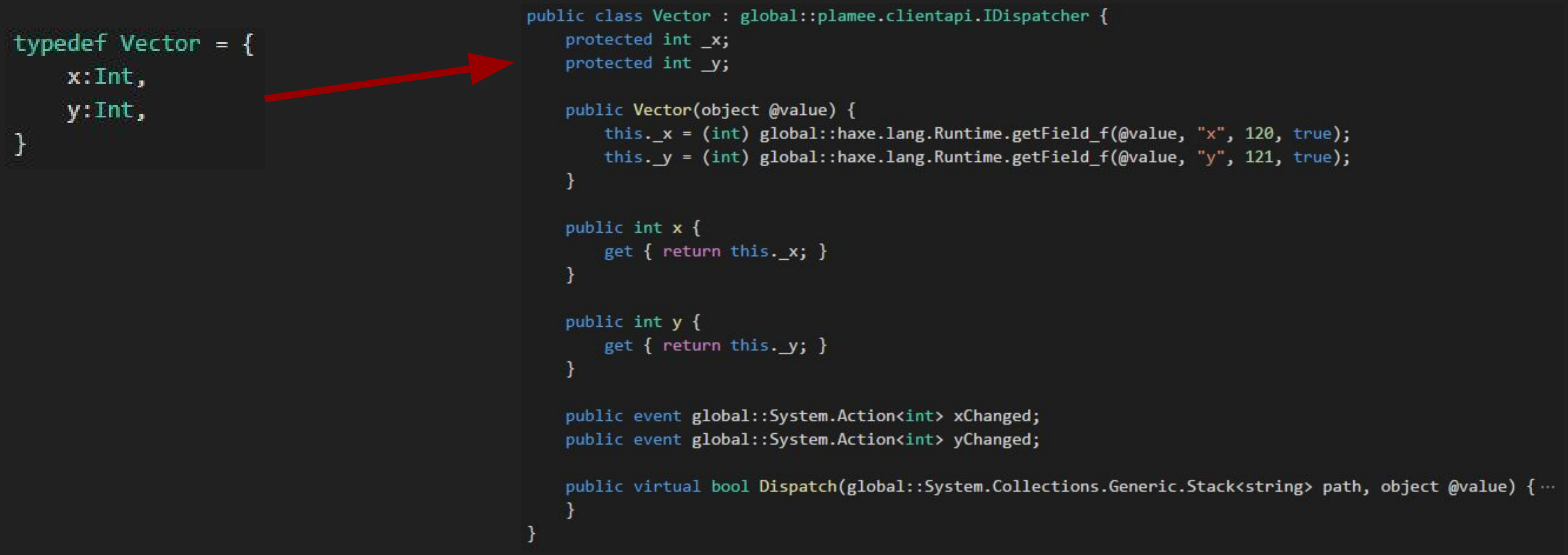
What went good: macros in the logic code

Usage of macros allows gaining extra safety, conciseness and performance:

- Working with JSON data:
  - validating and determining value type from path expressions. E.g.: `storage.set(user.name,value)` expanding to `storage._set([“user”,“name”],(value:String))`
  - generating read-only types (to avoid object copying)
- generating boilerplate code for RPC (similar to haxe.remote)

What went good: macros for C# glue code

Our game logic code manipulates typedef'd JSON structures. But we also need to display some of them in the client, so we auto-generate nice C# API

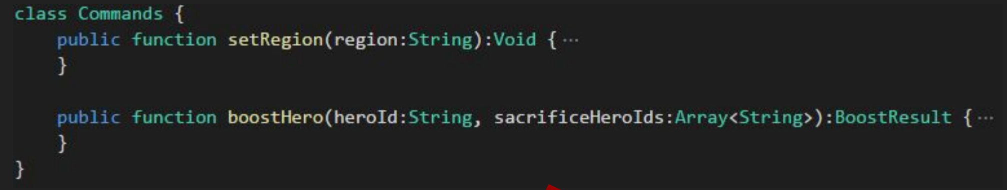


```haxe
typedef Vector = {
    x:Int,
    y:Int,
}
```

