Quest Zip

(namespace Progression)

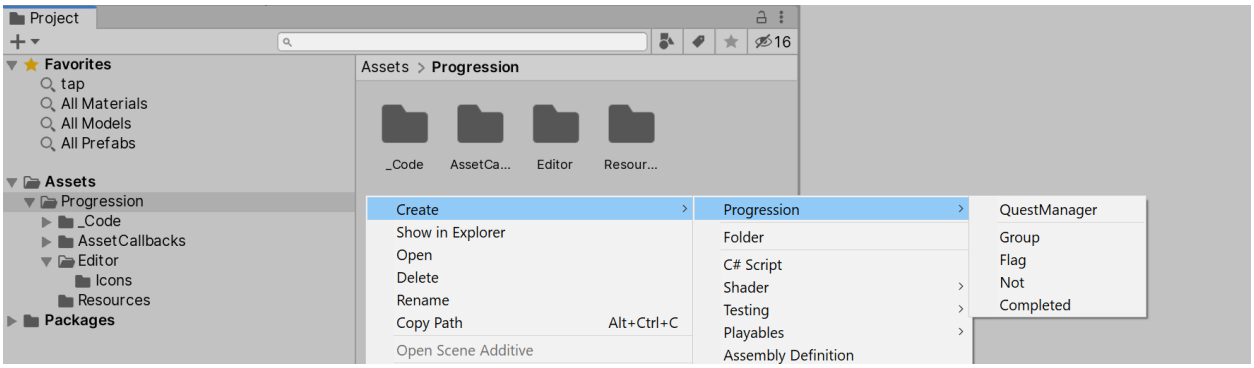
Introduction

Quest zip is an introductory framework to serializing player progress in a game like way. The core of this system is the quest management system. It helps organize player tasks with quests, progression points, and dependencies (think dungeon keys).

Getting Started

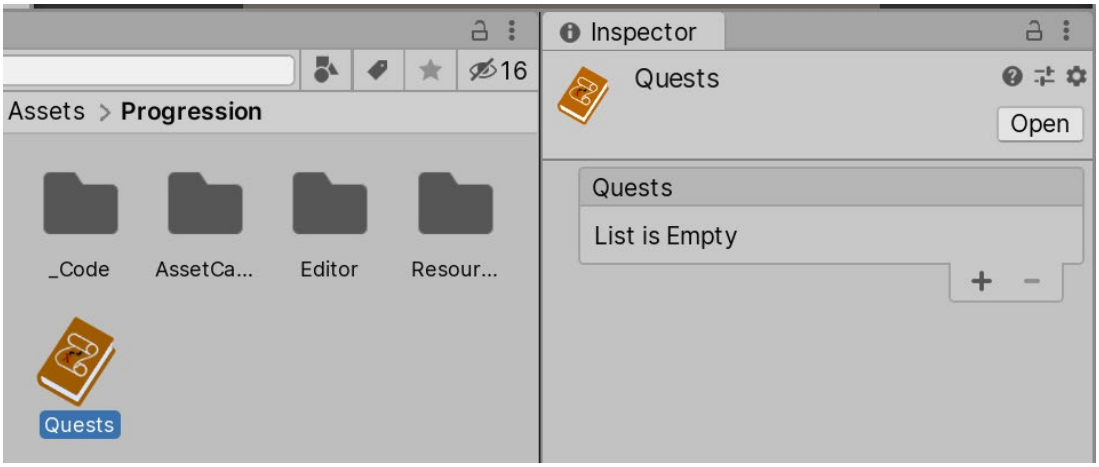
Download Quest Zip from the asset store and add it to your project folder (feel free to place anywhere in there, there are no location dependencies).

After it is imported, right click in an empty space of the Project window and take note of the new create menu options under (Progression->)



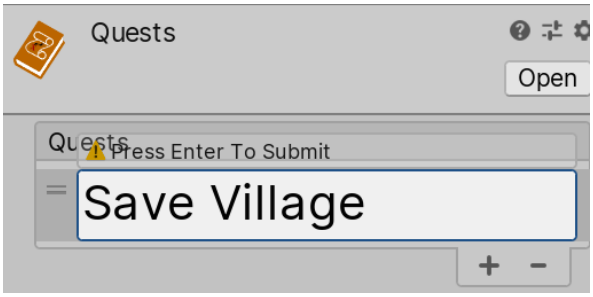
Click QuestManager.

