So a Quest is created in three parts:

* Quest
* Step
* Detail

A Quest is a high level overview. It contains only a few pieces of data:

* An ID
* A title("Rescue the princess")
* A description ("We need a hero")
* A version. I plan to use this to only build quests that are out of date
* A list of Steps
* The Current Step

A Step is a more focused view. It contains several pieces of data:

* An ID
* The ID of the quest the Step belongs to
* A title of sorts ("Reach the top of the castle", "Talk to Doug", etc)
* A description, for an updating journal
* Dialogue that will be said at the beginning of the step ("Can you get me 10 chicken furs", "Follow me to the top of the castle", more complicated dialogue, etc)
* Exception Dialogue, which is dialogue said when you attempt to turn in a Step without having completed it, or when you fail a step
* Two booleans to represent the use of Generic and Typed exception dialogue. This is so that we can make some Generic stuff ("Come back later") and some stuff typed to the Action you are attempting to do (ie, a Collect Action could have Typed stuff like "You haven't picked it all up yet" or something)
* A list of rewards
* A list of reward amounts
* The amount of attempts allowed on this Step. If this is -1 (should I switch it to 0?), you can retry this Step as many times as you need
* The amount of attemps completed. If amount of attemps allowed is not infinite, and this reaches that number, the Step is failed, which may fail the quest, may open a new branch
* A list of Details

A Detail is the actual requirements to complete a Step. They are very simple things, but you can have as many Details per Step as you want. For example, an Escort quest could contain a Detail where the escorted NPC has to travel to a location, and a Detail where the player has to protect the NPC. A Detail contains this data:

* An ID
* The ID of the Step it belongs to
* An Active Entity. This is the Entity (pretty much always a Character) that will be performing the Action
* A Passive Entity. This is the Entity on the receiving end of the Action. It can be an object, character, place, anything that has a GameObject representation
* An Action, which is selected from a pre determined list. The list is currently:
  + Kill
  + Travel
  + Obtain
  + Protect
  + Destroy
  + Use
  + Have Good Relation
* An Amount, which is how many times the Action has to be completed
* An Amount Completed, which is how many times you've completed the Action
* A Timer, which is how long in in-game minutes the player has to complete the Detail
* A Date Limit, which is a best before date for the Detail, using the in-game date. Not implemented yet because we don't have a calendar in-game
* A State, which can be InProgress, Completed, or Failed

That is the data side of things, logic wise, things look like this:

When a Quest is built, it creates a quest script that is named after the quest title ("Rescue The Princess" will result in a Class and Script called "RescueThePrincess"). This script/class that inherits from Quest. This script contains a constructor which sets its values. It then calls a func called CreateFirstStep, which creates an empty Step with ID 0 and title "Quest Locked". I might remove this, I just need to see the system in action first. After that, it calls a function called CreateSteps. This function is just a line of code per step, that looks like: "Steps.Add(CreateStep(1, QuestID, "Reach the top of the castle", "kjfhdg...", "", false, false, Gold, 3000));". One line per Step. Finally, it loops through the Steps and calls CreateDetails() for each one. Tbh, this is my least favourite part. I am probably going to do some changes here. That is the whole of the generated code.

Quest.cs contains functions to complete and fail a quest (they currently do nothing, because I'm not sure what to do here), and progress a step. ProgressStep calls a func in Step.cs (CompleteStep()), iterates the CurrentStep forward (I just realized this exists in two places, I'll change that), and calls StartStep() on the NEW current step.

Step.cs contains most of the code. StartStep does nothing atm, but it should bring up dialogue, update the journal, etc. CompleteStep gives the player their rewards for completing the Step. It has funcs to complete and fail Details, which in turn can complete or fail the Step. It also has a function to reset the step (on a failure), but this is not implemented because I am unsure which direction to take for Step rebuilding (fetch the data from the database, or keep a variable with each original value?). It also contains the function to create Details (but again, I want to move this), which looks like: "Details.Add(CreateDetail(12, 1, Player, 5, -1, -1, Doug));". That goes DetailID, StepID, Active, Action, Timer, DateLimit, Passive. The Action number is the index from the Action list.

Detail.cs is a small class. It has functions to create a timer, and a callback function which fails the Detail (and notifies the Step). It also creates an observer for the passive entity takes in the Action, and should then trigger on the Action being completed, and a callback function. That observer callback ticks up AmountCompleted, checks if is equal to Amount, and if it is notifies the Step that the Detail is complete (as well as stopping the timer and changing its state).

The Quest Creator is basically a GUI that lets you make Quests (and customize their name and description, make Steps within a Quest (customizing their name, dialogue, exception dialogue, generic and typed exception use, reward and reward amount), and make Details within Steps (customizing the entities, action, amount, and timers). It then pushes this data to a DB. It then has a button that will build each quest using that DB’s data, which generates a single script to be placed somewhere in Unity.

Currently on my //TODO list:

* I want to move CreateDetail out of Step and into the generated file
* Fix that redundant variable, refactor my database manager (also host the DB online)
* Add some pseudo code for a Date limit
* Add pseudo code for updating a journal
* Update my Detail creator to take in a number of allowed attempts
* Update my Step creator to better allow a list of items as a reward