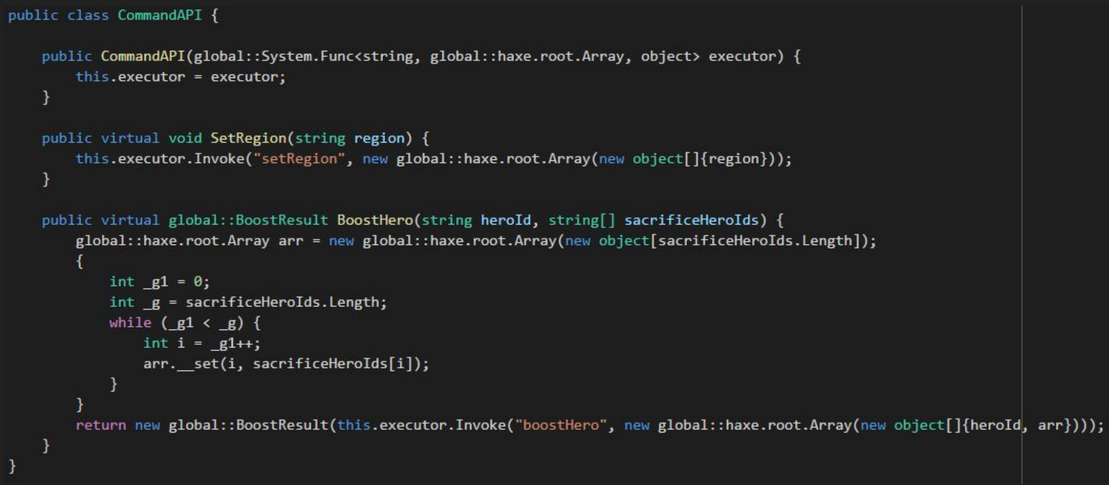
```csharp
public class Vector : global::plamee.clientapi.IDispatcher {
    protected int _x;
    protected int _y;

    public Vector(object @value) {
        this._x = (int) global::haxe.lang.Runtime.getField_f(@value, "x", 120, true);
        this._y = (int) global::haxe.lang.Runtime.getField_f(@value, "y", 121, true);
    }

    public int x {
        get { return this._x; }
    }

    public int y {
        get { return this._y; }
    }

    public event global::System.Action<int> xChanged;
    public event global::System.Action<int> yChanged;

    public virtual bool Dispatch(global::System.Collections.Generic.Stack<string> path, object @value) {⋯
    }
}
```

What went good: macros for C# glue code

We also auto-generate proxies for calling game logic methods from C# so that it feels natural to use from the C# side:



```
class Commands {
    public function setRegion(region:String):Void {…
    }

    public function boostHero(heroId:String, sacrificeHeroIds:Array<String>):BoostResult {…
    }
}
```



```
public class CommandAPI {

    public CommandAPI(global::System.Func<string, global::haxe.root.Array, object> executor) {
        this.executor = executor;
    }

    public virtual void SetRegion(string region) {
        this.executor.Invoke("setRegion", new global::haxe.root.Array(new object[]{region}));
    }

    public virtual global::BoostResult BoostHero(string heroId, string[] sacrificeHeroIds) {
        global::haxe.root.Array arr = new global::haxe.root.Array(new object[sacrificeHeroIds.Length]);
        {
            int _g1 = 0;
            int _g = sacrificeHeroIds.Length;
            while (_g1 < _g) {
                int i = _g1++;
                arr.__set(i, sacrificeHeroIds[i]);
            }
        }
        return new global::BoostResult(this.executor.Invoke("boostHero", new global::haxe.root.Array(new object[]{heroId, arr})));
    }
}
```

What went good: macros for validating static data

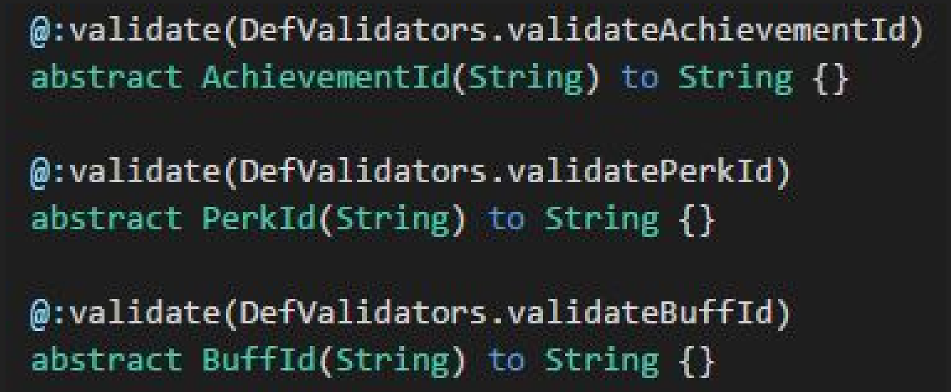
- define types in the logic code (also used by the code itself)
- macro-generate validation schema from those types
- validate JSON files against that schema
- PROFIT! (no “…”)

So, we only define types once and then use them for different purposes, following the DRY principle. Thank you Haxe!

PS: one could also generate fancy forms from those schemas

What went good: abstracts instead of primitives

- e.g. `MapItemId` instead of a `Int`, `ConsumableId` instead of a `String`
- makes code easier to understand
- adds more compile-time safety (because types are more specific!)
- brings more validation power (e.g. we implement “foreign key”-like checks)



```
@:validate(DefValidators.validateAchievementId)
abstract AchievementId(String) to String {}

@:validate(DefValidators.validatePerkId)
abstract PerkId(String) to String {}

@:validate(DefValidators.validateBuffId)
abstract BuffId(String) to String {}
```

What went good: abstracts in general

- wrapping raw JSON data in abstract types
- @:enum abstracts are great for defined JSON values
- zero-overhead null-safety

Things that are not so good:

By our experience, these things are often giving headaches:

- Inconsistent default values
- Inconsistent integer overflows

Not related to language, but still concerning:

- IDE (especially for C# people, spoiled by Visual Studio and ReSharper).
- Explaining the reasons of Haxe usage to newcomers can become annoying

Common questions from C#-ers

- Haxe doesn’t have method overloading?
- How do I pass a function (and then “wtf is this syntax”?:))
- Are there short lambdas? :-P

Questions?