You should now have a **quests manager** asset in your project that looks like this.

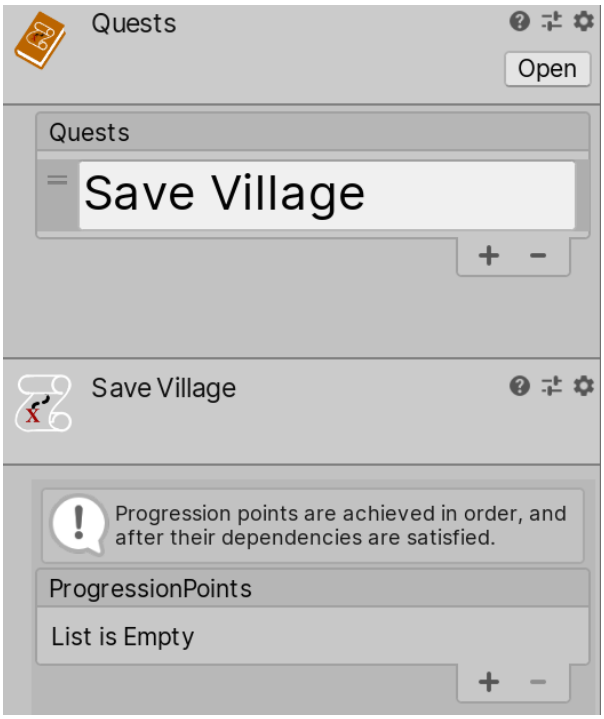


The quests manager is used to organize a group of related quests. This will be particularly useful when saving and loading progression in quests.

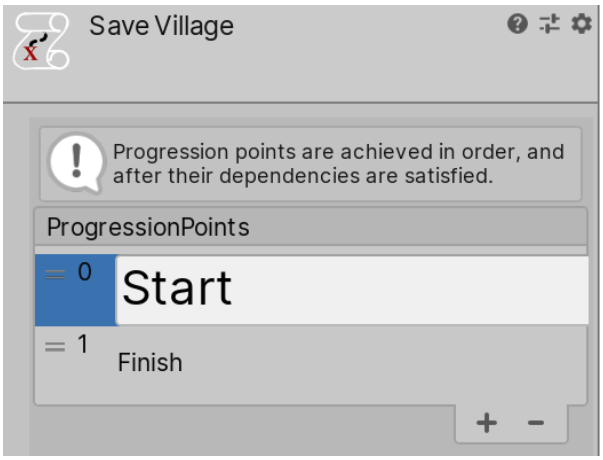
If you click the + button you will create a new **quest** asset. A quest is an ordered series of **progression points**, where a progression point represents one distinct piece of progress. A quest is completed when all of its progression points are completed.



Click in the text box, give the quest a name and hit enter. This will open a view of the quest below.



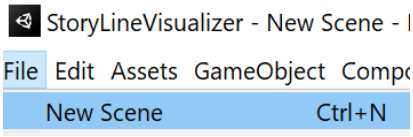
The quest has a collection of progression points. The most basic components of any quest that you should add are a “start” and “finish” progression point. You will notice a number to the left of the progression points in the list. These denote the order the player has to complete these progression points.



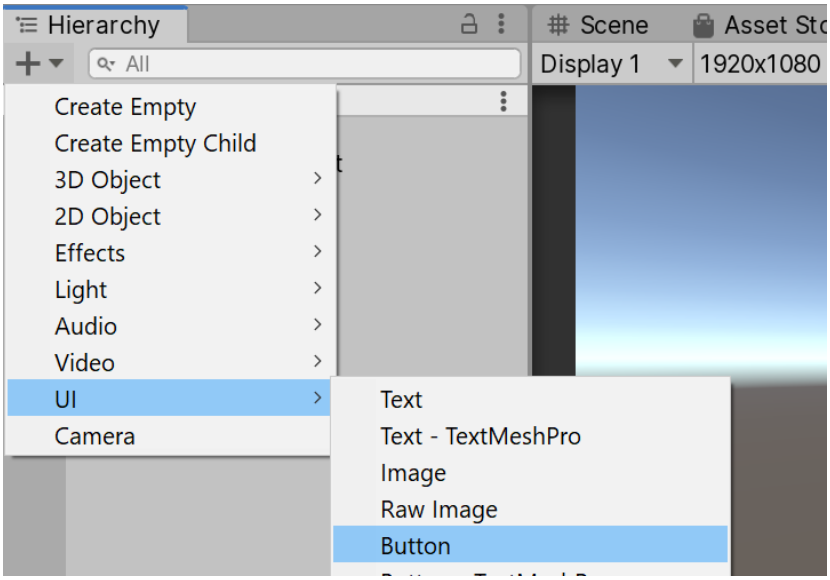
*To change the order simply click and drag on the handle to the left of the number.

At this point it may seem a little unclear how this is going to work in with any of the gameplay. This next part will demo it with a simple UI button. Feel free to skim this next section paying attention to the highlighted areas.

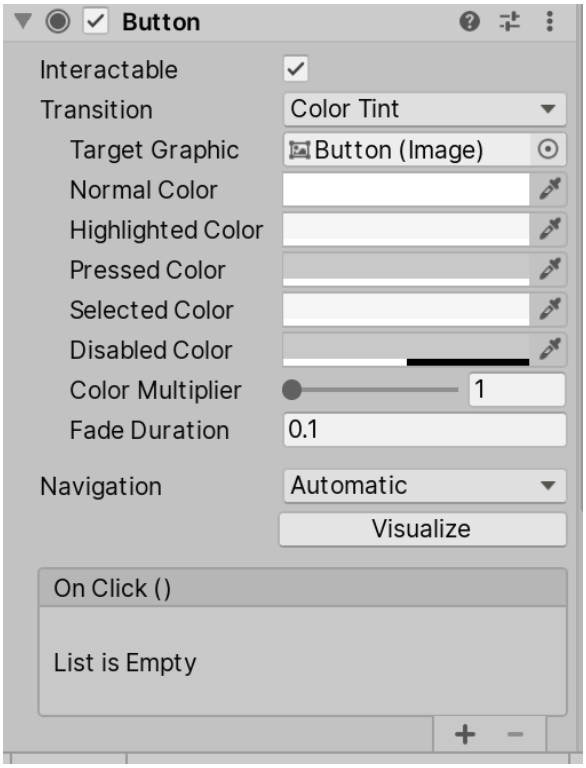
1) File->New Scene



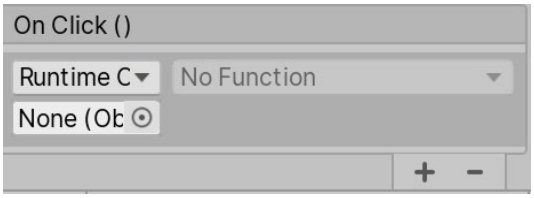
2) In the hierarchy hit the + then go to UI/Button and click



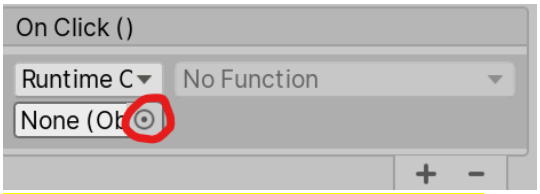
3) The button should now be selected in the hierarchy and the inspector should be populated with details like so.



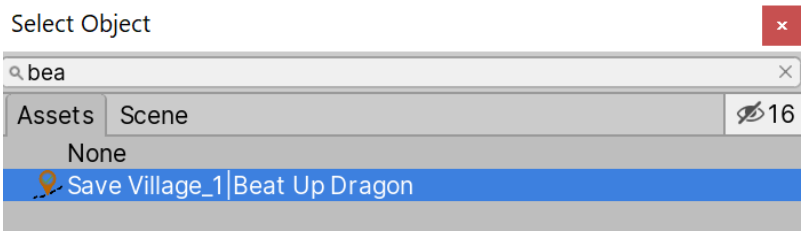
4) Click the + button



5) Click the selector icon of the Object box

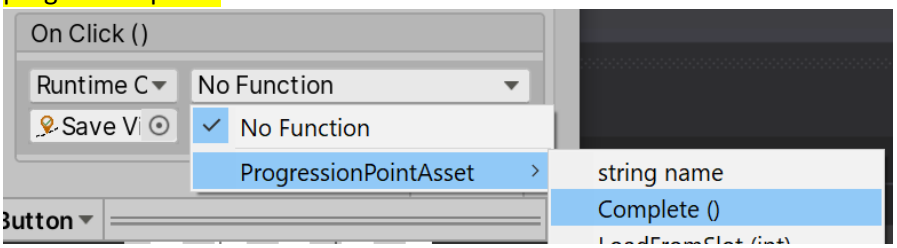


6) Get a reference to the Progression Point "Beat Up Dragon," by typing it in the search and clicking it.

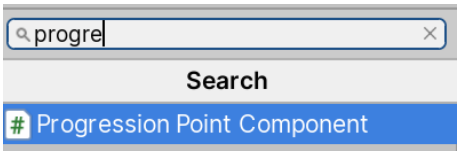


7) In the Section that says No Function, click the drop-down arrow and navigate to ProgressionPointAsset/Complete (). This complete function will attempt to complete the

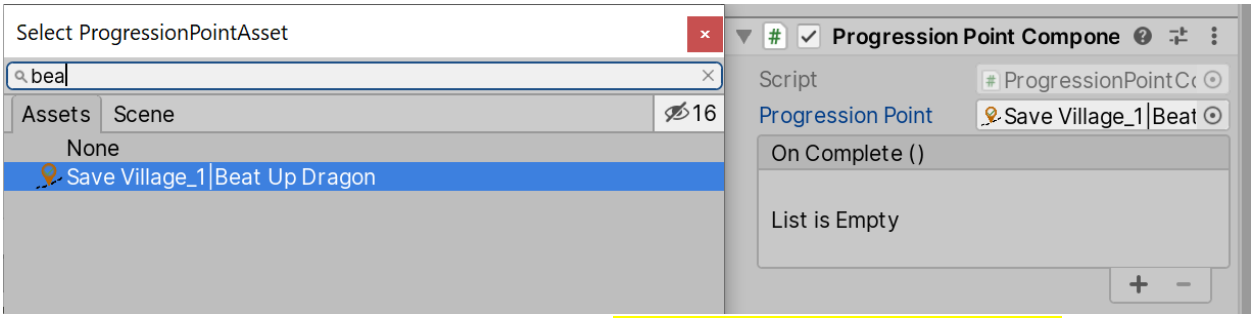
progression point.



8) To Test if this works we need to do something when the progression point completes. Add a Progression Point Component to this GameObject. Scroll down the details panel to add component, click it, type progression point and click the progression point component to add it.

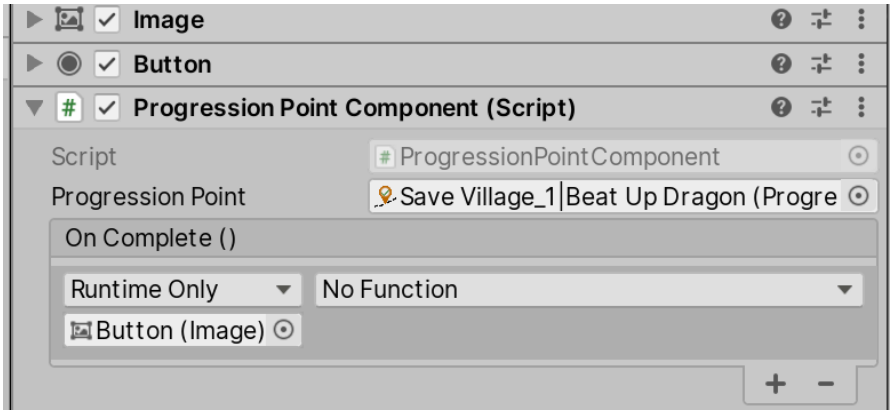


9) Click the object selector for the progression point field and select the same progress point.

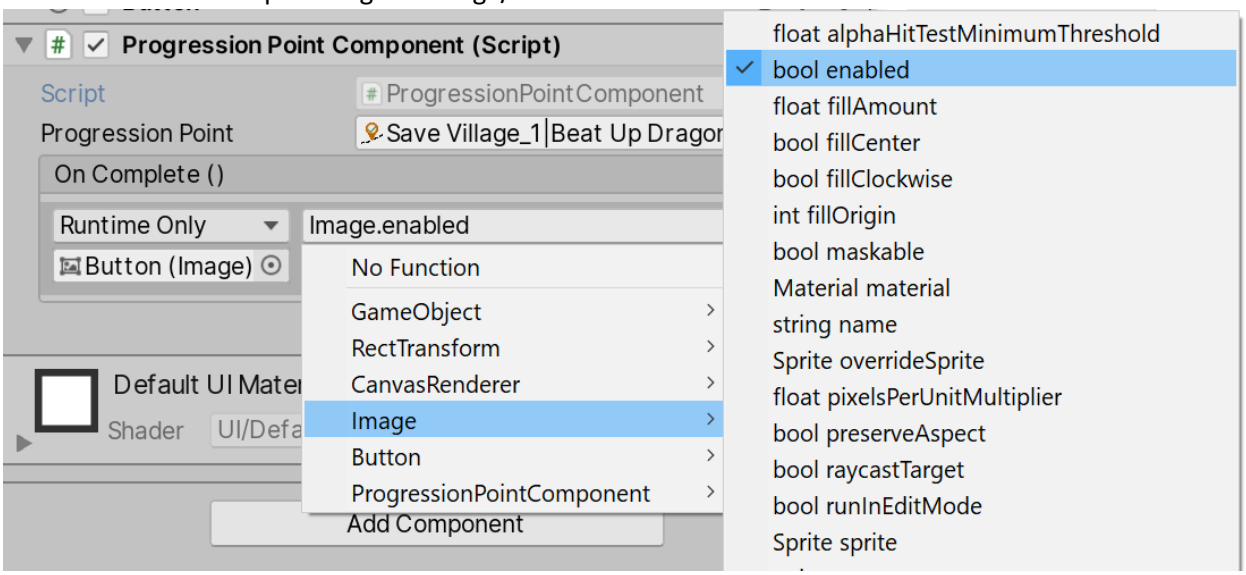


10) Click the + button for the On Complete event box. (The On Complete will only work if the progression point is the next progression point and all it's dependencies are satisfied. There are none right.)

11) In the inspector panel, click and drag the Image component into the Object field of the On Complete event.



12) In the no function drop down go to Image/bool enabled and click it.



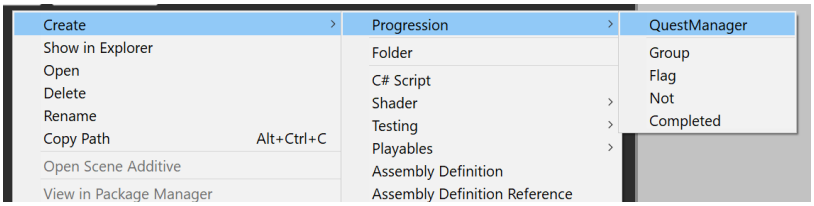
13) Hit the editor play button to test now.
14) Click the button. Notice that nothing happens. This is because we haven't completed the started point in the quest.
15) Add two more buttons and follow steps 4 through 12 for each, using the Start Progression Point for one and the Finish Progression Point for the other.
16) Hit the editor play button and notice that the buttons will now complete in order from "start" to "beat up dragon" and to "finish".
    Congratulations you have a functional quest.

Dependencies

When making a progression point you probably noticed that there is an area that says "drag and drop dependencies." Dependencies are a gating system for progression points that are agnostic to the quest, progression point and even the quest manager. This makes dependencies the ideal solution for nonlinear progression.

Example:
A player has a limited inventory, but can store items in a stationary chest.
The player has a ◆ key. Before they depart they choose to leave behind the ◆ key and take an extra set of ammo instead. On their journey, they find a ◆ door. The player tries to interact with the door. From a progress perspective, the player did achieve getting the diamond key, but they do not have it on them now. This door depends on the player having the key. Finding the key is a progression point, but having the key is a dependency.

There are a few types of dependencies. They are accessed through the create asset menu.



Flag is the quintessential “has the key” dependency. This dependency is satisfied when the flag is set to true.

Completed dependency is satisfied if the specified progression point is completed. Useful for gating quests with other quests. For example, the start progression point for one quest may be dependent on the finish progression point being completed from another quest.

Not dependency simply checks if another dependency is not satisfied.

Group dependency contains multiple dependencies. A group dependency can either be All or Any. If it is “All” it will be satisfied only if all the dependencies it contains are satisfied. If it is “Any” then it will be satisfied if one of the dependencies it contains is satisfied.

Working with Dependencies

Dependencies scale well in terms of project maintenance. They can be placed anywhere in your project. They also are extendable as well.

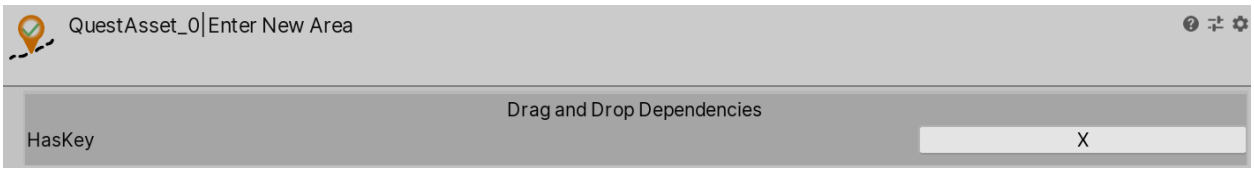
Use the create asset menu (can be found by right clicking in an empty space of the project tab, or in the menu bar under Assets->Create->Progression)

There is a dependency component which works the same way as the Progression Point Component. The difference is that instead of calling Complete(), you instead can use SatisfyIfAble(), only works if it is satisfied (ie a flag would have to be set to true). Of note, the On Satisfy Unity Event will be invoked every time Satisfy() or SatisfyIfAble() is called on the dependency as long as it is satisfied.

Workflow

Let’s say we have a simple pickup key script that when the player overlaps the key collider we disable the key in the world. On the key pickup script, we can add a Flag field. In the same place, we disable the key we can simply set the flag to true (denote that the player has the key).
Now we want to progress to the next area. Well this is where the dependencies and the progression points start to work together.

Let’s make a progression point called Enter New Area. In the drag and drop area of the progression point drag and drop the key asset. Now this Progression Point can only complete if the player has the key.



Lets add a progression point component to the door and set the progression point object field to the Enter New Area progression point.

For simplicity lets assume there is a door script on the door with an OpenDoor() function that will open the door when called.

On the progression point component hit the + button on the Completed Event and drag and drop the door component into the object field.

In the function drop down of the Completed Event select the OpenDoor() function.

We now have a basic key and lock system.

The Dependency class can be inherited from if custom behavior or logic is desired. Simply override the bool Satisfied() function. The namespace Progression will need to be added to gain access to the dependency class. Here is an example of a class that is satisfied if it has the correct string.

```
using Progression;

public class PasswordDependency : Dependency
{
    public string PasswordGuessed = "";
    public string CorrectPassword = "password";

    public override bool Satisfied()
    {
        return PasswordGuessed == CorrectPassword;
    }
}
```

Refer to the end of the saving and loading progress section for restrictions on saving and loading data.

Saving and Loading Progress

Saving and Loading is easy. To Save simply call the SaveToSlot(int slotIndex) function on any of the assets (Dependency, Progression Points, Quest, or QuestManager).
Saving the QuestManager will save all quests, progression points, and dependencies associated.
Saving a specific quest will save all progression points, and dependencies associated.

Saving a specific progression point will save all dependencies associated.

Saving a specific dependency will save all dependencies associated with it (ie a group dependency will individually save all dependencies in its group).

Loading works in much the same way. Simply call LoadFromSlot(int slotIndex).
The associations work the same way with loading.

Of note, this save system is synchronous (ie. when the function is called the data is written to disk and available immediately). This means saving or loading an enormous Quest manager (would have to be insanely large, you probably would have written your own system if it was this big) could stall for a while.

Keep in mind that without calling the loadfromslot function, the defaults will be used. The defaults are stored in slot -1. For this reason the system does not allow for saving over slot -1.

References to UnityEngine Objects work in unreliable ways.
References to UnityEngine Objects that do not change at runtime will load and save correctly.

References to UnityEngine Objects that are assigned dynamically at runtime are not guaranteed to load and save correctly.

For this reason there is an OnAssetSaved() and OnAssetLoaded() function which may be overridden OnAssetSaved is called before the asset is written to disk, and OnAssetLoaded is called after the asset is loaded. Feel free to create custom save and load logic for managing unique data that does not serialize well such as a dictionary.

Support

If you still have questions please feel free to reach out to me using the provided contact info. I am pretty quick to respond.

Name: Douglas Potesta

Email: dpolax14@aim.com

Twitter: @DougPotesta